THOMSON
COURSETECHNOLOGY
Professional ■ Trade ■ Reference

BEGINNING DIRECTX 9

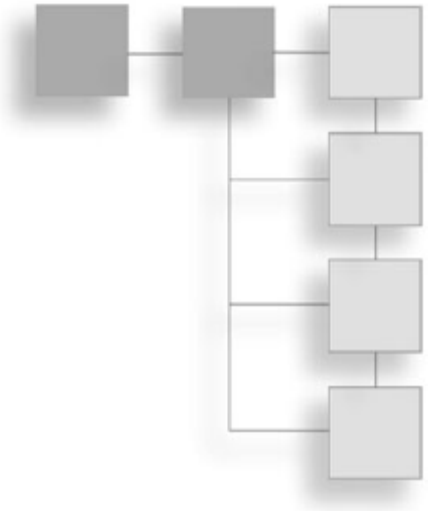
WENDY JONES

TRANSLATED BY VNGAMEDEV TRANSTEAM

ISBN: 1-59200-349-4
Library of Congress Catalog Card Number: 2004090736
Printed in the United States of America
04 05 06 07 08 BH 10 9 8 7 6 5 4 3 2 1

Course PTR, a division of Course Technology
25 Thomson Place
Boston, MA 02210
http://www.courseptr.com

Senior Vice President, Course PTR Group:
Andy Shafran
Publisher:
Stacy L. Hiquet
Senior Marketing Manager:
Sarah O'Donnell
Marketing Manager:
Heather Hurley
Manager of Editorial Services:
Heather Talbot
Senior Acquisitions Editor:
Emi Smith
Associate Marketing Manager:
Kristin Eisenzopf
Project/Copy Editor:
Karen A. Gill
Technical Reviewer:
Joseph Hall
Retail Market Coordinator:
Sarah Dubois
Interior Layout:
Marian Hartsough
Cover Designer:
Steve Deschene
CD-ROM Producer:
Brandon Penticuff
Indexer:
Sharon Shock
Proofreader:
Sean Medlock



Thông tin về tác giả

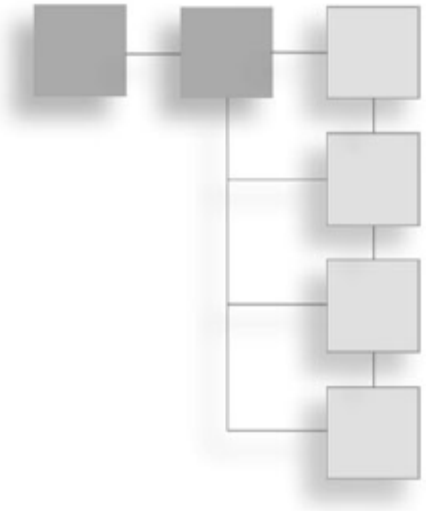
WENDY JONES là người đã dành trọn niềm đam mê của cô ấy cho máy tính ngay từ lần đầu tiên cô được nhìn thấy chiếc máy Apple IIe khi còn học ở trường phổ thông. Kể từ khi tiếp cận với nó, cô ấy đã dành mọi khoảng thời gian rảnh rỗi vào công việc học lập trình BASIC và kỹ thuật đồ hoạ, vẽ những ý tưởng của mình lên giấy vẽ và thực hiện nó trên máy tính. Ngoài BASIC thì cô cũng đã lần lượt tiếp cận với các ngôn ngữ khác như Pascal, C, Java và C++.

Nghề nghiệp của Wendy là một chuỗi thay đổi như chính sự phát triển của máy tính vậy, từ bỏ môi trường lập trình DOS, cô tự trang bị cho mình những kiến thức về lập trình Window và sao đó rất nhanh chóng nhảy sang thế giới .NET. Mặc dù các công ty Internet đem lại rất nhiều tiền, nhưng đó không phải là tất cả, chính vì thế Wendy đã mở rộng kỹ năng lập trình của mình sang lĩnh vực lập trình games và dồn tất cả sức lực của mình cho niềm đam mê đó.

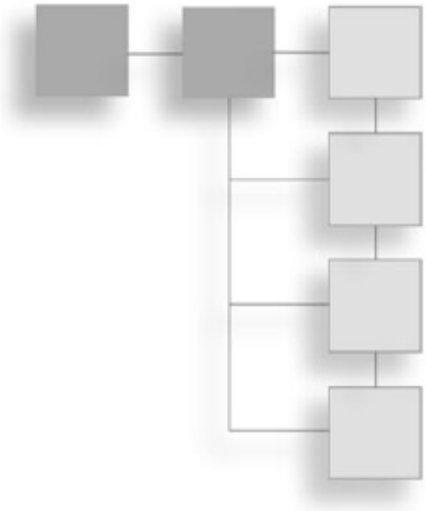
Wendy đã hiện thực hoá được niềm đam mê này khi cô nhận được một đề nghị làm việc cho công ty Atari's Humongous Entertainment với tư cách là một lập trình viên. Trong xuốt quá trình làm việc ở Atari, cô ấy đã làm việc trên cả hai môi trường PC và máy chơi game cá nhân, bị quấn theo hàng loạt những thách thức mà công việc đó đem đến.

Hiện tại Wendy đang làm việc các phần mềm dành cho PocketPC và trên những thiết bị chơi game cầm tay.

Nếu bạn có bất cứ góp ý hay những câu hỏi về quyển sách này, bạn có thể liên hệ với cô theo địa chỉ mail: gadget2032@yahoo.com



Bố cục của sách



LỜI GIỚI THIỆU

Lập trình game có thể nói là một công việc cực kỳ thú vị trong thế giới lập trình bởi nó đem lại cho bạn cơ hội được tạo ra một thế giới ảo nơi mà những sinh vật huyền bí, những vùng đất chưa hề được biết đến ngay cả trong những giấc mơ. Liệu bạn có thể tìm thấy điều này ở một nơi nào khác? Với nó, bạn có thể đem đến cho mọi người khả năng trở thành bất cứ một nhân vật nào, một con người nào mà họ mong muốn và hơn cả là một môi trường để họ có thể sống theo đúng sở thích.

Cũng chính bởi thế mà ngành công nghiệp Game đã phát triển không ngừng, vượt qua mọi giới hạn và kéo theo nó là phát triển của công nghệ phục vụ cho nó. Chỉ vài năm trước đây, có lẽ ở cấp độ người sử dụng còn chưa hề biết đến các thiết bị thể hiện hình ảnh 3D ngoại trừ các máy trạm đắt tiền SGI dựa trên nền tảng OpenGL. Tại thời điểm đó có thể nói OpenGL mới chỉ đang ở thời kỳ phôi thai. Đến khi các máy tính cá nhân PC đã được phổ biến hơn, OpenGL đã mở rộng nền tảng của mình và điều đó thực sự đã đem đến một cú huých mạnh đầu tiên trong lịch sử phát triển game cũng như những ứng dụng mô phỏng không gian 3D.

Tại thời điểm đó, Windows vẫn chưa phải là nền tảng tốt nhất để phát triển games, tuy nhiên họ cũng rất nhanh chóng nhận ra những lợi nhuận từ ngành công nghiệp này. Chính vì thế phiên bản DirectX đầu tiên cũng sớm được Microsoft tung ra. Tuy nhiên trong thời gian đầu này thì sự phổ biến của nền tảng này vẫn còn rất hạn chế do OpenGL đã được thừa nhận là phương thức chuẩn để phát triển đồ hoạ 3D trên môi trường Windows. Ngày nay, hầu hết các game phát triển trên PC bán trên thị trường đều được xây dựng trên nền tảng DirectX, nó đã giúp cho những game thủ có thể tận hưởng được một công nghệ đồ hoạ tiên tiến và hiện thực nhất về thế giới.

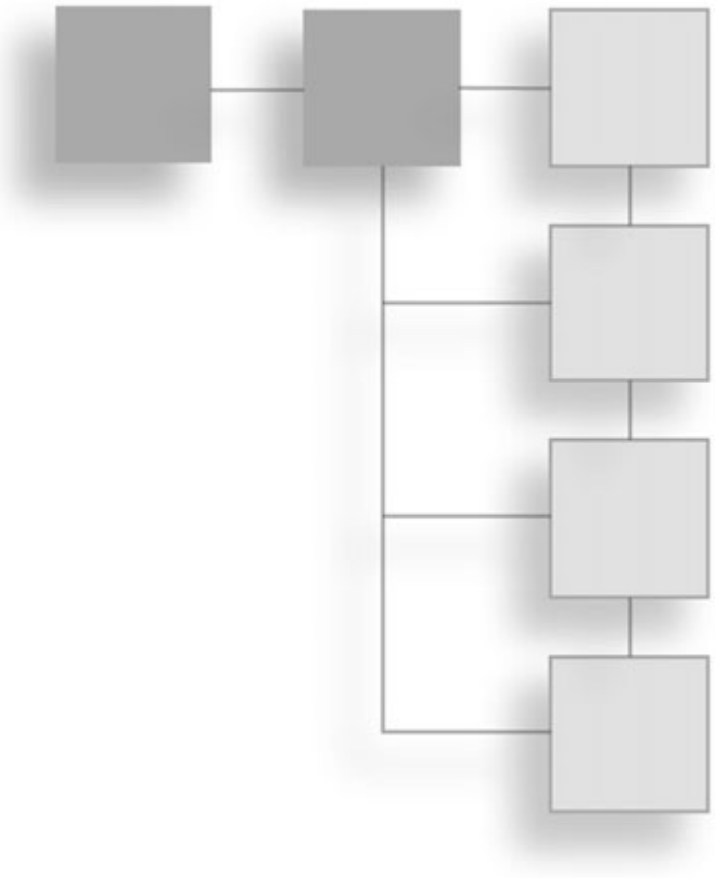
Những cần phải có những gì để tiếp cận công nghệ này?

Những kinh nghiệm và kiến thức về lập trình C++ cũng như lý thuyết lập trình hướng đối tượng (OOP) sẽ giúp bạn có thể hiểu được tất cả những bài giảng được trình bày trong quyển sách này. Ngoài ra bạn cũng nên trang bị thêm cho mình một vài kỹ năng về toán học và hình hoạ, tuy nhiên hầu hết những khái niệm toán học sử dụng cũng sẽ được giải thích chi tiết. Những hiểu biết về Visual Studio .NET hay bất kỳ sản phẩm nào trong bộ Visual Studio cũng sẽ rất là hữu ích.

Làm thế nào để sử dụng quyển sách này hiệu quả?

Quyển sách này được chia làm 3 phần chính. Phần đầu tiên sẽ mô tả cở bản về DirectX, làm thế nào để thiết lập và chạy một ứng dụng sử dụng nền tảng này. Phần thứ hai sẽ cung cấp cho bạn những khái niệm cơ bản phục vụ quá trình thiết kế và xây dựng một môi trường 3D, kèm theo đó là giới thiệu những khái niệm về 3D và Direct3D. Phần 3 và phần cuối sẽ xoay quanh các kiến thức khác của nền tảng DirectX như xử lý âm thanh thông qua DirectSound, lấy các thông tin đầu vào từ người dùng thông qua DirectInput. Quyển sách kết thúc bằng một dự án minh hoạ nhỏ với hi vọng đem đến cho bạn một cái nhìn toàn cảnh hơn về tất các các kiến thức chúng ta đã học và áp dụng cụ thể chúng như thế nào.

Nếu bạn đã từng làm quen với môi trường DirectX và cũng đã từng viết một vài ứng dụng sử dụng nền tảng này, bạn có thể bỏ qua phần một. Tôi nghĩ rằng bất kỳ ai khi mới tiếp cận với môi trường lập trình game và DirectX nên đọc quyển sách này một cách tuần tự để có thể hiểu hết được những khả năng mà DirectX có thể đem lại.



PHẦN I

KIẾN THỨC NỀN TẢNG

Chương 1.

DirectX là gì, tại sao và làm thế nào để sử dụng nó

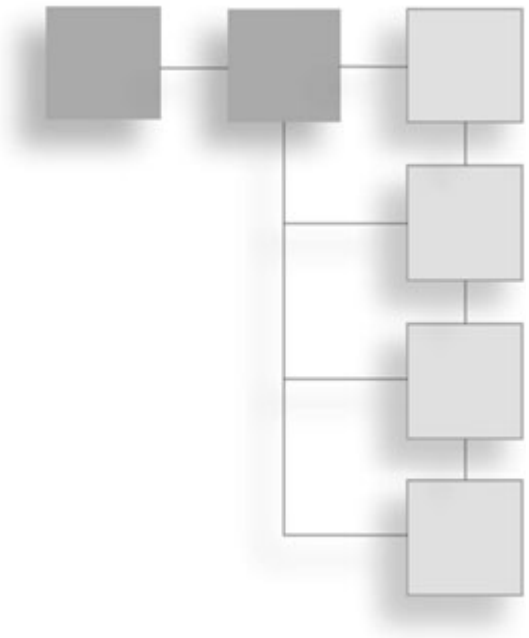
Chương 2.

Ví dụ đầu tiên sử dụng DirectX

Chương 3.

Phông nền, khung hình, hoạt cảnh.

CHƯƠNG 1



DirectX là gì, Tại sao và Làm thế nào để sử dụng nó

DirectX về thực chất là một hệ thống các giao diện lập trình ứng dụng (API) phục vụ cho quá trình phát triển các game trên nền tảng hệ điều hành Windows. Nếu như tại thời điểm một vài năm trước, những người phát triển game thường phải vật lộn với những vấn đề nảy sinh từ sự không tương thích của phần cứng - không thể phát triển những sản phẩm game để tất cả mọi người đều có thể tận hưởng được những tính năng của game đó. Thì ngày nay, Microsoft đã đem tới một nền tảng DirectX. Nó cung cấp cho những người phát triển game một phương thức đơn nhất, một nền tảng thư viện API "sạch" (đã được chuẩn hoá) để họ có thể viết và chạy trên hầu hết các nền tảng mà không cần phải lo ngại nhiều về phần cứng của PC. Chỉ một vài năm sau khi phiên bản DirectX được đề xuất, số lượng game được phát triển trên nền tảng Windows đã tăng lên một cách nhanh chóng.

Trong chương này, chúng ta sẽ đề cập tới các vấn đề:

- DirectX là gì vậy
- Tại sao nó lại hữu dụng đến thế
- Thành phần nào đã xây dựng nên nền tảng DirectX API

DirectX là gì?

DirectX là một tập hợp thư viện các hàm API được Microsoft thiết kế để cung cấp cho những người phát triển game một giao diện lập trình cấp thấp để liên kết tới các phần cứng của PC chạy trên hệ điều hành Windows. Phiên bản hiện tại của nó là 9.0(c), mỗi một đối tượng API của DirectX cung cấp một khả năng truy cập khác nhau tới từng loại phần cứng của hệ thống, nó bao gồm hệ thống đồ hoạ, âm thanh và kết nối mạng, tất cả chúng đều được xây dựng trên một giao diện chuẩn. Giao diện này cho phép những người phát triển có thể viết những game của họ bằng cách sử dụng tập hợp cách hàm này lại với nhau mà không cần phải quan tâm nhiều tới phần cứng mà nó sẽ được sử dụng.

Những thành phần tạo nên DirectX

Hệ thống thư viện API của DirectX được phân chia ra làm nhiều thành phần nhỏ, mỗi thành phần này đảm nhiệm những nhiệm vụ khác nhau của hệ thống. Chúng có thể được sử dụng hoàn toàn độc lập với nhau, bạn có thể chỉ cần thêm những thành phần nào mà game của bạn cần tới mà thôi. Sau đây là danh sách các thành phần đó:

- **DirectX Graphic**. Thành phần này đảm nhiệm tất cả các chức năng kết xuất đồ hoạ của hệ thống. Nó cung cấp những hàm API để người dùng có thể quản lý quá trình vẽ 2D cũng như 3D, ngoài ra nó cũng hỗ trợ cả quá trình khởi tạo và xác lập độ phân giải cho game của bạn.
- **DirectInput**. Tất cả những gì người dùng nhập vào sẽ được quản lý bởi các hàm API trong thành phần này. Nó bao gồm khả năng hỗ trợ các thiết bị như bàn phím, chuột, gamepad, joystick. Với phiên bản mới nhất hiện nay cũng đã hỗ trợ cả các thiết bị tương tác thực (force-feedback) như cần lái, chân ga cho các game đua tốc độ.
- **DirectPlay**. Khả năng kết nối mạng được cung cấp cho những hệ thống game của bạn thông qua thành phần DirectPlay này. Những hàm phục vụ kết nối này đem đến cho bạn khả năng phát triển ứng dụng có thể giao tiếp với bất kỳ một máy nào khác, cho phép không chỉ một người có thể chơi game đó. DirectPlay đem đến cho bạn một giao diện lập trình cấp cao nhằm giúp bạn tránh khỏi những vấn đề khó khăn trong quá trình lập trình ứng dụng liên kết mạng.
- **DirectSound**. Nếu bạn muốn chèn thêm các hiệu ứng âm thanh hoặc nhạc nền, bạn sẽ cần phải sử dụng tới những hàm API do thành phần này cung cấp. Chức năng của DirectSound là cho phép bạn có thể tải và chơi một hoặc nhiều file nhạc dạng WAV cũng như toàn bộ khả năng điều khiển quá trình chơi nhạc đó.
- **DirectMusic**. Thành phần này mang đến cho bạn khả năng tạo các bản nhạc "động" (tại thời điểm chương trình chạy, không cần phải lưu trữ trong cơ sở dữ liệu của game). Nó có khả năng hỗ trợ hầu hết các chức năng chính của một phần mềm chơi nhạc midi như tự chạy lại theo một qui trình đã xác lập, biến đổi âm lượng để phù hợp với hoạt cảnh trong game, thay đổi nhịp của giai điệu và nhiều chức năng cao cấp khác.
- **DirectShow**. Bạn có thể cắt một hoạt cảnh phim hoặc lồng ghép âm thanh và game của mình thông qua thành phần này. AVI, MP3, MPEG và ANF chỉ là một trong số rất nhiều các loại định dạng file mà DirectShow hỗ trợ. Với DirectShow, bạn sẽ không phải tải tất cả các đối tượng file này lên bộ nhớ, thay vào đó là truy cập trực tiếp dữ liệu trên ổ cứng hoặc CD-ROM.
- **DirectSetup**. Sau khi game của bạn đã hoàn thành, bạn sẽ muốn phát hành. DirectSetup cung cấp cho bạn những chức năng giúp ứng dụng có thể tự động cài đặt những phiên bản DirectX mới nhất lên hệ thống của người sử dụng.

Chú ý:

Hệ thống đồ hoạ DirectX bao gồm toàn bộ các chức năng của cả DirectDraw và Direct3D. Phiên bản DirectX 7.0 là phiên bản cuối cùng phân tách hai thành phần này thành hai đối tượng độc lập.

Tại sao DirectX lại cần thiết?

Trước khi hệ điều hành Windows được phát hành, các lập trình viên chủ yếu phát triển các game trên nền tảng hệ điều hành DOS. Đấy là một hệ điều hành đơn nhiệm, không hỗ trợ giao diện đồ hoạ (non-GUI) và cách duy nhất để thực hiện các thao tác đồ hoạ này là truy cập trực tiếp vào phần cứng và thực thi chúng. Điều này đem đến cả những lợi ích cũng như những vấn đề khó khăn cần giải quyết. Ta có thể ví dụ ra, bởi vì sử dụng cách truy cập trực tiếp giữa mã game và phần cứng, lập trình viên có thể có được toàn bộ sức mạnh cũng như quyền điều khiển của hệ thống, điều nay đem đến khả năng hoạt động với hiệu suất và chất lượng cao của hệ thống game. Tuy nhiên, mặt trái của vấn đề là những nhà phát triển game này sẽ phải tự xây dựng cho mình những thư viện hỗ trợ cho từng loại thiết bị phần cứng nếu họ muốn game có khả năng hỗ trợ. Nó bao gồm ngay cả những thiết bị rất phổ thông như card âm thanh và đồ hoạ.

Tại thời điểm đó, không phải tất cả các thiết bị đều tuân theo một chuẩn nhất định. Chính điều này đã làm cho quá trình thao tác trên màn hình đồ hoạ cực kỳ khó khăn đặc biệt khi bạn muốn phát triên các game có độ phân giải trên 320 x 240 pixel. Chỉ việc viết các game sử dụng độ phân giải 640 x 480 cũng đã khiến các lập trình viên phải truy cập trực tiếp vào bộ nhớ video và tự mình quản lý toàn bộ quá trình cập nhật các thanh ghi sử lý của card đồ hoạ đó. Và họ đã nghĩ rằng đã đến lúc cần phải tìm kiếm một cách đi khác đơn giản và hiệu quả hơn.

Khi mà phiên bản Windows 3.1 chính thức phát hành, những vấn đề đó vẫn chưa hề được giải quyết. Hơn thế, bởi vì Windows chạy trên nền DOS nên chính nó vô tình đã làm giảm đi tài nguyên hệ thống giành cho các game và nó cũng đã loại bỏ toàn bộ khả năng truy cập trực tiếp mà những nhà phát triển game vẫn hay dùng trước đó từ rất lâu rồi. Vậy là hầu hết các game hỗ trợ Window tại thời điểm đó chủ yếu là những game đánh bài và các game đơn giản khác. Trong khi đó các game hành động vẫn tiếp tục được phát triển trên nền tảng DOS.

Mặc dù trước đó, Microsoft đã cố gắng cung cấp cho người phát triển một cách cách truy cập nhanh hơn tới các thiết bị đồ hoạ thông qua hệ thống thư viện có tên WinG. Nó là tiền thân của DirectX và chỉ hỗ trợ rất ít chức năng. WinG thực sự là một ý tưởng hay, tuy nhiên nó vẫn chưa thể đem tới cho những người phát triển khả năng truy cập hệ thống tốt hơn so với những cách mà họ vẫn thích làm trên nền hệ điều hành DOS.

Chính vì thế, Microsoft đã có những cải tiến và bổ xung thêm khá nhiều chức năng trong phiên bản đầu tiên DirectX, hay còn được biết tới với cái tên GameSDK (bộ công cụ phát triển game). DirectX đã cung cấp cho các lập trình viên một thư viện độc lập để phát triển, nó là lớp trung gian giữa game và các phần cứng của PC. Lúc này thì quá trình thao tác đồ hoạ đã trở nên dễ dàng hơn rất nhiều. Tuy nhiên phiên bản đầu tiên của DirectX vẫn chưa hỗ trợ được nhiều dạng phần cứng khác nhau, nhưng đó thực sự đã là một thay đổi rất có ý nghĩa và thực sự đáp ứng được mong muốn của những người phát triển game. Chỉ một vài năm sau, họ đã phát hành phiên bản DirectX 9, mỗi một phiên bản đều đã được bổ xung và tăng khả năng hỗ trợ những công nghệ mới như kết nối mạng, kết xuất âm thanh, hình ảnh và hỗ trợ rất nhiều dạng thiết bị đầu vào mới.

Các thành phần của DirectX được kết hợp lại như thế nào

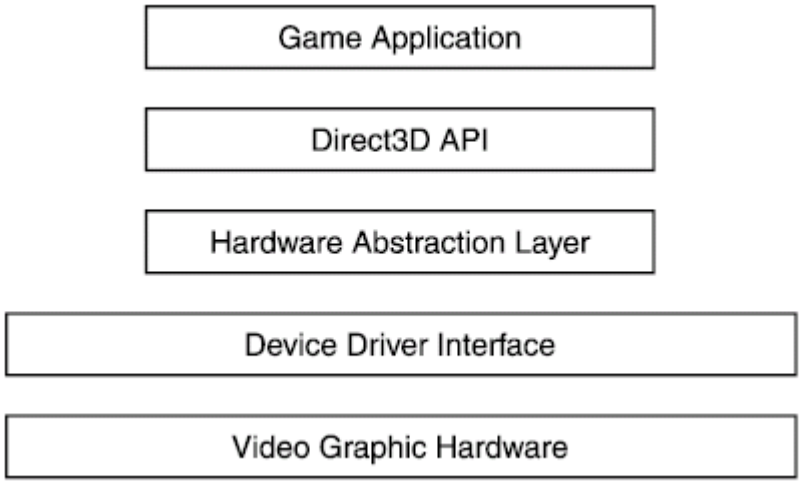
DirectX đem đến cho những lập trình viên khả năng phát triển các game trên môi trường Window một cách đơn giản hơn. Đó là một hệ thống thư viện được tổ chức dễ hiểu hơn với sự hỗ trợ đầy đủ của Microsoft, những người cuối cùng cũng nhận ra sự quan trọng và những lợi nhuận có thể đem lại từ ngành công nghiệp giải trí game trên máy tính.
Qua từng năm, các phiên bản của DirectX đã dần dần được bổ xung, phát triển thêm nhằm cung cấp sự hỗ trợ ngày càng nhanh hơn cũng như cho những phần cứng mới hơn. Microsoft muốn đảm bảo rằng mỗi một thành phần của DirectX có khả năng hỗ trợ các game đã được viết cho phiên bản trước sẽ không gặp bất kỳ vấn đề gì khi phiên bản mới được cài đặt vào hệ thống. Để thực hiện được điều này, Microsoft đã thiết kế DirectX trên nền tảng COM.

Mô hình đối tượng thành phần (COM – Component Object Model)

Các hàm DirectX API được thiết kế trên nền tảng *Component Object Model* (COM). Đối tượng *COM* bao gồm tập hợp các giao diện nhằm cung cấp các phương thức cho những người lập trình có thể truy cập DirectX. Đối tương COM có thể hiểu nôm na là các tệp tin DLL đã được đăng ký với hệ thống. Đối với những đối tượng DirectX COM, quá trình này sảy ra khi DirectX được cài đặt vào hệ thống. Mặc dù nó khá giống với các đối tượng C++, nhưng các đối tượng COM cần thiết phải có một giao diện để truy cập tới các phương thức bên trong nó. Đây thực sự là một thuận lợi so với cách sử dụng các đối tượng thông thường bởi vì ta có thể xây dựng rất nhiều kiểu giao diện cho một đối tượng COM, nó rất thuận lợi cho việc tương thích ngược (hỗ trợ các ứng dụng đã xây dựng trước đó).
Điều này được thể hiện, mỗi một phiên bản của DirectX đều chứa những giao diện mới của DirectDraw và nó có thể truy cập thông qua các hàm API, ngoài ra nó vẫn chứa phiên bản trước đó nên nó không hề tác động tới những mã lệnh đã được viết trước đó. Những game được viết để sử dụng Direct7 hoàn toàn có thể hoạt động với Direct9 mà không gặp bất kỳ vấn đề gì.
Thêm một lợi điểm nữa của kỹ thuật này là các đối tượng COM có khả năng hoạt động với rất nhiều ngôn ngữ khác chứ không phải chỉ bằng C++. Người lập trình có thể sử dụng Visual Basic, C++ hay C# mà vẫn có thể sử dụng cùng một thư viện DirectX.

Kiến trúc của DirectX

Nền tảng của DirectX bao gồm 2 lớp: lớp hàm API và lớp (giao diện) thiết bị (HAL – Hardware Abstraction Layer). Các hàm trên lớp API sẽ kết nối tới phần cứng thông qua lớp HAL. Lớp HAL cung cấp một giao diện đã được chuẩn hoá cho DirectX, ngoài ra nó có khả năng "nói chuyện" trực tiếp với phần cứng thông qua các xác lập đối với các loại thiết bị điều khiển của chính nó. Và bởi vì HAL cần phải biết cách hoạt động của phần cứng cũng như của các thiết bị điều khiển nên HAL được viết bởi chính các nhà phát triển phần cứng. Bạn không bao giờ phải kết nối tới HAL một cách trực tiếp trong quá trình viết game của mình. Thay vào đó chúng ta sẽ sử dụng kết nối gián tiếp thông qua các hàm mà DirectX cung cấp. Hình minh hoạ 1.1 thể hiện sơ đồ minh hoạ sự phân lớp giữa Direct3D và các thiết bị điều khiển phần cứng.



Hình 1.1 – Mô hình phân lớp DirectX

Trong các phiên bản trước, DirectX còn được phân chia lớp HAL ra thành một lớp khác nữa với tên gọi là *Lớp mô phỏng phần cứng* (HEL – Hardware Emulation Layer) hay thiết bị RGB. Lớp HEL sẽ mô phỏng một vài chức năng mà thiết bị phần cứng còn thiếu. Điều này cho phép cả những thiết bị rẻ tiền cũng như đắt tiền đều có khả năng cung cấp những chức năng giống nhau cho hàm DirectX API. Mặc dù các chức năng mà HEL cung cấp là giống nhau nhưng hiệu suất cũng như chất lượng sẽ không bằng. Bởi vì HEL thực hiện các chức năng này bằng phần mềm, thông thường thì tốc độ khung hình thể hiện sẽ thấp hơn so với chức năng được xây dựng sẵn trên phần cứng.

Lớp HEL này đã được thay đổi thành kiểu thiết bị phần mềm có thể cắm và chạy (tương tự thiết bị phần cứng Plug&Play). Thiết bị này thực hiện quá trình hiển thị thông qua các hàm được người lập trình viết nên. Hầu hết những game được thương mại đều không còn sử dụng kỹ thuật này nữa mà thay vào đó là yêu cầu thiết bị hiển thị hỗ trợ 3D bởi vì gần như tất cả các máy tính hiện nay đều hỗ trợ nó.

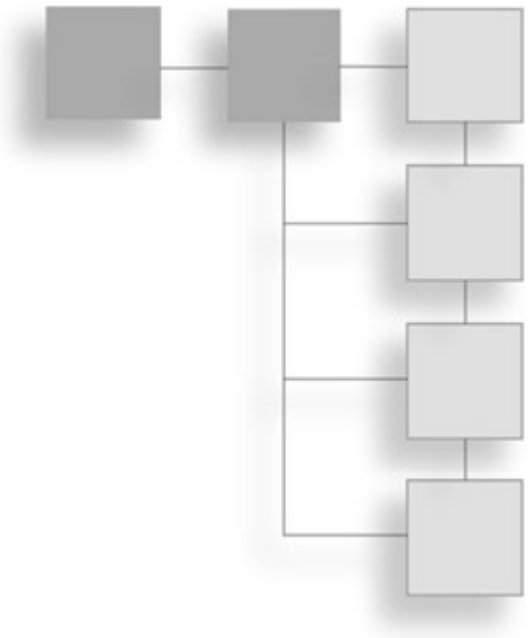
Chú ý:

> Bộ công cụ hỗ trợ phần cứng của DirectX (DDK – Device Driver Kit) cung cấp những thông tin cần thiết giúp bạn có khả năng tự phát triển một thiết bị cắm chạy được mô phỏng bằng phần mềm.

Tổng kết chương

Trong chương vừa rồi chúng ta đã tìm hiểu về những công nghệ ẩn chứa bên dưới những hình ảnh cũng như trả lời câu hỏi DirectX là gì, tại sao và sự cần thiết của nó như thế nào. Trong chương tiếp theo chúng ta sẽ đi sâu hơn vào những vấn đề kỹ thuật, giúp bạn có thể từng bước xây dựng được những ứng dụng đầu tiên sử dụng công nghệ DirectX.

CHƯƠNG 2



CHƯƠNG TRÌNH ĐẦU TIÊN

Trong phần này sẽ trình bày một ví dụ nhỏ sử dụng DirectX. Chúng ta sẽ từng bước đi qua tất cả những tiến trình cần thiết cho một chương trình sử dụng công nghệ DirectX. Hầu hết các ví dụ đi kèm với bộ công cụ hỗ trợ phát triển ứng dụng DirectX (DirectX SDK) được đặt trong thư mục sample, thư mục này chứa tập hợp rất nhiều mã nguồn và làm bạn sẽ phải tốn rất nhiều thời gian công sức để tìm hiểu nó. Trong những ví dụ và hướng dẫn chúng ta sẽ làm theo dưới đây, mặc dù những flatform đó có thể nó đã được viết sẵn trong ví dụ đi kèm của bộ SDK nhưng chúng ta sẽ không sử dụng lại. Điều đó sẽ giúp bạn hình dung được một cách rõ ràng hơn những công việc bạn phải làm để biến những ý tưởng của mình thành những game thực sự.

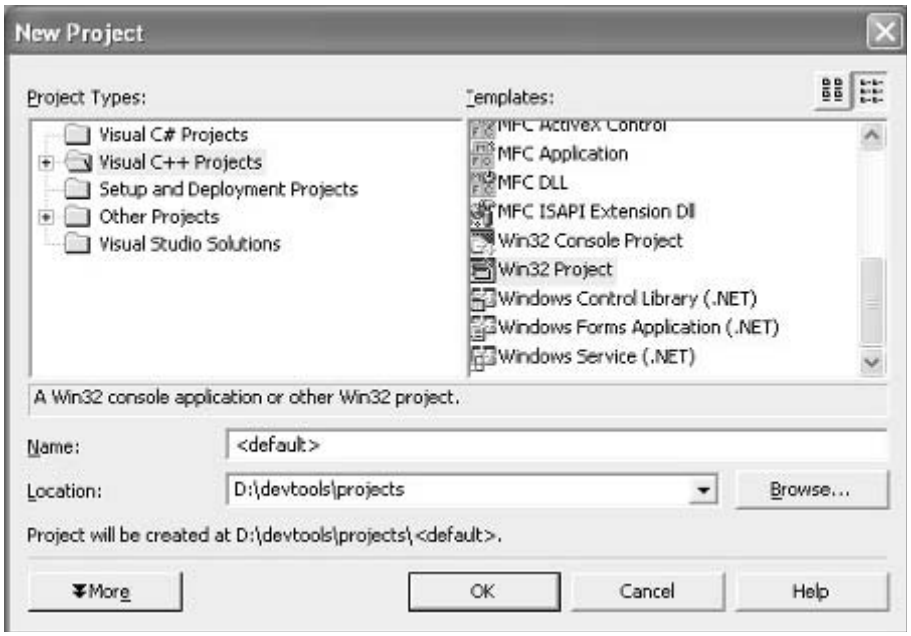
Dưới đây là nội dung những phần chúng ta sẽ xem xét trong chương này:

- Làm thế nào để tạo một dự án mới
- Xác lập các thông số cho ứng dụng
- Làm thế nào để khởi tạo ứng dụng với DirectX
- Làm thế nào để xoá màn hình ứng dụng
- Làm thế nào để thể hiện nền cho ứng dụng
- Làm thế nào để game của bạn phóng lên toàn màn hình
- Làm thế nào đề nhận biết khả năng hỗ trợ, cấu hình của hệ thống

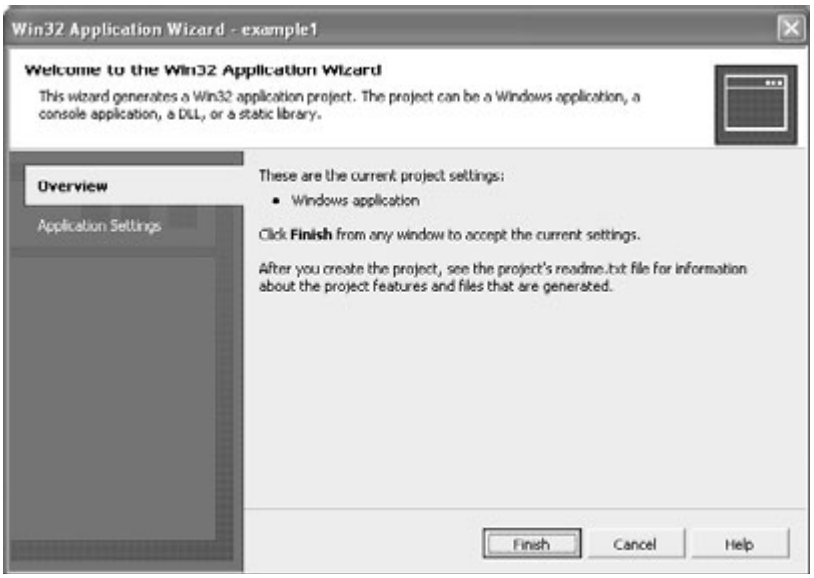
Xây dựng một dự án mới.

Bước đầu tiên bạn cần làm cho bất kỳ một ứng dụng nào là tạo một project mới trong Visual Studio. Bạn hãy chạy Visual Studio .Net với chế độ không mở bất kỳ một project nào.

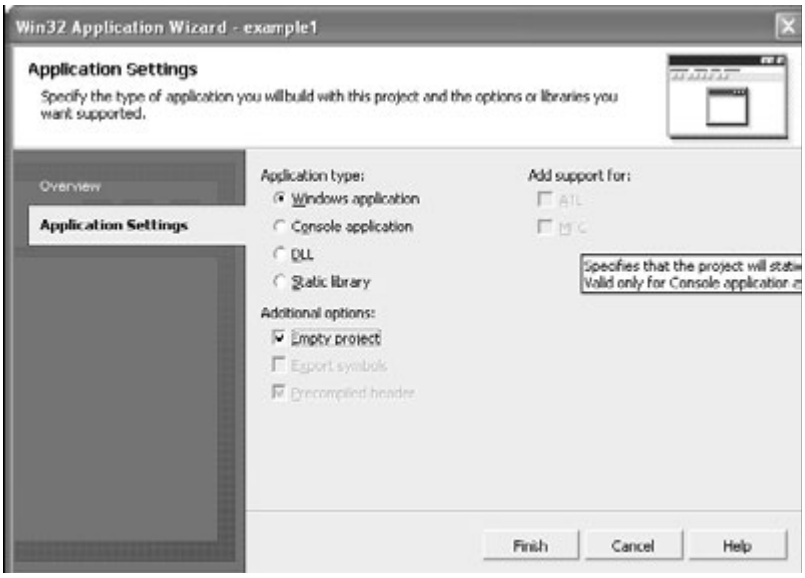
1. Lựa chọn New, Project từ thực đơn File, hộp thoại New Project sẽ xuất hiện có dạng tương tự hình minh hoạ 2.1 dưới đây.



2. Thay đổi tên của project thành example1 và lựa chọn loại ứng dụng là Win32 Project từ danh sách các project mẫu (project templates). Kích chọn nút OK để hoàn tất, một hộp thoại có tên Application Wizard sẽ xuất hiện với 2 tabs cho bạn lựa chọn và xác lập là: Overview, Application Settings. (hình minh hoạ 2.2)



3. Lựa chọn Application Settings tab và chắc chắn rằng lựa chọn Empty Project (tạo một dự án rỗng) đã được tích chọn, hình minh hoạ 2.3.



4. Kích chọn nút Finish để kết thúc.

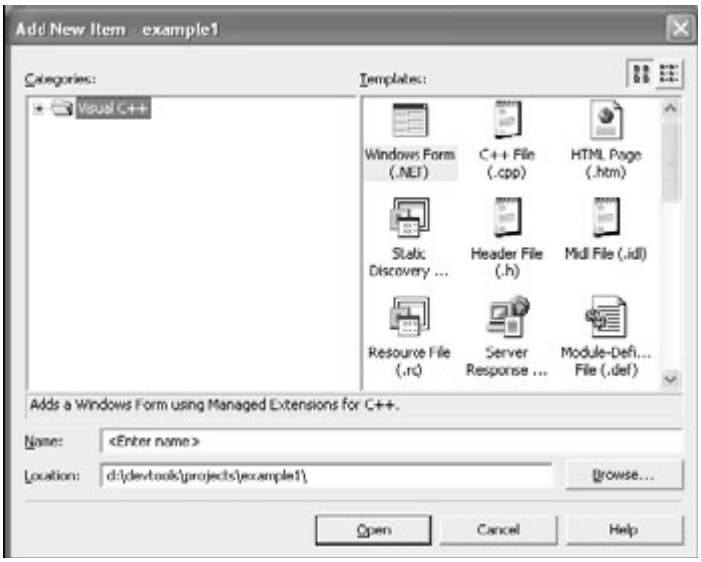
Chèn thêm mã lệnh cho chương trình

Sau khi bạn thực hiện các bước trên, Visual Studio đã tạo ra một project mới cho bạn tuy nhiên nó vẫn chưa chứa bất kỳ thứ gì cả. Bước tiếp theo, chúng ta sẽ tiến hành thêm mã lệnh để khởi tạo cho một ứng dụng trong môi trường window. Bạn hãy bắt đầu bằng cách chèn thêm một tệp tin mới chứa mã nguồn và dự án.

1. Lựa chọn Add New Item từ thực đơn Project. Một hộp thoại tương tự hình minh hoạ 2.4 phía dưới đây sẽ xuất hiện.

2. Lựa chọn kiểu file C++ File (.cpp) từ danh sách kiểu tệp tin mẫu (Templates).

3. Thay đổi tên tệp tin thành winmain.cpp



4. Kích chuột vào nút Open để mở tệp tin đó và bắt đầu viết mã lệnh.

WinMain

Phần đầu tiên của bất kỳ một ứng dụng nào bao giờ cũng là Entry point (điểm bắt đầu, khởi tạo). Trong các ứng dụng được viết dưới dạng Consolo, hàm bắt đầu có tên là main(), trong khi đó hàm bắt đầu của một ứng dụng Window bao giờ cũng là WinMain(). Hàm WinMain này được dùng để khởi tạo ứng dụng của bạn như tạo cửa sổ giao diện chương trình, thiết lập hàm quản lý sự kiện. Tại tời điểm này bạn có thể gõ lại theo đoạn code được liệt kê dưới đây hoặc mở file winmain.cpp trong thư mục chapter2\example1.

```
// khai báo sử dụng thư viện mẫu của Windows - cần thiết cho tất cả các ứng dụng
#include <windows.h>
HINSTANCE hInst; // khai báo biến toàn cục chứa con trỏ instance (biến thể) của chương trình
HWND wndHandle; // biến toàn cục chứa con trỏ quản lý cửa sổ ứng dụng

// khai báo hàm
bool initWindow( HINSTANCE hInstance );
LRESULT CALLBACK WndProc( HWND, UINT, WPARAM, LPARAM );

// Đây là hàm winmain, hàm được gọi đầu tiền của mọi ứng dụng trong window
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow )
{
	// khởi tạo cửa sổ ứng dụng
	if ( !initWindow( hInstance ) )
	return false;
	// vòng lặp chính dùng để quản lý thông điệp:
	MSG msg;
	ZeroMemory( &msg, sizeof( msg ) );
	while( msg.message!=WM_QUIT )
	{
		// Kiểm tra các sự kiện được gửi tới trong hàng đợi của ứng dụng
		while (GetMessage(&msg, wndHandle, 0, 0) )
		{
			TranslateMessage( &msg );
			DispatchMessage( &msg );
		}
	}
	return (int) msg.wParam;
}
```

Phần quan trọng nhất của hàm trên là phần quản lý các thông điệp của chương trình. Phần này đảm nhiệm quá trình quản lý các thông điệp mà window gửi tới ứng dụng. Hàm GetMessage làm nhiệm vụ kiểm tra các thông điệp trong hàng đợi của chương trình, nhận dạng những thông điệp mà người dùng gửi tới và đang chờ chương trình xử lý. Khi có bất kỳ một thông điệp nào được gửi tới (hàm GetMessage() trả về giá trị True) thì hàm TranslateMessage and DispatchMessage sẽ được gọi để xử lý.

Sau khi bạn đã tạo xong hàm WinMain, bây giờ là lúc chúng ta xây dựng hàm tạo cửa sổ ứng dụng.

InitWindow

Trong môi trường Window, bất kỳ một ứng dụng nào muốn tạo một cửa sổ trên màn hình desktop đều cần phải đăng ký 1 lớp đối tượng thuộc lớp window. Sau khi lớp đó được đăng ký ứng dụng của bạn sẽ có thể tạo những cửa sổ cần thiết. Đoạn mã dưới đây là một ví dụ quá trình đăng kí một cửa sổ thông thường với hệ thống và sau đó lớp này sẽ được dùng để tạo ra một cửa sổ thực sự trong môi trường Windows.

```
/*******************************************************************************
* bool initWindow( HINSTANCE hInstance )
* initWindow registers the window class for the application, creates the window
*******************************************************************************/
bool initWindow( HINSTANCE hInstance )
{
	WNDCLASSEX wcex;
	// Xác lập thuộc tính đối tượng kiểu  WNDCLASSEX structure.  Các thuộc tính này sẽ tác
	// động tới cách thể hiện của cửa sổ chương trình
	wcex.cbSize = sizeof(WNDCLASSEX); // hàm sizeof() trả về kích thước của một đối
					//tượng kiểu dữ liệu đầu vào – đơn vị tính là byte
	wcex.style = CS_HREDRAW | CS_VREDRAW; // xác lập kiểu lớp
	wcex.lpfnWndProc = (WNDPROC)WndProc; // xác lập tên hàm gọi lại callback procedure
	wcex.cbClsExtra = 0; // xác lập số byte cấp phát thêm cho Class
	wcex.cbWndExtra = 0; // xác lập số byte cấp phát thêm cho mỗi instance của Class
	wcex.hInstance = hInstance; // con trỏ (handle) trỏ tới instance của ứng dụng
	wcex.hIcon = 0; //loại biểu tượng chương trình
	wcex.hCursor = LoadCursor(NULL, IDC_ARROW);// xác lập kiểu con trỏ chuột mặc định
	wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1); // màu nền của cửa sổ
	wcex.lpszMenuName . = NULL; // con trỏ trỏ tới object dữ liệu thực đơn ứng dụng
	wcex.lpszClassName = “DirectXExample”; // tên lớp đăng ký với hệ thống
	wcex.hIconSm = 0; // con trỏ tới dữ liệu biểu tượng cửa sổ ứng dụng
	RegisterClassEx(&wcex);		//gọi hàm đăng ký lớp với hệ thống
	// Tạo cửa sổ mới
	wndHandle = CreateWindow(
	“DirectXExample”, //tên lớp sử dụng đã khai báo và đăng ký ở trên
	“DirectXExample”, //tiêu đề của cửa sổ chương trình
	WS_OVERLAPPEDWINDOW, //loại của sổ chương trình
	CW_USEDEFAULT, // toạ độ X của của sổ khi xuất hiện
	CW_USEDEFAULT, // toạ độ Y của của sổ khi xuất hiện
	640, // kích thước bề ngang của cửa sổ - đơn vị là pixel
	480, // kích thước chiều cao của cửa sổ
	NULL,   // con trỏ trỏ tới đối tượng cha ;
		//NULL = đối tượng quản lý là desktop của Windows
	NULL, // con trỏ đối tượng menu của chương trình; NULL = không sử dụng
	hInstance, // con trỏ instance của ứng dụng
	NULL); // không có giá trị gì được truyền cho cửa sổ
	// Kiểm tra lại xem quá trinh khởi tạo cửa sổ có thành công hay không
	if (!wndHandle)
	return false;
	// Thể hiện cửa sổ lên màn hình Window
	ShowWindow(wndHandle, SW_SHOW);
	UpdateWindow(wndHandle);
	return true;
}
```

Tất cả thông tin, ý nghĩa của các hàm sử dụng ở trên bạn có thể tham khảo thêm trong bất kỳ tài liệu hướng dẫn lập trình windows nào. Chính vì thế chúng ta sẽ đi nhanh qua các phần này, các mã lệnh sẽ được được viết một số chú thích để bạn dễ hình dung hơn tiến trình thực hiện của chương trình.

Tất cả các ứng dụng muốn thể hiện một cửa sổ trên màn hình đều phải thực hiện quá trình đăng ký lớp cửa sổ đó với hệ thống. Lớp cửa sổ này chứa những đặc tính của cửa sổ như màu màn hình nền, loại biểu tượng chuột sử dụng, và biểu tượng thể hiện của chương trình. Lớp cửa sổ này đã được cung cấp mâu khai báo sẵn theo cấu chúc đối tượng WNDCLASSEX. Sau khi đối tượng có kiểu WNDCLASSEX đã được xác lập đầy đủ các

thông tin, nó sẽ được truyền làm tham số cho lời gọi hàm đăng ký với hệ thống RegisterClassEx.

Hàm RegisterClassEx sẽ lấy thông tin được cung cấp trong đối tượng WNDCLASSEX truyền cho hàm và thực hiện quá trình đăng ký lớp cửa sổ ứng dụng với hệ thống. Sau khi bạn đã có một lớp cửa sổ được đăng ký, bạn sẽ có thể tạo ra các cửa sổ được sử dụng trong ứng dụng.

Bước tiếp theo đó là tạo ra cửa sổ, nó được thực hiện thông qua hàm CreateWindow.

Hàm CreateWindow này yêu cầu 11 đối số, mỗi một thông số được cung cấp sẽ được dùng để xác lập kiểu hiển thị của cửa sổ ứng dụng trên màn hình. Các thông số này đã được chú giải trong phần mã nguồn ở trên.

WndProc

Hàm *WndProc* (Window Procedure) là một hàm nữa bạn cần phải khai báo cho một cửa sổ ứng dụng. Mã nguồn của hàm này được minh hoạ ở dưới đây, hàm này sẽ làm nhiệm vụ đón nhận những sự kiện mà hệ thống gửi đến cho ứng dụng. Ví dụ như, khi sự kiện chuột được kích trên cửa sổ ứng dụng, hệ thống sẽ gửi sự kiện kích chuột này tới cho ứng dụng thông qua hàm *WndProc* này. Trong hàm này, bạn sẽ tiến hành khai báo sự những sự kiện nào sẽ được xử lý hay bỏ qua chúng. Đoạn mã ví dụ dưới đây sẽ minh hoạ những sự kiện đơn giản nhất mà một ứng dụng cần quản lý.

```
/*******************************************************************************
* LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
* LPARAM lParam)
* The window procedure
*******************************************************************************/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
        // Kiểm tra xem có thông điệp nào được gửi tới hàng đợi cửa ứng dụng không
        switch (message)                //lọc các thông điệp
        {
                case WM_DESTROY:   //bắt thông điệp yêu cầu kết thúc ứng dụng
                PostQuitMessage(0);     //gọi hàm xử lý
                break;
        }
        // Chú ý, hàm này yêu cầu bạn phải cung cấp thông điệp trả về cho quá trình
        // xử lý tiếp theo
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Tại thời điểm này, bạn đã có thể biên dịch ứng dụng và chạy thử, sau khi chạy, ứng dụng sẽ có dạng một cửa sổ có nền trắng tương tự hình minh hoạ dưới đây. Bạn cũng có thể tìm thấy chương trình này trong thư viện Chapter2\example1

Sử dụng DirectX

Với phiên bản DirectX 8 trước đây, phần vẽ các đối tượng được chia ra làm hai hệ thống giao diện: DirectDraw và Direct3D. DirectDraw được sử dụng để thể hiện các đối tượng 2D, nó không còn được tiếp tục phát triển thêm nữa. Chính vì thế với phiên bản D9, nó đã được gộp lại toàn bộ trong một giao diện thống nhất thông qua các hàm API của Direct3D.

Trong phần này chúng ta sẽ dần dần tiếp cận với cách để thực hiện tất cả quá trình vẽ thông qua Direct3D. Trước tiên là các bước sác lập hệ thống để có thể chạy được hệ thống giao diện lập trình Direct3D và tiếp đến là thực thi nó:

- Tạo một đối tượng Direct3D.
- Tạo một môi trường Direct3D (thiết bị - device) để thể hiện đối tượng.
- Vẽ các đối tượng lên môi trường đó.

Đối tượng Direct3D

Đối tượng Direct3D là một đối tượng cung cấp một giao diện được sử dụng bởi các hàm để tập hợp và nhận biết khả năng tương thích của môi trường Direct3D. Có thể ví dụ, một đối tượng Direct3D đem đến cho bạn khả năng tìm và xác nhận xem hệ thống có bao nhiêu thiết bị hiển thị đồ hoạ đã được cài đặt trên hệ thống cũng như kiểm tra khả năng tương thích (hỗ trợ) của chúng.

Một đối tượng Direct3D được tạo thông qua cách sau:

```
IDirect3D9 *Direct3DCreate9( D3D_SDK_VERSION );
```

Chú ý:

D3D_SDK_VERSION chỉ là một tham số mặc định có thể gửi cho hàm Direct3DCreate9

Hàm này sẽ trả về một con trỏ trỏ tới giao diện IDirect3D9. Giá trị trả về của hàm sẽ là NULL nếu quá trình tạo mới một đối tượng thất bại.

Bạn có nhớ là tôi đã đề cập tới khả năng truy suất số lượng thiết bị hiển thị video hay adapters trên máy? Đây là một chức năng đã được cung cấp trong Direct3D, để làm được điều này bạn chỉ cần làm những việc sau.

```
UNIT IDirect3D9::GetAdapterCount(VOID);
```

Hàm GetAdapterCound có khả năng cho phép bạn biết số lượng thiết bị hiển thị của hệ thống. Hàm này không yêu cầu bất kỳ một tham số nào phải truyền cho nó và thông thường nó sẽ có giá trị trả về là 1.

Chú ý:

Nếu hệ thống chỉ có một thiết bị hiển thị được cài đặt (card đồ hoạ), thì thiết bị đó sẽ được coi là thiết bị chính để hiển thị. Trong trường hợp có nhiều hơn 1 thì thiết bị đầu tiên sẽ là thiết bị chính.

Tạo một thiết bị kết xuất (Rendering)

Một *Direct3D device*, thông qua giao diện IDirect3DDevice9 sẽ cung cấp các phương thức cho các ứng dụng có thể sử dụng để kết xuất hình ảnh ra màn hình. Thông qua giao diện này tất cả các thao tác vẽ trong game của bạn sẽ được thực hiện.

Một thiết bị Direct3D này có thể được tạo bằng cách gọi tới hàm CreateDevice.

```
HRESULT CreateDevice(
        UINT Adapter,
        D3DDEVTYPE DeviceType,
        HWND hFocusWindow,
        DWORD BehaviorFlags,
        D3DPRESENT_PARAMETERS *pPresentationParameters,
        IDirect3DDevice9** ppReturnedDeviceInterface
);
```

Đối tượng thiết bị kết quả trả về này sẽ được sử dụng trong toàn bộ game của bạn để truy cập và kết xuất kết quả vẽ tới các thiết bị hiển thị phần cứng. Hàm CreateDevice này yêu cầu 6 tham số đầu vao và trả về một kết quả kiểu HRESULT. Nếu như quá trình thực hiện hàm này thanh công, kết quả trả về sẽ là D3D_OK; trong các trường hợp khác nó có thể là một trong các giá trị dưới đây:

- **D3DERR_INVALIDCALL**. Một trong những tham số cung cấp cho hàm không hợp lệ.
- **D3DERR_NOTAVAILABLE**. Thiết bị không hỗ trợ bạn gọi hàm này.
- **D3DERR_OUTOFVIDEOMEMORY**. Card đồ hoạ không đủ bộ nhớ để thực hiện hàm lời gọi hàm này.

Chú ý:

Một thói quen tốt mà bạn nên tạo lập đó là luôn luôn kiểm tra kết quả trả về của hàm để đảm bảo chắc chắn quá trình tạo đối tượng đã thực hiện chính xác mặc dù hầu hết các trường hợp hàm này sẽ trả về kết quả D3D_OK.

Các tham số cần thiết của hàm CreateDevice:

- **Adapter**. Có kiểu UINT. Có giá trị là số hiệu của card đồ hoạ mà thiết bị dùng để hiển thị. Hầu hết các ứng dụng game sẽ gửi giá trị D3APAPTER_DEFAULT cho hàm để mặc định sử dụng thiết bị đồ hoạ chính của hệ thống.

- **DeviceType**. Có kiểu D3DDEVTYPE. Có 3 kiểu thiết bị mà bạn có thể lựa chọn để khởi tạo:
  - D3DDEVTYPE_HAL. Thiết bị sử dụng chức năng hỗ trợ từ phần cứng card đồ hoạ tốc độ cao.
  - D3DDEVTYPE_REF. Các tính năng của Direct3D sẽ được cài đặt trong phần mềm, tuy nhiên sẽ sử dụng các chức năng đặc biệt mà CPU hỗ trợ nếu có thể.
  - D3DDEVTYPE_SW. Thiết bị cắm-chạy có chức mô phỏng bằng phần mềm sẽ được sử dụng.
- **hFocusWindow**. Có kiểu HWND. Đây là con trỏ của cửa sổ sử dụng thiết bị đó.
- **BehaviorFlags.** Có kiểu DWORD. Tham số này cho phép xác lập nhiều giá trị cờ trạng thái khác nhau cho quá trình khởi tạo thiết bị. Trong ví dụ minh hoạ ở đây chúng ta chỉ sử dụng giá trị cờ trạng thái xác lập tiến trình sử lý vertex sẽ được thực hiện bằng phần mềm: D3DCREATE_SOFTWARE_VERTEXPROCESSING.
- **PresentationParamters.** Có kiểu D3DPRESENT_PARMETERS. Đây là tham số điều khiển quá trình hiển thị của thiết bị như xác lập kiểu hiển thị cửa sổ ứng dụng/fullcreen hay có sử dụng hay không bộ nhớ đệm backbuffer. Cấu trúc của tham số này có dạng sau:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
        UINT BackBufferWidth, BackBufferHeight;
        D3DFORMAT BackBufferFormat;
        UINT BackBufferCount;
        D3DMULTISAMPLE_TYPE MultiSampleType;
        DWORD MultiSampleQuality;
        D3DSWAPEFFECT SwapEffect;
        HWND hDeviceWindow;
        BOOL Windowed;
        BOOL EnableAutoDepthStencil;
        D3DFORMAT AutoDepthStencilFormat;
        DWORD Flags;
        UINT FullScreen_RefreshRateInHz;
        UINT PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

  Bảng 2.1 sẽ mô tả chi tiết hơn các tham số trên.
- **ppReturnedDeviceInterface.** Có kiểu IDirect3Ddevice9**. Đây là biến chứa con trỏ thiết bị được tạo hàm tạo ra nếu quá trình khởi tạo thành công.

Sau khi một thiết bị hiển thị đã được khởi tạo bạn sẽ có thể gọi tiếp các phương thức khác của Direct3D để lấy hoặc vẽ bất kỳ một đối tượng nào lên màn hình.

Bảng 2.1 D3DPRESENT_PARAMETERS

Thuộc tính	Mô tả chi tiết
BackBufferWidth	Bề rộng của vùng đệm BackBuffer
BackBufferHeight	Chiều cao của vùng đệm BackBuffer
BackBufferFormat	Kiểu dữ liệu vùng đệm – nó có kiểu D3DFORMAT. Chú ý trong chế độ cửa sổ ứng dụng bạn phải truyền cho hàm này giá trị D3FMT_UNKNOW.
BackBufferCount	Số lượng vùng đệm BackBuffer được tạo
MultiSampleType	The levels of full-scene multisampling. Unless multisampling is being supported specifically, pass D3DMULTISAMPLE_NONE.
MultiSampleQuality	The quality level. Pass 0 to this parameter unless multisampling is enabled.
SwapEffect	Kiểu cháo đổi dữ liệu khi chuyển đổi bộ đệm backbuffer. Trong ví dụ này ta sử dụng D3DSWAPEFFECT_DISCARD
hDeviceWindow	Cửa sổ ứng dụng chứa thiết bị được tạo.
Windowed	Có giá trị TRUE nếu ứng dụng kiểu cửa sổ và FALSE cho ứng dụng chiếm toàn bộ màn hình
EnableAutoDepthStencil	Giá trị điều khiển độ sâu bộ đệm (depth buffer) cho ứng dụng. Xác lập giá trị TRUE nếu muốn Direct3D quản lý bộ đệm này cho bạn.
AutoDepthStencilFormat	Định dạng cho bộ đệm – có kiểu D3DFORMAT
Flags	Trừ trường hợp bạn muốn tự mình xác lập giá trị cho nó, không thì bạn có thể truyền giá trị 0 theo mặc định.
FullScreen_RefreshRateInHz	Tốc độ làm tươi màn hình. Trong chế độ cửa sổ ứng dụng, tham số này phải được xác lập là 0.
PresentationInterval	Điều khiển này xác lập tốc độ tráo đổi của bộ nhớ đệm

Xoá màn hình

Sau khi một thiết bị Direct3D đã được tạo ra, bây giờ bạn có thể bắt đầu tô điểm lên màn hình, bạn có thể làm điều này với các bức ảnh hay thông qua một số đa giác. Tuy nhiên công việc đầu tiên bạn phải làm trong vòng lặp chính của ứng dụng game là xoá sạch màn hình. Việc xoá màn hình này sẽ cho bạn một khung hình hoàn toàn mới để thao tác và vẽ các đối tượng lên đó. Tệp tin winmain.cpp đã được cập nhật thêm hàm này bạn có thể tìm thấy trong thư mục chapter2\example2 trên CD-ROM.

Bạn có thể xoá toàn bộ màn hình bằng cách gọi tới hàm Clear.

```
HRESULT Clear(
        DWORD Count,
        const D3DRECT *pRects,
        DWORD Flags,
        D3DCOLOR Color,
        float Z,
        DWORD Stencil
);
```

Bạn phải truyền đầy đủ 6 tham số cần thiết cho hàm.

Tham số thứ nhất, Count – số lượng vùng hình chữ nhật được xoá. Nếu tham số này là 0 thì tham số thứ 2 pRects phải là giá trị NULL. Trong trường hợp đó, toàn bộ vùng nhien của màn hình sẽ được xoá, đây là các tham số hay được sử dụng nhất. Nếu giá trị Count lớn hơn 0, pRects phải trỏ tới mảng các đối tượng kiểu D3DRECT lưu giữ toạ độ các vùng chữ nhật được xoá.

Tham số cờ trạng thái Flags xác lập bộ đệm sẽ được xoá. Có 3 trường hợp:

- D3DCLEAR_STENCIL
- D3DCLEAR_TARGET
- D3DCLEAR_ZBUFFER

Giá trị mà bạn sẽ sử dụng tại thời điểm này là D3DCLEAR_TARGET, nó dùng đề xác lập đối tượng được xoá là vùng dữ liệu hiển thị hiện thời.

Color – giá trị màu (có kiểu D3DCOLOR) sẽ được xác lập sau khi xoá. Có rất nhiều marcos sẽ trợ giúp bạn xác lập giá trị màu theo ý muốn tuỳ theo các chuẩn màu sắc khác nhau, ví dụ như D3DCOLOR_XRGB.

Tham số Z xác lập giá trị chiều sâu bộ đệm. Giá trị này nằm trong khoảng từ 0.0f tới 1.0f. Chúng ta sẽ nói sâu hơn về nó trong các chương sau.

Tham số Stencil lưu giá trị số hiệu bộ đệm mẫu tô. Nếu bạn không sử dụng bộ đệm này thì bạn phải truyền giá trị này cho hàm là 0.

Thể hiện một phông cảnh

Sau khi bạn đã xoá sạch một khung hình, bây giờ là lúc chúng ta thể hiện nó lên màn hình. Direct3D sử dụng hàm Present để làm điều này. Hàm Present này sẽ thực hiện việc chuyển đổi trang của bộ đệm nền. (có thể ví quá trình hiển thị hình ảnh như việc lật những bức vẽ (từ bộ đệm) lên màn hình).

Thực chất thì tất cả quá trình bạn thao tác vẽ cho tới thời điểm này đều được thực hiện trên bộ nhớ đệm. Bộ nhớ đệm *back buffer* này là một vùng nhớ để lưu giữ toàn bộ kết quả vẽ trước khi nó được kết xuất lên màn hình. *Page flipping* (lật trang) là một tiến trình lấy các thông tin từ bộ nhớ đệm và hiển thị nó lên màn hình. Việc làm này nhằm giảm hiện tượng giật hình do tốc độ hiển thị của màn hình thường chậm hơn nhiều so với tốc độ sử lý của bộ nhớ trên card màn hình. Hàm này sẽ đảm bảo quá trình hiển thị sẽ mượt mà hơn và tất cả những gì bạn thực hiện trên bộ nhớ đệm sẽ được hiển thị lên màn hình. (!)

Chú ý:

> Lật trang là quá trình tráo đổi dữ liệu từ bộ nhớ đệm lên bộ nhớ thực của màn hình, để thực hiện quá trình hiển thị thì bạn phải gọi tới hàm yêu cầu lật trang này mỗi khi thực hiện xong các thao tác vẽ trên bộ nhớ đệm.

Cấu trúc của hàm Present:

```
HRESULT Present(
    CONST RECT *pSourceRect,
    CONST RECT *pDestRect,
    HWND hDestWindowOverride,
    CONST RGNDATA *pDirtyRegion
);
```

Hàm Present này yêu cầu 4 tham số đầu vào:

- **pSourceRect** là một con trỏ có kiểu RECT trỏ tới dữ liệu vùng chữ nhật thể hiện từ bộ nhớ đệm backbuffer. Giá trị này phải được gán là NULL nếu bạn muốn sử dụng toàn bộ dữ liệu vùng nhớ đệm backbuffer – đây cũng là trường hợp thường dùng.
- **pDesRect** là một con trỏ kiểu RECT khác chứa địa chỉ vùng chữ nhật đích.
- **hDestWindowOverride** là con trỏ trỏ tới của sổ đích được dùng để hiển thị. Giá trị này phải được xác lập là NULL nếu bạn sử dụng những xác lập cửa sổ trước đó trong phần khai báo tham số hiển thị ở trên.
- **pDirtyRegion** là con trỏ trỏ tới những vùng mà bộ đệm cần được cập nhật dữ liệu. Một lần nữa giá trị này là NULL nếu bạn muốn cập nhật toàn bộ dữ liệu bộ đệm.

Dọn dẹp bộ nhớ (Cleaning Up)

Công việc cuối bạn phải làm trong bất kỳ một ứng dụng DirectX nào là dọn dẹp vào giải phóng bộ nhớ cho các đối tượng bạn đã sử dụng. Ví dụ như, tại thời điểm bắt đầu của ứng dụng, bạo đã tạo ra cả một đối tượng Direct3D cũng như một thiết bị Direct3D. Khi ứng dụng được đóng lại, bạn cần thiết phải giải phóng những đối tượng đó để trả lại tài nguyên các đối tượng đó đã chiếm chỗ cho hệ thống thực hiện các công việc khác.

Các đối tượng COM, nền tảng của DirectX, luôn được quản lý bởi hệ thống thông qua một biến đếm lưu giữ số lượng đối tượng mà ứng dụng đã tạo ra cũng như còn lại sau khi vừa được loại bỏ. Bằng việc sử dụng hàm Release, bạn vừa thực hiện việc giải phóng đối tượng và vừa tác động tới biến đếm (count) đó. Khi mà biến đếm đạt giá trị 0, hệ thống sẽ giải phóng toàn bộ bộ nhớ mà các đối tượng đã sử dụng.

Lấy vị dụ, khi giải phóng một thiết bị hiển thị Direct3D, bạn sử dụng lệnh sau:

```
If ( pd3dDevice != NULL )
        pd3dDevice->Release( );
```

Nếu điều kiện trong câu lệnh If được thoả mãn (tức là biến pd3dDevice có chứa dữ liệu – nó được gán cho địa chỉ của một thiết bị được khởi tạo trước đó), thì thực hiện quá trình giải phóng biến đó.

Chú ý:

Bạn nên kiểm tra xem chắc chắn một đối tượng DirectX có khác rỗng (NULL) không trước khi gọi hàm Release đối với nó. Nếu bạn gọi hàm này trên một đối tượng rỗng thì chắc chắn sẽ xảy ra lỗi cho ứng dụng.

Cập nhật mã nguồn chương trình.

Sau khi đã tìm hiểu qua một vài lệnh và thông tin cơ bản về DirectX, bây giờ là lúc bạn sẽ xem xét làm thế nào để xác lập và chạy một ứng dụng DirectX, là lúc để chúng ta thêm các mã lệnh mới vào ứng dụng. Những mã lệnh này sẽ được chèn thêm vào trong tệp tin winmain.cpp mà chúng ta đã tạo trong phần trước.

Bước đầu tiên trước khi viết bất kỳ một ứng dụng DirectX nào đó là chèn tệp tin header của Direct3D. Bạn thực hiện khai báo như sau trong ứng dụng:

```
#include <d3d9.h>
```

Tiếp theo bạn cần khai báo 2 biến toàn cục sẽ cần thiết trong quá trình lưu giữ con trò đối tượng để toàn bộ ứng dụng sử dụng.

```
LPDIRECT3D9         pD3D;       //Đối tượng Direct3D
LPDIRECT3DDEVICE9 pd3dDevice;   //thiết bị hiển thị Direct3D
```

Biến được khai báo kiểu LPDIRECT3D9 là một con trỏ sử dụng giao diện IDirect3D9, tương tự biến LPDIRECT3DDEVICE9 là con trỏ sử dụng giao diện IDirect3DDevice9.

Tiếp theo, bạn sẽ cần phải gọi tới hàm initDirrect3D, bạn có thể thực hiện theo cách tôi sử dụng dưới đây. Chú ý là bạn phải đặt đoạn code gọi tới hàm này sau khi đã gọi hàm initWindow ở trong hàm WinMain.

```
// hàm này được gọi sau khi quá trình khởi tạo cửa sổ ứng dụng kết thúc
If ( !initDirect3D( ) )
        return false;
```

Thay đổi mã lệnh trong vòng lặp quản lý sự kiện

Sau khi đã xác lập các thông số khởi tạo, bây giờ là lúc chúng ta thay thế vòng lặp quản lý sự kiện Windows bằng một mã lệnh khác hữu dụng hơn cho các ứng dụng game. Trong vòng lặp quản lý sự kiện ban đầu chúng ta sử dụng thì vòng lặp này sẽ liên tục gọi tới hàm GetMessage để kiểm tra sự kiện mỗi khi nó được gửi tới và chờ ứng dụng sử lý; Nếu thông điệp được gửi tới, hàm GetMassage sẽ phải đợi tới khi thông điệp sử lý xong mới tiếp tục. Chính vì thế trong phần này thay vì sử dụng hàm GetMassage thì chúng ta sẽ sử dụng hàm PeekMessage, hàm này cũng có chức năng kiểm tra thông điệp tuy nhiên nó trả điều khiển về ngay lập tức, cho phép ứng dụng game của bạn có thể gọi lại chính nó trong vòng lặp.

Trong ví dụ này, chúng ta sẽ chèn thêm mệnh đề else sau lời gọi PeekMessage để gọi tới hàm render của game. Hàm render này sẽ thực hiện quá trình vẽ các đối tượng lên màn hình, hàm này sẽ được định nghĩa ngay ở phần sau.

```
if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
{
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
}
else
{
        render( );
}
```

Hàm khởi tạo Direct3D

Trong hàm initDirect3D này chúng ta sẽ thực hiện việc tạo một đối tượng Direct3D và thiết bị để sử dụng chúng.

```
/**************************************************************************
* initDirect3D
**************************************************************************/
bool initDirect3D(void)
{
        pD3D = NULL;
        pd3dDevice = NULL;
        // Create the DirectX object
        if( NULL == ( pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
        {
                return false;
        }
        // Fill the presentation parameters structure
```

```
        D3DPRESENT_PARAMETERS d3dpp;
        ZeroMemory( &d3dpp, sizeof( d3dpp ) );
        d3dpp.Windowed = TRUE;
        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
        d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
        d3dpp.BackBufferCount = 1;
        d3dpp.BackBufferHeight = 480;
        d3dpp.BackBufferWidth = 640;
        d3dpp.hDeviceWindow = wndHandle;
        // Create a default DirectX device
        if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                                        D3DDEVTYPE_REF,
                                        wndHandle,
                                        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                        &d3dpp,
                                        &pd3dDevice ) ) )
        {
                return false;
        }
        return true;
}
```

Tại đoạn đầu của hàm trên, chúng ta đã tạo một lời gọi tới hàm Direct3DCreate9. Nó sẽ thực hiện lệnh tạo một đối tượng Direct3D, từ đó bạn có thể tạo một thiết bị hiển thị Direct3D. Tiếp đến là quá trinh xác lập các thông số hiển thị của ứng dụng. Chúng ta đã xác lập một cửa số có kích thước 640 x 480.

Tiếp theo là hàm CreateDevice sẽ được gọi với tham số thứ 2 ở cuối cùng là biến cấu trúc với dữ liệu đã vừa được định nghĩa. Cũng tại đây chúng ta muốn hàm CreateDevice sẽ sử dụng thiết bị hiển thị đồ hoạ chính của hệ thống nên giá trị truyền vào là D3DAPAPTER_DEFAULT. Ngoài ra giá trị D3DDEVTYPE_REF cũng nhằm để xác lập rằng thiết bị sẽ sử dụng là thiết bị chuẩn Direct3D. Chúng ta cũng xác lập một tham số khác với giá trị D3DCREATE_SOFTWARE_VERTEXPROCESSING nhằm đảm bảo rằng ví dụ của chúng ta sẽ chạy được trên hầu hết tất cả các phần cứng. Khả năng truy cập chức năng Vertex trên phần cứng chỉ có được trên một vài card đồ hoạ mới. Và tham số cuối cùng &pd3dDevice là biến lưu giữ kết quả tạo đối tượng thiết bị Direct3D mà hàm trả về.

Hàm kết xuất hình ảnh (Render)

Hàm render là nơi bạn thực sự thao tác vẽ lên thiết bị. Như đã giới thiệu ở trên, thì hàm này sẽ được gọi bên trong vòng lặp chính và được gọi sau mỗi khung hình hiển thị.

```
/**************************************************************************
* render
**************************************************************************/
void render(void)
{
        // Kiểm tra xem chắc chắn thiết bị Direct3D đã được tạo hay chưa
        if( NULL == pd3dDevice )
                return;// Xoá toàn bộ bộ đệm màn hình về trạng thái mầu xanh da trời
        pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
                                D3DCOLOR_XRGB( 0,0,255 ), 1.0f, 0 );
        // Thể hiện dữ liệu bộ đệm lên màn hình
        pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Trên đây chỉ là một ví dụ đơn giản của hàm render. Trước hết là chúng ta phải kiểm tra chắc chắn kết quả trả về sau khi gọi hàm CreateDevice, nếu nó có giá trị NULL tức là có lỗi khởi tạo thiết bị Direct3D. Tất nhiên đối với trường hợp này thì chúng ta sẽ chẳng thể thao tác gì trên đối tượng thiết bị này nữa ngoài việc thoát khỏi hàm.

Tiếp đó, chúng ta sử dụng hàm Clear đã được giới thiệu ở trên. Bởi vì chúng ta muốn xoá toàn bộ bộ đệm, nên bạn cần truyền tham số 0 và NULL cho 2 tham số đầu của hàm. Xác lập D3DCLEAR_TARGET sẽ yêu cầu DirectX xoá toàn bộ bộ nhớ đệm. Thông số tiếp theo là một biến có kiểu D3DCOLOR. Trong ví dụ này chúng ta sử dụng macro D3DCOLOR_XRGB để lựa chọn màu cần xoá là màu xanh nước biển với các giá trị màu tương ứng trong hệ RGB là: R=0, G=0, B=255.

Ngoài ra bạn cũng cần phải truyền một giá trị thực 1.0 để xác lập độ sâu của vùng đệm. Độ sâu của vùng đệm sẽ giúp Direct3D nhận biết khoảng nhìn giữa người chơi và cảnh vật xung quanh. Bạn có thể xác lập giá trị này trong khoảng 0.0 tới 1.0. Giá trị cao đồng nghĩa với phạm vi nhìn của người chơi sẽ xa hơn.

Bộ đệm stencil (!) cho phép bạn đánh dấu vùng chính diện của bức ảnh mà chúng không được hiển thị. Bởi vì chúng ta không sử dụng nó nên tham số này sẽ được xác lập là 0.

Công việc cuối cùng cần phải thực hiện trong hàm render là thể hiện đối tượng từ bộ đệm lên màn hình. Nó được thực hiện thông qua lời gọi tới hàm Present. Bởi vì bạn muốn thể hiện toàn bộ vùng đệm lên màn hình, giá trị NULL sẽ được truyền cho tất cả các tham số yêu cầu của hàm. Đây cũng là xác lập bạn thường dùng nhất.

Khai báo hàm cleanUp

Dĩ nhiên là sau khi ứng dụng kết thúc, bạn sẽ muốn giải phóng hết tất cả các đối tượng đã tạo ra. Nó được thực hiện trong đoạn code mô tả dưới đây.

```
void cleanUp (void)
{
        // Giải phóng đối tượng và thiết bị hiển thị DIrect3D
        if( pd3dDevice != NULL )
                pd3dDevice->Release( );
        if( pD3D != NULL )
                pD3D->Release( );
}
```

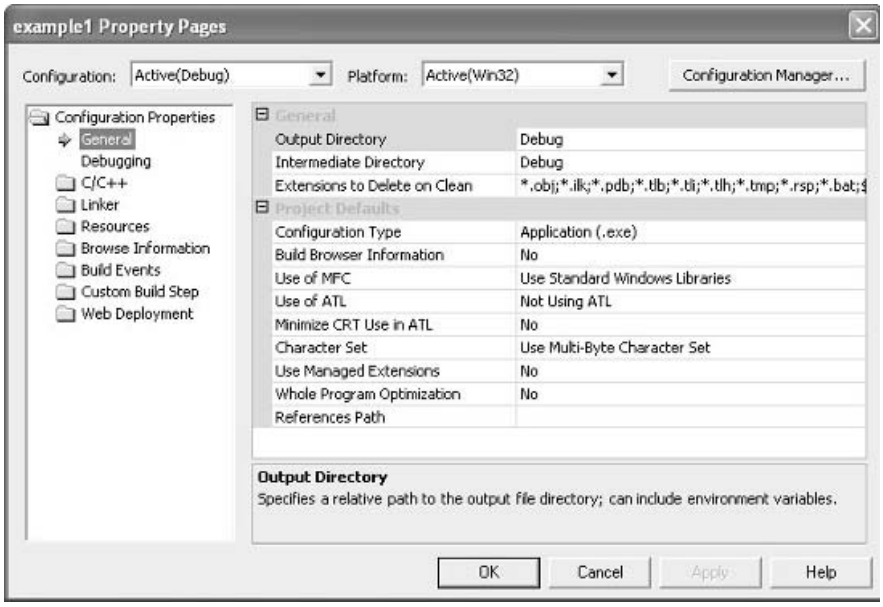
Trước khi giải phóng một đối tượng nào, bạn cần phải kiểm tra xem đối tượng đó có khác NULL không. Nếu đối tượng đó thực ra đã được tạo ra, phương thức Release sẽ được gọi để tiến hành giải phóng đối tượng đó khỏi bộ nhớ. Hàm này sẽ được đặt trước lời gọi xác lập giá trị trả về của hàm WinMain.

Chèn thư viện DirectX vào chương trình

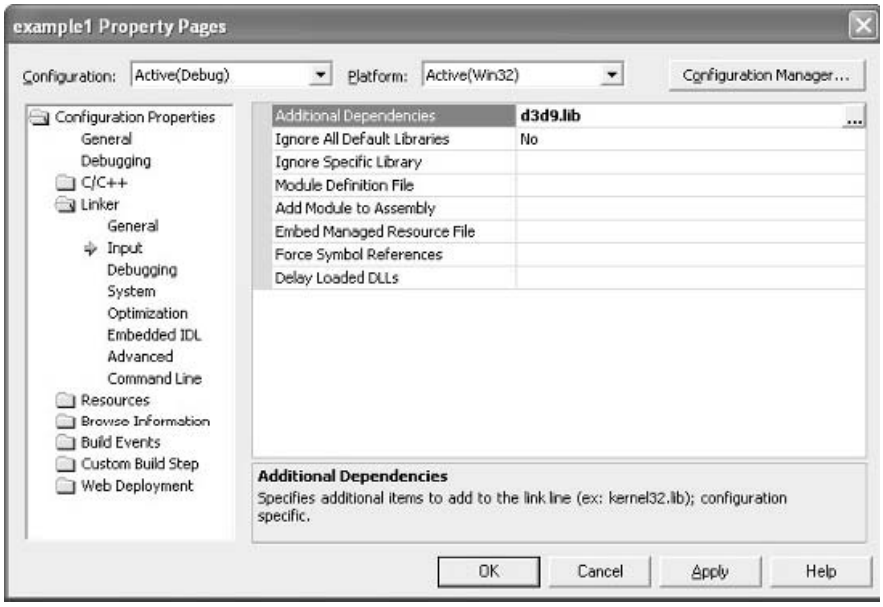
Cuối cùng thì bạn cũng đã có tất cả những đoạn code cần thiết đều tạo một ứng dụng DirectX đâu tiên. Tuy nhiên trước khi bạn có thể biên dịch ứng dụng và chạy nó, bạn sẽ cần phải thực hiện một vài xác lập: liên kết thư viện DirectX. Trong ví dụ đơn giản này, bạn sẽ chỉ cần tạo liên kết tới tệp thư viện d3d9.lib.

1. Lựa chọn Properties option trong thực đơn Project. Cửa sổ hội thoại Property Pages sẽ xuất hiện. Hộp thoại này có hình dạng tương tự hình minh hoạ 2.6

2. Lựa chọn Linker trong hộp panel bên trái. Đối tượng này sẽ được mở rộng và cho phép bạn xác lập các tuỳ chọn trong đó.
3. Tiếp đến, bạn lựa chọn thông số đầu vào Input. Hộp thoại sẽ thay đổi nội dung và sẽ có dạng tương tự hình minh hoạ 2.7 dưới đây.



Hình 2.6 Cửa sổ hộp thoại Property Pages.



Hình 2.7 Xác lập các thông số trong tuỳ chọn Linker

4. Gõ vào d3d9.lib trong mục Additional Dependencies và kích chọn OK để kết thúc.

Biên dịch và chạy ứng dụng. Không giống như ví dụ ban đầu, cửa sổ ứng dụng này sẽ có một màu nền xanh nước biển. Mặc dù ứng dụng này không đi sâu vào những thứ mà DirectX có thể làm, nhưng nó đã cung cấp cho bạn những kiến thức nền tảng để bắt đầu với công nghệ này.

Chú ý:

Có rất nhiều tệp tin thư viện cần thiết cho những chức năng khác nhau của DirectX. Bạn chỉ cần liên kết tới những thư viện nào chứa những hàm cần thiết mà bạn muốn sử dụng.

Xác lập ứng dụng chạy ở chế độ toàn màn hình

Trong ví dụ mà chúng ta đã thực hiện ở trên, ứng dụng của chúng ta đều là dạng cửa sổ có kích thước 640x480 và được hiển thị trên màn hình desktop. Mặc dù kiểu giao diện này phù hợp cho các ứng dụng thông thường tuy nhiên đối với một ứng dụng game, bạn cần phải tạo các hiệu ứng ấn tượng người chơi bằng một không gian ảo nhưng phải thật nhất. Chính vì vậy bạn không thể không sử dụng chế độ chạy ứng dụng ở dạng toàn màn hình.

Để tạo ứng dụng dạng này bạn chỉ cần phải thay đổi một đoạn code rất nhỏ trong ứng dụng mà chúng ta vừa tạo, thay đổi này được xác lập chủ yếu trong hàm CreateWindow.

Đây là đoạn code gọi hàm CreateWindow mà chúng ta đã dùng:

```
wndHandle = CreateWindow(“DirectXExample”,
                         “DirectXExample”,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT,
                         CW_USEDEFAULT,
                         640,
                         480,
                         NULL,
                         NULL,
                         hInstance,
                         NULL);
```

Bạn có thể thấy tham số thứ ba, chúng ta đã xác lập kiểu ứng dụng là dạng cửa sổ với giá trị xác lập là WS_OVERLAPPEDWINDOW. Xác lập này bao gồm kiểu ứng dụng có thanh tiêu đề, có viền bao cửa sổ và các nút nhấn thu nhỏ (Minimize) và đóng cửa sổ. Trước khi chúng ta tiến hành xác lập cửa sổ dạng thể hiện toàn màn hình, kiểu dáng cửa sổ chúng ta cần xác lập các thông số sau:

```
WS_EX_TOPMOST | WS_POPUP | WS_VISIBLE
```

Phần xác lập đầu tiên, WS_EX_TOPMOST, có tác dụng đặt vị trí hiên thị của cửa sổ này luôn ở trạng thái bên trên tất cả các cửa sổ khác. WS_POPUP có tác dụng xác lập cửa sổ không có viền (border), không thanh tiêu đề và không có nút Minimize và Close. Phần xác lập cuối cùng, WS_VISIBLE yêu cầu với windows để ứng dụng tự cập nhật lại màn hình ứng dụng.

Lời gọi tới hàm CreateWindow lúc này có dạng sau:

```
wndHandle = CreateWindow(“DirectXExample”,
                         “DirectXExample”,
                         WS_EX_TOPMOST | WS_POPUP | WS_VISIBLE,
                         CW_USEDEFAULT,
                         CW_USEDEFAULT,
                         640,
                         480,
                         NULL,
                         NULL,
                         hInstance,
                         NULL);
```

Tiếp theo chúng ta sẽ tiến hành thay đổi một chút mã lệnh trong hàm initDirect3D. Trong biến có kiểu cấu trúc D3DPRESENT_PARAMETERS chúng ta đã truyền cho hàm CreateDevice, bạn cần thay đổi hai biến tham số: Windowed và BackBufferFormat. Hiện tại thì chúng ta đang sử dụng khai báo:

```
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
d3dpp.Windowed = TRUE;
```

Chú ý:

Chúng ta phải sử dụng tham số D3DFMT_UNKNOWN cho kiểu bộ đệm BackBufferFormat bởi vì chúng ta đang sử dụng kiểu hiển thị ứng dụng dạng cửa sổ.

Để xác lập ứng dụng dạng toàn màn hình trong DirectX, tham số Windowed và BackBufferFormat chúng ta cần phải xác lập lại như sau:

```
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
d3dpp.Windowed = FALSE;
```

Thay đổi mà bạn có thể thấy ngay được đó là thuộc tính d3dpp.Windowed đã được đảo ngược giá trị. Đơn giản bởi vì nó là FALSE bởi vì bạn muốn CreateDevice biết rằng ứng dụng của bạn là dạng chiếm toàn màn hình.

Xác lập thuộc tính khác d3dpp.BackBufferFormat có lẽ bạn cảm thấy khó nắm bắt hơn. Đối với ứng dụng kiểu cửa sổ mà chúng ta đã tạo lúc đầu, bạn không cần thiết phải quan tâm tới giá trị này bởi vì bạn sử dụng những xác lập mặc định của màn hình desktop bằng cách truyền tham số D3DFMT_UNKNOWN. Khi bạn muốn xác lập ứng dụng ở dạng chiếm toàn bộ màn hình, bạn cần thiết phải tự mình xác lập các thông số D3DFORMAT mà bạn muốn sử dụng. Tham số kiểu D3DFORMAT có giá trị thể hiện chất lượng mầu sắc hiển thị của màn hình. Ví dụ, trong trường hợp này chúng ta sẽ chọn D3DFMT_X8R8G8B8 là giá trị mặc định mà hầu hết các thiết bị card đồ hoạ đều hỗ trợ. D3DFMT_X8R8G8B8 thể hiện bạn sẽ sử dụng 32-bit để định dạng cho một điểm ảnh bao gồm 8 bit cho xác lập sắc mầu đỏ, 8 bít cho màu xanh lá cây, 8 bit cho màu xanh nước biển. Định dạng này cũng bao gồm 8 bit lưu trữ dữ liệu mở rộng.

Tất cả các xác lập lại code trên bạn có thể tìm thấy trong mã nguồn chương trình tại thư mục chapter2\example3 trên đĩa CD-ROM đi kèm sách.

Trong phần tiếp theo, chúng ta sẽ đi tìm hiểu cách để kiểm tra xem hệ thống có khả năng hỗ trợ những độ phân giải và kiểu thể hiện màu sắc ra sao.

Chế độ hiển thị màn hình và các kiểu định dạng

Nếu ứng dụng game của bạn chạy ở chế độ cửa sổ trên màn hình desktop thì việc kiểm tra xem hệ thống hỗ trợ các chế độ phân giải nào là không cần thiết. Bạn chỉ cần truyền giá trị D3DFMT_UNKNOWN cho hàm, chương trình sẽ tự động xác lập giá trị mặc định mà màn hình desktop đang lựa chọn.

Video Modes and Formats

Nếu game mà bạn viết chỉ chạy ở chế độ cửa sổ trên desktop, thì việc xác định chế độ đồ hoạ mà máy tính của bạn hỗ trợ không quan trọng lắm, tuy nhiên khi bạn muốn game của bạn chạy ở chế độ toàn bộ màn hình (fullscreen) , thì việc xác định những chế độ hiển thị được hỗ trợ là rất cần thiết. Tất cả các máy tính đều hỗ trợ chế độ phân giải màn hình

640x480, nhưng còn chế độ 800x600 hay 1024x768 thì sao? Không phải tất cả các thiết bị đồ hoạ đều hỗ trợ những chế độ phân giải đó. Và nếu có thì liệu chúng có được chất lượng màu bạn mong muốn? Đó là lý do tại sao khi bạn viết một game ở chế độ toàn bộ màn hình, thì tốt nhất là bạn nên kiểm tra phần cứng để chắc rằng nó hỗ trợ những gì mà game của bạn cần. Để thực hiện điều đó, Direct3D cung cấp cho bạn một số hàm thông qua giao diện IDirect3D9. Hàm đầu tiên mà bạn cần đã được giới thiệu ở phần trên:

```
UINT IDirect3D9::GetAdapterCount(VOID);
```

Thông thường, hàm này sẽ trả về một giá trị kiểu integer không dấu (unsigned integer) chỉ ra số lượng thiết bị đồ hoạ của hệ thống. DirectX hỗ trợ cùng lúc nhiều thiết bị đồ hoạ, cho phép game chạy trên nhiều màn hình. Để mọi việc đơn giản, hãy nghĩ rằng, chỉ có một thiết bị đồ hoạ.

Gathering Video Adapter and Driver Information

Viêc nắm được thông tin chắc chắn về thiết bị đồ hoạ trên máy của bạn là rất hữu ích. Ví dụ, bạn có thể muốn biết độ phân giải mà thiết bị đồ hoạ hỗ trợ, hoặc thông tin hãng sản suất. Sử dụng hàm GetAdapterIdentifier, bạn có thể thu thập được rất nhiều thông tin. Nó được định nghĩa như sau:

```
HRESULT GetAdapterIdentifier(
                        UINT Adapter
                        DWORD Flags,
                        D3DADAPTER_IDENTIFIER9 *pIdentifier
                        );
```

■ Tham số đầu tiên có kiểu integer không dấu để xác định thiết bị đồ hoạ nào mà bạn muốn lấy thông tin. Bởi vì, chúng ta đang chấp nhận rằng hiện tại chỉ có một thiết bị đồ hoạ, nên giá trị đưa vào là D3DADAPTER_DEFAULT, nghĩa là thiết bị đồ hoạ chính sẽ được chọn.

■ Tham số thứ hai, là một cờ (flag), xác nhận WHQLLevel của trình điều khiển.

■ Tham số thứ ba là con trỏ tới vùng nhớ của cấu trúc D3DADAPTER_IDENTIFIER9. Cấu trúc này lưu thông tin về thiết bị đồ hoạ do hàm trả về.

Cấu trúc D3DADAPTER_IDENTIFIER9 cung cấp những thông tin sau:

```
typedef struct _D3DADAPTER_IDENTIFIER9 {
                    // tên của trình điều khiển
                    char Driver[MAX_DEVICE_IDENTIFIER_STRING];
                    // mô tả về thiết bị
                    char Description[MAX_DEVICE_IDENTIFIER_STRING];
                    // version của thiết bị
                    char DeviceName[32];
                    // version của trình điều khiển hiện được cài đặt
                    LARGE_INTEGER DriverVersion;
                    // This value holds the bottom 32 bits of the driver version
                    DWORD DriverVersionLowPart;
                    // This value holds the upper 32 bits of the driver version
                    DWORD DriverVersionHighPart;
                    // số ID sản xuất
                    DWORD VendorId;
                    // the ID of the particular device
                    DWORD DeviceId;
                    // the second part of the device ID
```

```
DWORD SubSysId;
// the revision level of the device chipset
DWORD Revision;
// a unique identifier for the device
GUID DeviceIdentifier;
// the level of testing that this driver has gone through
DWORD WHQLLevel;
} D3DADAPTER_IDENTIFIER9;
```

Cấu trúc này lưu giữ toàn bộ những đặc trưng có liên quan đến thiết bị và trình điều khiển thiết bị được cài đặt. Đầy đủ hơn về cấu trúc này xem ở tài liều của DirectX.

Getting the Display Modes for an Adapter

Bước tiếp theo là lấy thông tin chi tiết về các chế độ hiển thị mà thiết bị đồ hoạ hỗ trợ. Để làm được điều đó, trước tiên bạn phải kiểm tra xem có bao nhiêu chế độ hiển thị có thể. Việc này được thực hiện qua hàm GetAdapterModeCount, định nghĩa như sau:

```
UINT GetAdapterModeCount(
        UINT Adapter,
        D3DFORMAT Format
        );
```

Tham số đầu tiên xác định thiết bị mà bạn muốn kiểm tra. Theo như đã nói ở trên, ta đưa vào giá trị D3DADAPTER_DEFAULT.
Tham số thứ hai xác định loại định dạng đồ hoạ (D3DFORMAT) mà bạn muốn kiểm tra. Như ở trên ta dùng D3DFMT_X8R8G8B8, với 8 bit cho màu đỏ, 8 bit cho màu xanh lá cây, 8 bit cho xanh nước biển. 8 bit chưa dùng đến. Bạn có thể đưa vào bất kì loại định dạng nào mà Direct3D định nghĩa sẵn, và GetAdapterModeCount sẽ trả về số chế độ hiển thị phù hợp với định dạng này. Bảng 2.2 đưa ra một vài định dạng D3DFORMATs có thể dùng trong DirectX: EnumAdapterModes

Table 2.2 D3DFORMATs

Format	Description
D3DFMT_R8G8B8	24-bit RGB pixel format with 8 bits per channel.
D3DFMT_A8R8G8B8	32-bit ARGB pixel format with alpha, using 8 bits per channel.
D3DFMT_X8R8G8B8	32-bit RGB pixel format, where 8 bits are reserved for each color.
D3DFMT_R5G6B5	16-bit RGB pixel format with 5 bits for red, 6 bits for green, and 5 bits for blue.
D3DFMT_X1R5G5B5	16-bit pixel format, where 5 bits are reserved for each color.
D3DFMT_A1R5G5B5	16-bit pixel format, where 5 bits are reserved for each color and 1 bit is reserved for alpha.
D3DFMT_A4R4G4B4	16-bit ARGB pixel format with 4 bits for each channel.
D3DFMT_R3G3B2	8-bit RGB texture format using 3 bits for red, 3 bits for green, and 2 bits for blue.

Hàm cuối cùng mà bạn cần dùng là EnumAdapterModes. Hàm này sẽ lấy thông tin vào biến có cấu trúc D3DDISPLAYMODE ứng với từng chế độ có thể. Sau đây là định nghĩa của hàm:

```
HRESULT EnumAdapterModes(
                UINT Adapter,
                D3DFORMAT Format,
                UINT Mode,
                D3DDISPLAYMODE* pMode
                );
```

Như ở trên, tham số đầu tiên ta đưa vào D3DADAPTER_DEFAULT. Tham số thứ hai là định dạng D3DFORMAT mà bạn đang kiểm tra. Tham số thứ ba, là số thứ tự của chế độ mà bạn muốn lấy thông tin. Nhớ rằng, hàm GetAdapterModeCount trả về số chế độ hiển thị mà thiết bị có. Và tham số thứ ba sẽ nằm trong khoảng từ 0 đến số này. Tham số cuối cùng là con trỏ đến biến có cấu trúc D3DDISPLAYMODE. Cấu trúc này lưu thông tin về chế độ đồ hoạ, như chiều rộng, chiều cao, độ quét và định dạng.

A Code Example for Querying the Video Adapter

Đoạn code dưới đây được lấy từ ví dụ 4, ở thư mục chapter2\example4 trên đĩa CD. Đoạn code này chỉ ra chính xác từng bước và các lệnh cần thiết để đưa ra một hộp thoại chưa danh sách những chế độ hiển thị có thể cho riêng từng định dạng D3DFORMAT. Tôi đã lấy hàm initDirect3D ở ví dụ trên và chỉnh sửa lại để nó có thể lấy ra thông tin về thiết bị đồ hoạ:

```
bool initDirect3D()
{
        pD3D = NULL;
        // Create the DirectX object
        if( NULL == ( pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
                return false;
};
```

Trước tiên bạn cần tạo ra một đối tượng Direct3D. Thông qua nó bạn sẽ sử dụng được những hàm cần thiết.

```
// Phần này dùng để lấy thông tin chi tiết về thiết bị đồ hoạ
D3DADAPTER_IDENTIFIER9 ident;
pD3D->GetAdapterIdentifier(D3DADAPTER_DEFAULT, 0, &ident);
```

Ở đây ta định nghĩa một cấu trúc IDENTIFIER9 và đưa vào hàm GetAdapterIdentifier. Dùng nó, tôi lấy được những thông tin sau:

```
addItemToList(“Adapter Details”);
addItemToList(ident.Description);
addItemToList(ident.DeviceName);
addItemToList(ident.Driver);
```

Ở đây hàm addItemToList dùng để bổ sung thông tin vào hộp thoại sẽ sau này.

```
// lấy ra các chế độ hiển thị mà thiết bị đồ hoạ có
UINT numModes = pD3D->GetAdapterModeCount(
D3DADAPTER_DEFAULT,
D3DFMT_X8R8G8B8);
```

Tiếp theo, ta sử dụng GetAdapterModeCount để lấy ra số chế độ tương ứng với định dạng trên (D3DFMT_X8R8G8B8). Sau đó, ta dùng con số này để thực hiện vòng lặp duyệt qua tất cả các chế độ và lấy thông tin cho từng chế độ đó.

```
for (UINT i=0; I < numModes; i++)
{
        D3DDISPLAYMODE mode; // khai báo biến kiểu D3DDISPLAYMODE
        char modeString[255]; // mảng char
        // lấy thông số cho cấu trúc D3DDISPLAYMODE với định dạng D3DFMT_X8R8G8B8
        pD3D->EnumAdapterModes(D3DADAPTER_DEFAULT,
        D3DFMT_X8R8G8B8,
        i,
        &mode);
        // In ra một dòng trống
        addItemToList(“”);
        // đưa ra chiều rộng màn hình
        sprintf(modeString, “Width=%d”,mode.Width);
```

```
            addItemToList(modeString);
            // Đưa ra chiều cao màn hình
            sprintf(modeString, “Height=%d”,mode.Height);
            addItemToList(modeString);
            // Đưa ra độ quét
            sprintf(modeString, “Refresh Rate=%d”,mode.RefreshRate);
            addItemToList(modeString);
            }
            return true;
}
```

Đây chỉ đơn giản là một hàm trợ giúp việc xuất số liệu, với đầu vào là một chuỗi, và nó sẽ lưu chuỗi này vào vị trí cuối cùng của vecto. Ở cuối hàm initDirect3D, vector này sẽ chứa toàn bộ thông tin về thiết bị đồ hoạ

```
// Biến adapterDetails là một vector chứa các chuỗi (string), mỗi chuỗi sẽ chứa thông tin
// của các chế độ đồ hoạ khác nhau
std::vector<std::string> adapterDetails;
void addItemToList(std::string item)
{
        adapterDetails.push_back(item);
}
```

Hình 2.8 là hộp thoại chứa các thông tin chi tiết, nó sẽ hiện ra khi bạn chạy chương trình ví dụ này. Bởi vì mỗi người có những card đồ hoạ khác nhau, vì thế mà những chi tiết lấy được cũng thay đổi tuỳ thuộc vào cái máy mà chương trình đang chạy trên đó.

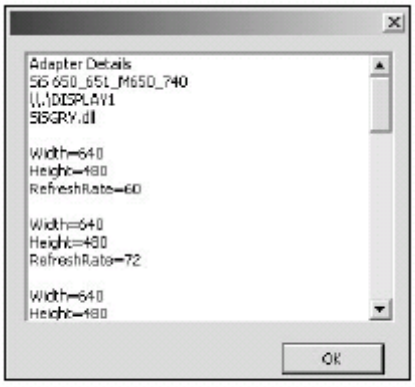


Figure 2.8 Video modes details.

Chú ý:

Thư viện hàm mẫu chuẩn (STL – Standard Template Library) cung cấp cho bạn rất nhiều đối tượng hữu dụng, như kiểu đối tượng string hay vector mà bạn đã từng sử dụng. Việc sử dụng thư viện này sẽ làm đơn giản hoá công việc của bạn bởi vì nó hỗ trợ phát triển trên nhiều nền tảng khác nhau như UNIX hoặc các thiết bị chơi game cá nhân.

Tổng kết chương

Trong chương này chúng ta đã lần lượt đi qua rất nhiều các kiến thức cơ bản, từ làm thế nào để tạo một dự án mới trong Visual C cho tới xác lập và xây dựng một dự án sử dụng nền tảng DirectX. Những ví dụ mà chúng ta đã thực hiện có thể khá đơn giản nhưng nó thực sự là những bước mà chúng ta sẽ phải làm khi xây dựng bất kỳ một dự án nào.

Những vấn đề đã đề cập trong chương này

Trong chương này, chúng ta đã lần lượt đi qua các lý thuyết cơ bản về các phần:

- Làm thế nào để khởi tạo một đối tượng và thiết bị Direct3D.
- Phương thức để xoá hàm hình ứng dụng.
- Thay đổi hàm quản lý sự kiện ứng dụng, bắt các sự kiện chính phục vụ cho quá trình phát triển một ứng dụng game.
- Thực hiện các quá trình khai báo cho một ứng dụng sử dụng DirectX.
- Làm thế nào để nhận biết khả năng hỗ trợ của các thiết bị đồ hoạ của hệ thống.

Trong chương tiếp theo, chúng ta sẽ đề cập tới một số vấn đề về bề mặt và hoạt cảnh cũng như làm thế nào để xây dựng đối tượng chuyển động trong game.

Hỏi đáp

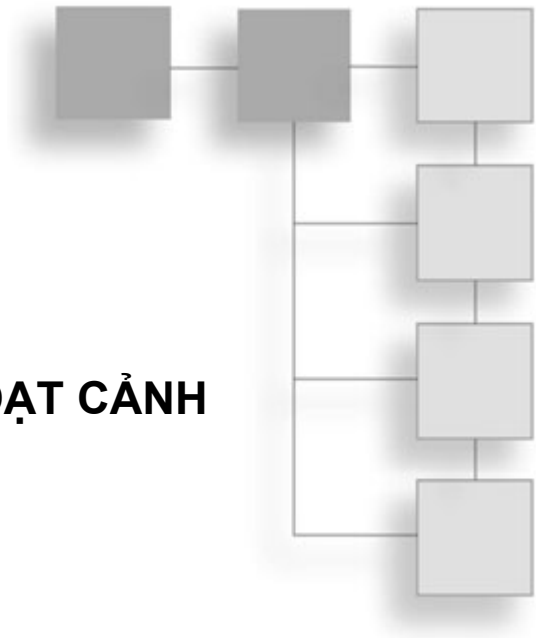
Những câu trả lời của phần Hỏi đáp và Bài tập tự làm bạn có thể tra cứu trong phụ lục A "Giải đáp các câu hỏi cuối mỗi chương" để so sánh.

1. Đối tượng DirectX đầu tiên mà bạn cần phải tạo cho bất kỳ ứng dụng nào là gì?
2. Chức năng chính của hàm GetAdapterCount?
3. Xác lập D3DFMT_A8R8G8B8 định nghĩa sử dụng bao nhiêu bit màu cho mỗi một điểm ảnh trên màn hình?
4. Hàm nào trong DirectX có khả năng xoá và xác lập mầu nền hiển thị màn hình?
5. Hàm nào bạn sử dụng để nhận biết số lượng kiểu độ phân giải màn hình mà thiết bị đồ hoạ hỗ trợ?

Bài tập tự làm

1. Thay đổi ví dụ 2 trên CD-ROM sao cho giá trị màu cần xoá là màu xanh lá cây.
2. Cập nhật lại ví dụ 4 trên CD-ROM để tìm các kiểu màu mà thiết bị đồ hoạ hỗ trợ (các giá trị kiểu D3DFORMAT khác mà thiết bị hỗ trợ ngoài D3DFMT_X8R8G8B8).

CHƯƠNG 3



PHÔNG NỀN, KHUNG HÌNH, HOẠT CẢNH

Khung hình (sprite) – nền tảng của kỹ thuật game 2D, thể hoại game vẫn còn chiếm một thị phần lớn trong ngành công nghiệp game hiện nay. Không phải tất cả các game đều cần thiết phải có những hệ thống phần cứng 3D mới nhất mới chạy được; những trò chơi vui nhộn và chiếm ít thời gian như Tetris vẫn được viết hoàn toàn 2D và vẫn tỏ ra rất phổ biến hiện này. Trong chương trình sẽ giới thiệu tới bạn những cách đơn giản nhất để sử dụng DirectX cho quá trình tạo một khung hình cơ sở của game.

Những phần mà bạn sẽ được giới thiệu trong chương này:

- Thế nào là phông nền và làm thế nào để sử dụng nó
- Làm thế nào để thực hiện truy cập tới bộ nhớ đệm (back buffer)
- Làm thế nào để tạo một offscreen surface.
- Làm thế nào để tải một file ảnh vào bộ nhớ một cách đơn giản
- Làm thế nào để tạo và sử dụng sprites (những khung hình)
- Làm thế nào để tạo một hoạt cảnh sprites.
- Làm thế nào để xác lập các thông số thời gian sao cho hiệu quả thể hiện hoạt cảnh tốt nhất.

You've just touched the surface

Surfaces là một phần của DirectX. Surface bao gồm những vùng bên trong bộ nhớ được sử dụng để lưu trữ dữ liệu thông tin ảnh. Chúng có thể lưu trữ các bức ảnh hay vật liệu tô phục vụ quá trình hiển thị sau này. Surface được lưu trữ trong bộ nhớ dưới dạng từng khối liên tiếp nhau và thông thường là trên card đồ hoạ, tuy nhiên đôi khi nó cũng có thể được lưu trữ trên bộ nhớ chính của máy.

Chương này bao gồm hay vấn đề chính: bộ đệm hiển thị và bộ đệm nền (offscreen).

Bộ đệm hiển thị

Có hai dạng bộ nhớ đệm mà bạn cần xem xét: bộ đệm chính (front buffer) và bộ đệm phụ (back buffer). Cả hai đều là vùng bộ nhớ để ứng dụng game của bạn thực hiện các thao tác vẽ lên nó.

Bộ đệm chính là những vùng bề mặt có thể nhìn trên cửa sổ ứng dụng game. Tất cả những gì bạn có thể thấy trong các ứng dụng windows đều sử dụng bộ đệm chính hay vùng vẽ. Trong chế độ chạy toàn màn hình, vùng bộ nhớ đệm chính được mở rộng ra và chiếm toàn bộ màn hình. Vùng đệm thứ hai là vùng đệm phụ (vùng đệm nền). Như chúng ta đã đề cập ở trên, vùng đệm phụ - back buffer là nơi bạn thực hiện tất cả các thao tác vẽ. Sau khi quá trình vẽ hoàn tất, bạn sẽ sử dụng hàm Present để thể hiện chúng (copy dữ liệu từ vùng đệm phụ lên vùng đệm chính).

Vùng đệm phụ được tạo trong quá trình gọi tới hàm CreateDevice bằng cách xác lập tham số BackBufferCount với kiểu dữ liệu D3DPRESENT_PARAMETERS.

Chú ý:

Việc thực hiện vẽ trực tiếp lên bộ đệm chính sẽ làm cho hình ảnh thể hiện bị nháy và giật. Các đối tượng đồ hoạ thông thường phải được vẽ lên bộ đệm phụ trước, sau đó gọi tới hàm Present để thể hiện.

Offscreen Surfaces

Offscreen surfaces là vùng trên bộ nhớ đồ hoạ hay hệ thống được dùng để lưu trữ những đối tượng hình hoạ mà game cần sử dụng. Có thể lấy vì dụ, nếu bạn đang tiến hành khởi tạo một game nhập vai, bạn sẽ cần phải có một vùng để lưu trữ những dữ liệu để thể hiện nhiều dạng địa hình khác nhau, hay những hình ảnh cho nhận vật của bạn. Offscreen surface có lẽ là sự lựa chọn tốt nhất cho công việc này.

Thông thường các hình ảnh sử dụng trong DirectX đều là dạng bitmaps. Hình minh hoạ 3.1 kế bên là ví dụ cho các ảnh bitmaps có thể được sử dụng trong ứng dụng game của bạn để thể hiện các dạng địa hình khác nhau.

Hình 3.1 Các ảnh hay được sử dụng trong game nhập vai

Chú ý:

Một vài thiết bị đồ hoạ cũ chỉ hỗ trợ tạo offscreen surfaces phù hợp với bộ đệm (primary buffer). Các thiết bị đồ hoạ mới hơn cho phép bạn có thể tạo được các surface lớn hơn.

Offscreen surface, được sử dụng thông qua giao diện IDirect3DSurface9 và được tạo bởi lời gọi tới hàm CreateOffscreenPlainSurface. Bạn phải gọi tới hàm này cho mỗi đối tượng surface mà bạn muốn sử dụng. Hàm CreateOffscreenPlainSureface này được định nghĩa như sau:

```
HRESULT CreateOffscreenPlainSurface(
                UINT Width, // bề ngang của surface
                UINT Height, // chiều cao của the surface
                D3DFORMAT Format, // đối tượng có kiểu D3DFORMAT
                DWORD Pool, // bộ nhớ dùng chung pool
                IDirect3DSurface9** ppSurface, // con trỏ đối tượng kết quả trả về
```

```
                    HANDLE* pHandle // luôn luôn có giá trị NULL
);
```

Hàm CreateOffscreenPlainSurface yêu cầu 6 tham số đầu vào:

- **Width**. Tham số này xác lập bề rộng tính theo pixel của bộ đệm.
- **Height**. Tham số xác lập chiều cao tính theo pixel của bộ đệm.
- **Format**. Tham số định dạng kiểu bộ đêm có cấu trúc D3DFORMAT.
- **Pool**. Vùng bộ nhớ sử dụng lưu trữ surface. Bạn có thể lựa chọn một trong các kiểu sau đây.
  - **D3DPOOL_DEFAULT**. Hệ thống sẽ lựa chọn vùng nhớ phù hợp nhất (trên thiết bị đồ hoạ hoặc bộ nhớ hệ thống) để lưu chữ surface.
  - **D3DPOOL_MANAGED**. Dữ liệu sẽ được copy vào bộ nhớ chính khi cần thiết.
  - **D3DPOOL_SYSTEMMEM**. Surface sẽ được khởi tạo trên bộ nhớ hệ thống.
  - **D3DPOOL_SCRATCH**. Quá trình khởi tạo sẽ được thực hiện trên bộ nhớ hệ thống nhưng không thể truy cập trực tiếp bằng DirectX.
- **PpSurface**. Đây là con trỏ trỏ tới đối tượng có giao diện IDirect3DSurface9. Biến này dùng để quản lý đôi tượng surface sau khi được tạo ra.
- **pHandle**. Đây là tham số dùng để dự phòng và nó luông được gán giá trị NULL.

Ví dụ mẫu dưới đây sẽ minh hoạ quá trình gọi tới hàm CreateOffscreenPlainSurface. Trong đó đối tượng surface sẽ có độ phân giải 640x480 và định dạng kiểu thể hiện là D3DFMT_X8R8G8B8.

```
hResult = CreateOffscreenPlainSurface(
                        640, // Bề rộng của surface được tạo ra
                        480, // Chiều cao của surface được tạo ra
                        D3DFMT_X8R8G8B8, // Định dạng thể hiện của surface
                        D3DPOOL_DEFAULT, // kiểu dữ liệu bộ nhớ pool được sử dụng
                        &surface, // con trỏ lưu surface đã được tạo ra
                        NULL); // tham số dự phòng, mặc định luôn gán cho giá trị NULL
// Kiểm tra xem kết quả trả về của hàm có thành công hay không
if (FAILED(hResult))
        return NULL;
```

Tải ảnh Bitmap cho Surface

Bởi vì định dạng ảnh kiểu Bitmap rất hay được sử dụng trong các ứng dụng đồ hoạ Windows. Chính vì thế chúng ta cũng sẽ sử dụng định dạng này trong các ví dụ tiếp theo. DirectX cung cấp ta khá nhiều hàm trong thư viện D3DX để trợ giúp chúng ta có thể dễ dàng tải nhanh những bức ảnh này để thực hiện quá trình vẽ tiếp theo.

Chú ý:

Có rất nhiều kiểu định dạng ảnh đang được sử dụng trong quá trình phát triển game hiện nay. Thông thường một số công ty thường hay sử dụng kiểu định dạng Bitmap hoặc Targa, tuy nhiên cũng rất nhiều công ty tự xây dựng cho mình một kiểu định dạng ảnh khác nhau nhằm bảo vệ những dữ liệu ảnh của họ. Ngăn cản người dùng hoặc những người phát triển game khác có thể chỉnh sửa hoặc sử dụng lại dữ liệu ảnh đó.

Những chức năng chính của thư viện D3DX

Hệ thống thư viện D3DX bao gồm tập hợp những hàm thường hay dùng được Microsoft cung cấp kèm them với bộ DirectX SDK. Nó bao gồm tập hợp các hàm với chức năng:

- Các hàm quản lý quá trình đọc các dữ liệu ảnh
- Đọc và xử lý làm mịn các đối tượng 3D
- Các hàm thực hiện hiệu ứng shader
- Các hàm phục vụ quá trình biến đổi và xoay các đối tượng

Bạn có thể sử dụng những hàm trong thư viện D3DX này bằng cách thêm dòng lệnh khai báo #include <d3dx9.h> và liên kết tới tệp tin thư viện d3dx9.lib

Hàm D3DXLoadSurfaceFromFile được dùng để đọc dữ liệu ảnh bitmap từ tệp tin vào vùng đệm offscreen surface. Cấu trúc lời họi hàm có dạng như sau:

```
HRESULT D3DXLoadSurfaceFromFile(
                    LPDIRECT3DSURFACE9 pDestSurface,
                    CONST PALETTEENTRY* pDestPalette,
                    CONST RECT* pDestRect,
                    LPCTSTR pSrcFile,
                    CONST RECT* pSrcRect,
                    DWORD Filter,
                    D3DCOLOR ColorKey,
                    D3DXIMAGE_INFO* pSrcInfo
);
```

Hàm D3DLoadSurfaceFromFile này yêu cầu 8 tham số đầu vào:

- **pDestSurface**. Con trỏ đối tượng surface quản lý các ảnh bitmap được tải vào.
- **pDestPalette**. Con trỏ đối tượng kiểu PALLETTEENTRY. Tham số này chỉ sử dụng cho loại ảnh bitmapp 256 màu. Đối với loại ảnh 16-, 24-, 32-bit màu tham số này phải được xác lập là NULL.
- **pDestRect**. Con trỏ đối tượng có cấu trúc RECT dùng để thể hiện vùng chữ nhật của surface mà ảnh bitmap sẽ tải vào.
- **pSrcFile**. Tên tệp tin ảnh bitmap được tải vào (bao gồm cả đường dẫn nếu tệp tin ảnh khác thư mục với tệp tin chương trình).
- **pSrcRect**. Con trỏ đối tượng kiểu RECT lưu trữ vị trí vùng dữ liệu ảnh gốc sẽ được tải vào cho đối tượng surface.
- **Filter**. Tham số có kiểu D3DX_FILTER dùng để xác định kiểu bọ lọc được sử dụng trong quá trình tải ảnh bitmap gốc.
- **ColorKey**. Đối tượng kiểu D3DCOLOR nhằm xác lập giá trị màu được gán kiểu thể hiện "trong xuốt" (transparency color). Giá trị mặc định của tham số là 0.
- **pSrcInfo**. Con trỏ đối tượng kiểu D3DIMAGE_INFO chứa các thông tin thuộc tính của tệp tin ảnh bitmap như chiều cao, bề ngang và chất lượng màu của một điểm ảnh (số lượng màu mà một điểm ảnh có thể thể hiện – được tính bằng -bit).

Sau đây là một ví dụ đơn giản gọi tới hàm D3DLoadSurfaceFromFile, nó thực hiện tải ảnh bitmap từ tệp tin test.bmp vào trong vùng đệm offscreen surface. Chú ý là bạn phải tạo đối tượng surface lưu trữ bằng hàm CreateSurfaceFromFile trước khi gọi tới hàm này.

```
IDirect3DSurface9* surface;
hResult = D3DXLoadSurfaceFromFile( surface,
                                   NULL,
                                   NULL,
                                   “test.bmp”,
                                   NULL,
                                   D3DX_DEFAULT,
                                   0,
                                   NULL );
if ( FAILED( hResult ) )
        return NULL;
```

Sau lời gọi trên đây, dữ liệu toàn bộ ảnh bitmap trong tệp tin test.bmp sẽ được tải vào bộ nhớ và sẵn sàng để bạn sử dụng.

Sử dụng DirectX để thể hiện một hình ảnh

Chúng ta đã học cách tạo một surface cũng như làm thế nào để tải một ảnh bitmap vào trong nó, bây giờ là lúc chúng ta sẽ thể hiện nó. Để làm được điều này, bạn phải tạo một số thay đổi trong hàm Render mà chúng ta đã tạo trước đó

Trong phần trước, chúng ta đã xây dựng hàm Render như đoạn mã minh hoạ dưới đây:

```
/*****************************************************************************
* Render(void)
*****************************************************************************/
void Render(void)
{
        // Kiểm tra xem đối tượng Direct3D device đã thực sự được khởi tạo hay chưa.
        if( NULL == pd3dDevice )
                return;
        // Xoá bộ đệm màn hình (back buffer) bằng màu xanh nước biển
        pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
        // Thể hiện hình ảnh từ dữ liệu trên bộ nhớ đệm màn hình
        pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Để hiển thị ảnh bitmap đó lên màn hình, bạn cần phải sử dụng hàm StretchRect, hàm này sẽ thực hiện việc sao chép và kéo dãn, thay đổi tỷ lệ hình ảnh nếu 2 vùng chữ nhật lưu trữ ảnh gốc và ảnh đích có kích thước khác nhau.

Hàm StretchRect này được định nghĩa như sau:

```
HRESULT StretchRect(
                IDirect3DSurface9 *pSourceSurface,
                CONST RECT *pSourceRect,
                IDirect3DSurface9 *pDestSurface,
                CONST RECT *pDestRect,
                D3DTEXTUREFILTERTYPE Filter
);
```

Các tham số đầu vào của hàm StretchRect này bao gồm:

- **pSourceSurface**. Con trỏ đối tượng surface quản lý các ảnh bitmap được tải vào.

- **pSourceRect**. Con trỏ kiểu RECT chứa dữ liệu vùng được sao chép. Nếu tham số này là NULL, toàn bộ dữ liệu surface gốc sẽ được sao chép.
- **pDestSurface**. Con trỏ chứa đối tượng surface đích. Trong hầu hết các trường hợp, nó là con trỏ của bộ đệm phụ back buffer.
- **pDestRect**. Con trỏ kiểu RECT chứa dữ liệu vùng thể hiện đối tượng được sao chép lên surface đích. Tham số này có thể là NULL nếu bạn kô muốn xác lập vùng kết xuất.
- **Filter**. Kiểu lọc sử dụng trong quá trình sao chép. Bạn có thể xác lập giá trị này là D3DTEXF_NONE nếu không muốn xác lập kiểu lọc.

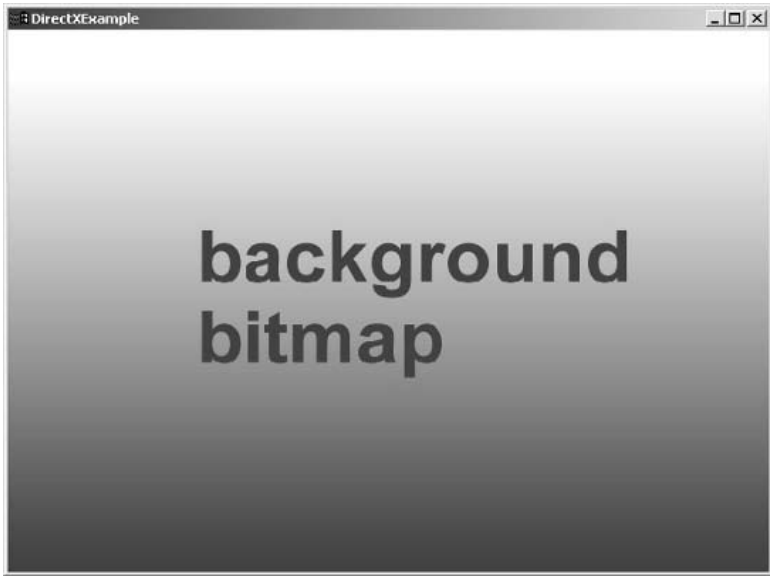
Có lẽ bạn sẽ tự hỏi làm thế nào để có thể lấy được con trỏ chứa dữ liệu của bộ đệm back buffer surface. Hàm để thực hiện chức năng này có tên là GetBackBuffer, cấu trúc của nó có dạng như sau:

```
GetBackBuffer( 0,  // giá trị thể hiện kiểu cháo đổi
               0, // chỉ số của bộ đệm
                  // 0 nếu chỉ có một bộ đệm được sử dụng
               D3DBACKBUFFER_TYPE_MONO, // một đối số định kiểu
               &backbuffer); // đối tượng trả về có kiểu IDirect3DSurface9
```

Kết hợp thêm các lời gọi tới hàm StretchRect và GetBackBuffer, hàm Render của chúng ta lúc này sẽ có dạng tương tự dưới đây:

```
/*****************************************************************************
* Render
*****************************************************************************/
void Render(void)
{
        // Con trỏ bộ đệm back buffer
        IDirect3DSurface9* backbuffer = NULL;
        // Kiểm tra đối tượng Direct3D device đã tồn tại
        if( NULL == pd3dDevice )
                return;
        // Xoá toàn bộ bộ đệm về màu xanh nước biển
        pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
        // Lấy con trỏ bộ đệm back buffer
        pd3dDevice->GetBackBuffer( 0,
                                   0,
                                   D3DBACKBUFFER_TYPE_MONO,
                                   &backbuffer );
        // Sao chép toàn bộ dữ liệu offscreen surface vào bộ đệm
        pd3dDevice->StretchRect( srcSurface,
                                 NULL,
                                 backbuffer,
                                 NULL,
                                 D3DTEXF_NONE );
        // Thể hiện hình ảnh từ bộ đệm lên màn hình
        pd3dDevice->Present ( NULL, NULL, NULL, NULL );
}
```

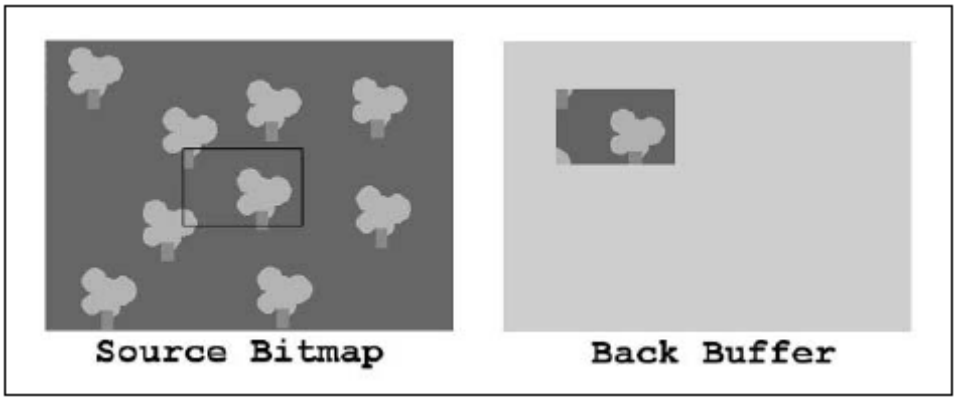
Bạn có thể tìm thấy mã nguồn của ví dụ này trong thư mục chapter3\example1 trên CD-ROM. Biên dịch và chạy ứng dụng, một cửa sổ chương trình sẽ xuất hiện có dạng sau:



Hình 3.2 Hiển thị ảnh nền cho cửa sổ ứng dụng

Xem xét lại hàm StretchRect

Trong phần trước, chúng ta đã sử dụng hàm StretchRect để sao chép toàn bộ ảnh offscreen surface vào bộ đệm, nhưng đó không phải là toàn bộ chức năng hữu dụng nhất của hàm này. StretchRect còn cho phép bạn sao cheo một hoặc nhiều khung hình nhỏ của ảnh offscreen surface, nó cho phép một surface có thể tạo nền từ nhiều hình nhỏ hơn. Ví dụ, một ảnh offscreen surface có thể chứa rất nhiều khung hình chuyển động nhỏ của một nhân vật, hoặc các hình ảnh nhỏ để tạo nên một puzzle game. Hàm StretchRect có 2 tham số - pSourceRect và pDesRect – dùng để xác định vùng dữ liệu sẽ copy từ đâu đến đâu. Hình 3.3 sẽ minh hoạ việc sử dụng chức năng này.



Hình 3.3 Ảnh bên tay trái là ảnh gốc và hình chữ nhật miêu tả vùng dữ liệu sẽ được sao chép. Ảnh bên phải mô tả vị trí và vùng ảnh được sao chép lên bộ đệm.

Trong ví dụ tiếp theo chúng ta sẽ sử dụng chức năng này để thể hiện một thông điệp lên màn hình từ dữ liệu ảnh gốc chứa toàn bộ các chữ trong bảng chữ cái. Hình 3.4 minh hoạ ảnh dữ liệu gốc này, như bạn có thể thấy tất cả các chữ cái được lưu trong các vùng chữ nhật có kích thước giống nhau. Việc xác lập lưu các chữ cái với cùng một kích cỡ sẽ giúp chúng ta tiện lợi hơn rất nhiều trong quá trình xử lý và tìm tới chữ cái cần thiết một cách nhanh nhất.

Bởi vì chúng ta sẽ phải sao chép nhiều khung hình nên chúng ta sẽ phải gọi tới hàm StretchRect nhiều lần. Để mọi thứ trở nên đơn giản, chúng ta sẽ đặt tất cả những lời gọi đó vào trong một vòng lặp bên trong hàm Render.

```
/*****************************************************************
* Render
*****************************************************************/
void Render(void)
{
	int letterWidth=48; // Thông số bề ngang mặc đinh của một ô chữ cái
	int letterHeight=48; // chiều cao mặc định của một ô chữ cái
	int destx = 48; // Toạ độ X của đểm trên cùng phía bên trái của chữ cái đầu tiên
	int desty = 96; // Toạ độ Y của đểm trên cùng phía bên trái của chữ cái đầu tiên
	// Biến chứa con trỏ bộ đệm
	IDirect3DSurface9* backbuffer = NULL;
	// Kiểm tra đối tượng Direct3D device
	if( NULL == pd3dDevice )
		return;
	// Xoá bộ đệm bằng màu xanh nước biển
	pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
	D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
	// Lấy con trỏ của bộ đệm
	pd3dDevice->GetBackBuffer(0,0,D3DBACKBUFFER_TYPE_MONO, &backbuffer);
	// Xác lập biến chứa vị trí chữ cái đang được xem xét
	int count=0;
	// Thực hiện vòng lặp trên từng ký tự của chuỗi thông điệp đầu vào
	for ( char *c = message; c != " "; c++ )
	{
		RECT src; // Dữ liệu vùng chữ nhật gốc và kết xuất
		RECT dest;
		int srcY = ( ( ( *c - 'A' ) / 6 ) ) * letterHeight; // Tìm toạ độ vùng dữ liệu sao chép gốc
		int srcX = ( ( ( *c - 'A' ) %7 ) * letterWidth );
		src.top = srcY ;
		src.left = srcX;
		src.right = src.left + letterWidth;
		src.bottom = src.top + letterHeight;
		// Tìm toạ độ vùng dữ liệu sẽ được sao chép tới
		dest.top = desty;
		dest.left = destx + ( letterWidth * count );
		dest.right = dest.left + letterWidth;
		dest.bottom = dest.top + letterHeight;
		// Tăng biến đếm lên 1
		count++;
		// sao chép dữ liệu vào bộ đệm
		pd3dDevice->StretchRect( srcSurface, // Dữ liệu surface gốc
					src, // Vùng dữ liệu muôn sao chép
					backbuffer, // Dữ liệu được sao chép vào
					dest, // vùng dữ liệu được sao chép
					D3DTEXF_NONE); // kiểu bộ lọc sử dụng
	}
	// Thể hiện dữ liệu từ bộ đệm lên màn hình
	pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Kết quả của ví dụ trên là thể hiện dòng chữ "HELLO WORLD" và đem lại cho người dùng cảm giác của một bức thư tống tiền :). Hình 3.5 minh hoạ kết quả kết xuất của chương trình. Bạn có thể tìm mã nguồn đầy đủ trong thư mục chapter3\example2.

Trước khi vòng lặp được thực hiện đọc từng chữ cái của thông điệp, nó phải được định nghĩa trước ở bên ngoài hàm render có dạng như sau:

```
char *message = “HELLO WORLD”;
```

Trong mỗi bước nhảy của vòng lặp, chúng ta sẽ tiến hành lọc trên từng chữ cái một. Ví dụ, trong lần lặp đầu tiên, chúng ta sẽ chỉ làm việc với chữ cái *H* trong từ “HELLO”. Đoạn mã tiếp theo sẽ tính toán vùng chữ nhật gốc bao gồm toạ độ X và Y của đỉnh trên cùng góc bên trái của hình.

```
int srcY = ( ( ( *c - ‘A’ ) / 6 ) ) * letterHeight;
int srcX = ( ( ( *c - ‘A’ ) %7 ) * letterWidth);
```

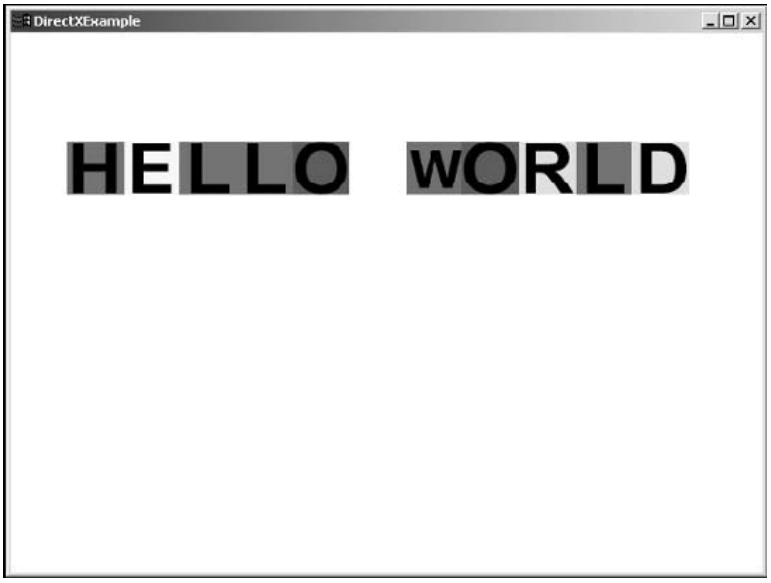
Sau khi chúng ta đã có toạ độ của điểm này, chúng ta sẽ biết được đoạ độ điểm dưới cùng bên phải của hình chữ nhật chứa chữ cái đó thông qua chiều cao và chiều rộng của nó.

```
src.top = srcY ;
src.left = srcX;
src.right = src.left + letterWidth;
src.bottom = src.top + letterHeight;
```

Tiếp đến chúng ta sẽ phải xác định vị trí mà chữ cái sẽ được đặt trên bộ đệm.

```
dest.top = desty;
dest.left = destx + ( letterWidth * count );
dest.right = dest.left + letterWidth;
dest.bottom = dest.top + letterHeight;
```

Chúng ta đã xác lập biến cout dùng để kiểm tra xem bao nhiêu chữ cái đã được vẽ trên màn hình. Thông qua biến count này, chúng ta sẽ tính toán được toạ độ điểm của các chữ cái cần chèn.



Hình 3.5 Hình minh hoạ sử dụng ảnh bitmap các chữ cái để thể hiện chuỗi, Hello world.

Chú ý:

Bạn cũng có thể sử dụng hàm StretchRect để thực hiện quá trình kéo dãn, phóng to một hình ảnh trong quá trình sao chép. Nếu vùng chữ nhật đích có kích thước khác: lớn hơn hay nhỏ hơn thì hình ảnh đó sẽ tự động điều chỉnh lại sao cho được phủ đầy vùng ảnh kết xuất đó.

Sprites

Như đã đề cập ở trong phần trước, bạn có thể thực hiện quá trình sao chép những vùng ảnh giữa các surfaces. Trong ví dụ thứ 2, chúng ta đã thể hiện một dòng thông bao từ các ảnh bitmap. Sử dụng phương pháp tương tự, chúng ta cũng sẽ có thể xây dựng được một hệ thống để thể hiện các sprites.

Sprites là những đối tượng đồ hoạ dạng 2D và nó thường được sử dụng trong các trò chơi dưới dạng hình ảnh của các nhân vật hay bất kỳ một đối tượng nào. Ví dụ, trong chế độ nền của game, một sprite có thể được thể hiện quá trình nhân vật di chuyển trên màn hình. Sprites đơn giản là một chuỗi các khung hình của hoạt cảnh về một nhân vật hay đối tượng nào đó trong game, nó có thể được di chuyển bởi người chơi, có thể tương tác với các đối tượng khác trong thế giới game đó. Trong phần này chúng ta sẽ đi qua các khái niệm cơ bản cũng như học cách tạo và sử dụng chúng trong game.

Chú ý:

Một khung hình đơn giản chỉ là một hình ảnh trong chuỗi các hình ảnh của một hoạt cảnh. Việc thể hiện liên tiếp các hình ảnh này sẽ tạo nên hiệu ứng chuyển động tương tự trong kỹ thuật làm film.

Một sprite cần phải có những gì?

Điều đầu tiên mà tất cả các sprites đều cần đó là hình ảnh để thể hiện. Hình ảnh này có thể được sử dụng làm một hay nhiều khung hình thể hiện trong một hoạt cảnh.

Sprites cũng cần một thông số khác đó là vị trí mà sprites được hiển thị trên màn hình. Giá trị này thường là tham số X, Y trong hệ toạ độ.

Để có thể sử dụng được trong các trò chơi, sprites có thể còn phải lưu trữ thêm một vài thông tin khác nữa tuy nhiên 2 thông số trên thực sự cần thiết cho bất kỳ một sprites nào.

Mã nguồn mô tả đối tượng sprite

Cấu trúc của đối tượng sprite sẽ lưu giữ toàn bộ thông tin về từng sprite mà bạn muốn tạo, cấu trúc cơ bản có thể có dạng sau:

```
struct {
        RECT sourceRect;
        // Vùng lưu giữ vị trí của sprit (trong offscreen surface)
        int X; // toạ độ X của sprite trên màn hình
        int Y; // toạ độ Y của sprite trên màn hình
} spriteStruct ;
```

Trong cấu trúc spriteStruct ở trên, dữ liệu ảnh của sprite được xác định thông quá biến có kiểu cấu trúc RECT. Biến sourceRect này nắm giữ vị trí của sprite trong ảnh gốc.

Giá trị toạ độ X và Y được sử dụng là giá trị kiểu nguyên. Bởi vì trong ví dụ này chúng ta chỉ hỗ trợ độ phân giải 640x480, nên giá trị kiểu nguyên là đủ để lưu trữ toạ độ sprite.

Những gì chúng ta đã làm ở trên chỉ là một xác lập ban đầu hết sức đơn giản cho một sprite. Tiếp đến chúng ta sẽ thực hiện quá trình thể hiện chúng lên màn hình.

Tạo một đối tượng Sprite

Để tạo một sprite, bạn sẽ cần phải sử dụng một vài hàm mà chúng ta đã đề cập trước đó.

- D3DXLoadSurfaceFromFile
- CreateOffscreenPlainSurface
- StretchRect

Mỗi một hàm được liệt kê ở trên đều có một chức năng riêng trợ giúp quá trình tạo và sử dụng sprites. Hàm D3DLoadSurfaceFromFile thực hiện quá trình đọc dữ liệu ảnh của sprites từ tệp tin, hàm CreateOffscreenPlainSurface tạo một vùng trên bộ nhớ để lưu trữ những hình ảnh bạn cần sử dụng và hàm StretchRect trợ giúp hiện thị hình ảnh lên màn hình ứng dụng.

Đọc dữ liệu ảnh của Sprites

Bạn có thể sử dụng hàm D3DXLoadSurfaceFromFile để đọc dữ liệu ảnh cho một đối tượng IDirect3DSurface9 đã được tạo trước đó bằng hàm CreateOffscreenPlainSurface.

Để thực hiện quá trình trên, chúng ta sẽ đặt chúng vào trong một hàm duy nhất có tên là getSurfaceFromBitmap. Hàm này chỉ có một tham số đầu vào duy nhất là một chuỗi string chứa tên của file cần đọc.

```
/************************************************************
* getSurfaceFromBitmap
************************************************************/
IDirect3DSurface9* getSurfaceFromBitmap(std::string filename)
{
	HRESULT hResult;
	IDirect3DSurface9* surface = NULL;
	D3DXIMAGE_INFO imageInfo; // tạo biến lưu giữ thông tin của ảnh gốc
	// Lấy thông tin chiều rộng, chiều cao của hình ảnh gốc
	hResult = D3DXGetImageInfoFromFile(filename.c_str(), &imageInfo);
	// Chắc chắn quá trình gọi hàm D3DXGetImageInfoFromFile đã thành công
	if FAILED (hResult)
		return NULL;
	// Tạo một offscreen surface lưu giữ liệu ảnh gốc
	hResult = pd3dDevice->CreateOffscreenPlainSurface( width,
					height,
					D3DFMT_X8R8G8B8,
					D3DPOOL_DEFAULT,
					&surface,
					NULL )
	// Chắc chắn quá trình gọi hàm kô thất bại
	if ( FAILED( hResult ) )
		return NULL;
	// Đọc dữ liệu ảnh cho đối tượng surface vừa tạo từ tệp tin
	hResult = D3DXLoadSurfaceFromFile( surface,
					NULL,
					NULL,
					filename.c_str( ),
					NULL,
					D3DX_DEFAULT,
```

```
                                                0,
                                                NULL );
        if ( FAILED( hResult ) )
                return NULL;
        return surface;
}
```

Hàm getSurfaceFromBitmap được sử dụng như sau:

```
IDirect3DSurface9* spriteSurface;
spriteSurface = getSurfaceFromBitmap( “sprites.bmp”);
If (spriteSurface == NULL)
        return false;
```

Trước tiên bạn tạo một biến để lưu một surface mới, sao đó gọi tới hàm getSurfaceFromBitmap với tham số là tên của file ảnh bitmap cần đọc.

Bạn nên chú ý kiểm tra kết quả trả về để chắc chắn hàm getSurfaceFromBitmap đã được thực hiện thành công. Hình 3.6 minh hoạ một ảnh có chứa nhiều sprites.

Hình 3.6 Một ảnh bitmap chứa các hình ảnh của sprite

Hàm GetSurfaceFromBitmap sẽ được sử dụng trong toàn bộ các phần tiếtp theo của quấn sách mỗi khi cần thiết.

Khởi tạo các Sprites.

Sau khi bạn đã đọc dữ liệu ảnh của sprite, bây giờ là lúc chúng ta xác lập thông tin chính xác cho từng sprite. Bởi vì chúng ta sẽ sử dụng từng đối tượng surface đơn lẻ để chứa đựng lần lượt tất cả các sprité, một ý tưởng tốt là đặt tất cả những đoạn mã khởi tạo cho từng sprite này vào trong một vòng lặp for. Hàm initSpites dưới đây minh hoạ kỹ thuật này.

```
#define SPRITE_WIDTH 48
#define SPRITE_HEIGHT 48
#define SCRN_WIDTH 640
#define SCRN_HEIGHT 480
/*****************************************************************************
* bool initSprites(void)
*****************************************************************************/
bool initSprites(void)
{
        // Vòng lặp thực hiện khởi tạo cho 10 sprite
        for (int i = 0; i < 10; i++ )
        {
                spriteStruct[i].srcRect.top = 0;
                spriteStruct[i].srcRect.left = i * SPRITE_WIDTH;
                spriteStruct[i].srcRect.right = spriteStruct[i].srcRect.left +
                                SPRITE_WIDTH;
                spriteStruct[i].srcRect.bottom = SPRITE_HEIGHT;
                spriteStruct[i].posX = rand()% SCRN_WIDTH – SPRITE_WIDTH;
                spriteStruct[i].posY = rand()% SCRN_HEIGHT – SPRITE_HEIGHT;
        }
        return true;
}
```

Vòng lặp for sẽ thực hiện quá trình khởi tạo trong 10 lần lặp, kết quả trả về là 10 sprite khác nhau được đọc.

Bên trong vòng lặp, biến srcRect phải được xác lập và thay đổi. Nó lưu giữ vị trí của sprite bên trong ảnh gốc. Tham số cuối cùng X và Y lưu toạ độ hiển thị của sprite, trong trường hợp này chúng ta sẽ đặt một vị trí ngẫu nhiên để hiển thị sprite trên màn hình.

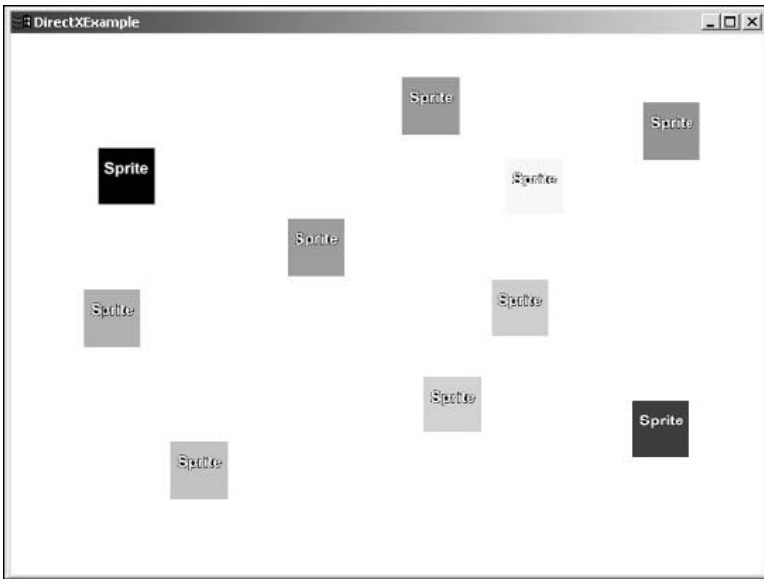
Hiển thị Sprites

Đến thời điểm này, công việc cuối cùng bạn phải làm là thể hiện các sprites đó lên màn hình. Một lần nữa, chúng ta sẽ thực hiện một vài thay đổi trong hàm render. Lần này, vòng lặp for đã được tạo trước đó sẽ gọi tới hàm StretchRect nhiều lần, mỗi một lần đó tương ứng với một sprite được hiển thị.

```
/*****************************************************************************
* Render(void)
*****************************************************************************/
void Render(void)
{
	// Biến lưu con trỏ của bộ đệm
	IDirect3DSurface9* backbuffer = NULL;
	if( NULL == pd3dDevice )
		return;
	// Xoá bộ đệm với màu đen
	pd3dDevice->Clear( 0,
			NULL,
			D3DCLEAR_TARGET,
			D3DCOLOR_XRGB(0,0,0),
			1.0f,
			0 );
	// Lấy về con trỏ của bộ đệm
	pd3dDevice->GetBackBuffer(0,0,D3DBACKBUFFER_TYPE_MONO, &backbuffer);
	// Vòng lặp hiển thị tất cả các sprites
	for ( int i = 0; i < 10; i++ )
	{
		RECT destRect; // Tạo một đối tượng RECT lưu trữ giá trị tạm
		// Xác lập giá trị cho biến RECT vừa khai báo với dữ liệu tương ứng
		// của sprite hiện tại đang được xét
		destRect.left = spriteStruct[i].posX;
		destRect.top = spriteStruct[i].posY;
		destRect.bottom = destRect.top + SPRITE_HEIGHT;
		destRect.right = destRect.left + SPRITE_WIDTH;
		// Hiển thị sprite lên bộ nhớ đệm back buffer
		pd3dDevice->StretchRect( spriteSurface,
				srcRect,
				backbuffer,
				destRect,
				D3DTEXF_NONE);
	}
	// Hiển thị dữ liệu trên bộ đệm lên màn hình
	pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Trong đoạn mãn trên một lần nữa chúng ta đã thực hiện một vòng lặp chạy lần lượt trên 10 sprites. Bên trong vòng lặp chúng ta đã thực hiện tạo và xác lập giá trị cho một biến RECT chứa dữ liệu tạm. Hàm StretchRect sẽ sử dụng biến kiểu RECT mà bạn đã tạo này để yêu cầu DirectX thể hiện đúng toạ độ sprite sẽ được hiển thị. Cuối cùng là lời gọi của hàm StretchRect như đã nói, quá trình này được thực hiện lần lượt từng bước, từng bước. Dữ liệu của sprites sẽ được đặt lên bộ nhớ đệm trước. Hình 3.7 minh hoạ toàn bộ 10 sprites được hiển thị lên màn hình sau khi kết thúc lời gọi tới hàm render.

Bạn có thể tìm thấy toàn bộ mã nguồn của ví dụ này trong thư mục chapter3\example3 trên đĩa CD-ROM đi kèm với sách.



Hình 3.7 Hình ảnh minh hoạ 10 sprites được hiển thị ngẫu nhiên

Di chuyển các Sprites

Các sprites của bạn đã được hiển thị toàn bộ lên màn hình, liệu bạn đã hài lòng với cách hiển thị này? Có lẽ là chưa. Một trong những kỹ thuật nổi bật của một đối tượng sprite là nó có thể di chuyển được. Tôi chắc chắn rằng trò chơi *Sonic the Hedgehog* sẽ chẳng có gì hấp dẫn nếu Sonic chẳng thể di chuyển được. Các Sprites cần phải biết đường đi, khoảng cách và hướng đi mà nó cần phải di chuyển trong mỗi một khung hình.

Để khắc phục vấn đề này, bạn cần khai báo thêm một vài biến khác trong cấu trúc của sprite mà chúng ta đã định nghĩa trước đó.

```
struct {
        RECT srcRect; // Lưu giữ vị trí của sprite trên vùng ảnh gốc
        // Vị trí hiển thị của sprite
        int posX; // Toạ độ của sprite theo phương X
        int posY; // Toạ độ của sprite theo phương Y
        // Khả năng di chuyển trong mỗi một khung hình
        int moveX; // khoảng cách tính theo pixel mà sprite có thể di chuyển theo phương X
        int moveY; // tương tự đối với phương Y
} spriteStruct;
```

Như bạn đã thấy, hai biến moveX và moveY đã được thêm vào. Hai biến này sẽ được sử dụng để lưu giữ giá trị số lượng pixels trên mỗi khung hình mà bạn muốn sprite đó di chuyển. Giá trị moveX và moveY này sẽ được cộng thêm vào cho giá trị posX và posY tương ứng trong quá trình gọi tới hàm StretchRect trên mỗi sprite. Ví dụ như, trên mỗi một khung hình chúng ta có thể sử dụng đoạn mã nguồn sau:

```
for (int i = 0; i < 10; i++)
{
        spriteStruct[ i ].posX += spriteStruct[ i ].moveX;
        spriteStruct[ i ].posY += spriteStruct[ i ].moveY;
}
```

Các sprites sau đó sẽ được gửi tới hàm Render để thực hiện quá trình hiển thị. Trên mỗi khung hình, biến lưu trữ toạ độ sẽ được cập nhật, kết quả là sprite đó sẽ được di chuyển trên màn hình ứng dụng.

Dĩ nhiên, bạn sẽ phải kiểm tra vị trí posX và posY để chắc chắn sprite đó được hiển thị đúng với độ phân giải màn hình. Ví dụ, trong đoạn mã ví dụ trên có thể được thay đổi để đảm bảo các sprites luôn nằm trong vùng hiển thị 640x480 của ứng dụng:

```
for (int i = 0; i < 10; i++)
{
        // Thay đổi toạ độ X của sprite
        spriteStruct[ i ].posX += spriteStruct[ i ].moveX;
        // Kiểm tra xem posX có lớn hơn 640 không
        if (spriteStruct[ i ].posX > SCRN_WIDTH)
        {
                // Nếu giá trị posX lớn hơn thì thực hiện đảo dấu của moveX bằng cách nhân
                // với giá trị -1. Điều này sẽ làm cho sprite sẽ chuyển động theo chiều ngược
                // lại tại các khung hình sau
                spriteStruct[ i ].moveX *= -1;
        }
        // Tương tự đối với toạ độ theo phương Y của sprite
        spriteStruct[ i ].posY += spriteStruct[ i ].moveY;
        // Kiểm tra posY với SCRN_HEIGHT (=480)
        if (spriteStruct[ i ].posY > SCRN_HEIGHT)
        {
                spriteStruct[ i ].moveY *= -1;
        }
        // Bởi vì sprite cũng có thể chuyển động ngược lại nên một ý tưởng tốt là chúng ta
        // sẽ kiểm tra cả trường hợp toạ độ của sprite có nhỏ hơn 0 không
        if (spriteStruct[ i ].posX < 0)
        {
                // Nếu trường hợp này xảy ra thì đảo chiều chuyển động của sprite
                spriteStruct[ i ].moveX *= -1;
        }
        // Tương tự đối với posY
        if (spriteStruct[ i ].posY < 0)
        {
                spriteStruct[ i ].moveY *= -1;
        }
}
```

Đoạn mã nguồn trên sẽ làm cho sprite luôn chuyển động trên vùng hiển thị 640x480 của màn hình.

Chú ý:

Nếu trong quá trình ứng dụng game đang chạy mà bạn thay đổi lại kích thước của cửa sổ ứng dụng, bạn phải chắc chắn giá trị posX và posY phải được kiểm tra lại để chắc chắn sprite vẫn còn nằm trong vùng hiển thị.

Tạo các sprite chuyển động

Trong phiên bản trước của sprite chúng ta đã cập nhật để nó hỗ trợ khả năng di chuyển của sprite trên màn hình, nhưng thực sự thì những sprites này vẫn chưa hấp dẫn. Những sprites này chỉ thể hiện một hình ảnh tĩnh duy nhất. Trong phần tiếp theo, bạn sẽ thực

hiện chèn thêm nhiều khung hình của sprite hơn để tạo một sprite chuyển động thực sự sinh động.

Để thực hiện được khả năng này, chúng ta sẽ tiến hành cập nhật lại cấu trúc của sprite, mã nguồn mới sẽ được liệt kê dưới đây và hình minh hoạ 3.8 ở bên là dữ liệu của một sprite mẫu.

Hình 3.8. Dữ liệu ảnh của một sprite chuyển động

```
struct {
        RECT srcRect; // lưu trữ vị trí vùng ảnh gốc của sprite
        // dữ liệu về toạ độ của sprite
        int posX; // toạ độ theo phương X
        int posY; // toạ độ theo phương Y
        // dữ liệu di chuyển của sprite
        int moveX; // khoảng cách di chuyển tính bằng pixels theo phương X
        int moveY; // khoảng cách di chuyển tính bằng pixels theo phương Y
        // dữ liệu animation sprite
        int numFrames; // số lượng khung hình của animation sprite
        int curFrame; // khung hình đang được hiển thị của sprite
} spriteStruct;
```

Để hỗ trợ khả năng hiển thị chuyển động của sprite, chúng ta thêm vào 2 biến:

- **numFrames.** Số lượng khung hình của một animation sprite.
- **curFrame.** Khung hình hiện tại đang được hiển thị của animation sprite.

Hai biến này sẽ giúp chúng ta quản lý và lưu giữ trạng thái hiển thị các ảnh động của sprite trong vòng lặp hiển thị sprite.

Bởi vì chúng ta đã thêm vào các biến mới trong kiểu cấu trúc của sprite nên chúng ta sẽ phải thay đổi lại mã nguồn hàm initSprite để hỗ trợ nó.

Mã nguồn mới của hàm initSprite được liệt kê dưới đây:

```
/*******************************************************************************
* bool initSprites(void)
*******************************************************************************/
bool initSprites(void)
{
        // Vòng lặp khởi tạo tất cả các sprite
        for (int i=0; i < 10; i++)
        {
                // Xác lập vị trí dữ liệu chứa sprite trên ảnh gốc
                spriteStruct[i].srcRect.top = 0;
                spriteStruct[i].srcRect.left = i * 64;
                spriteStruct[i].srcRect.right = spriteStruct[i].srcRect.left + 64;
                spriteStruct[i].srcRect.bottom = 23;
                // Xác lập toạ độ hiển thị ngẫu nhiên cho sprite
                spriteStruct[i].posX = rand()%600; //
                spriteStruct[i].posY = rand()%430;
                // Xác lập dữ liệu ảnh động của sprite
                spriteStruct[i].curFrame = 0; // Start at frame 0
                spriteStruct[i].numFrames = 4; // Số lượng khung hình của một sprite
                // Xác lập dữ liệu di chuyển của sprite
                spriteStruct[i].moveX = 1; // Di chuyển sprite theo từng pixel
                // chỉ cho sprite di chuyển theo chiều trái – phải
                spriteStruct[i].moveY = 0; // sprite không thể di chuyển theo phương Y
        }
return true;
```

```
}
```

Bây giờ thì sprite đã được khởi tạo với toạ độ và dữ liệu ảnh động của nó, chúng đã sẵn sàng để bạn hiển thị lên màn hình.

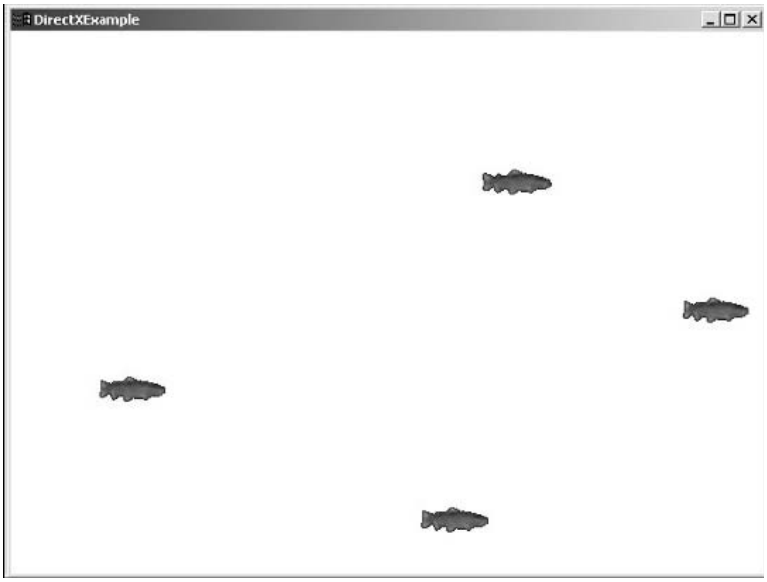
Thể hiện một hình sprite động lên màn hình

Hàm render một lần nữa sẽ được cập nhật để hỗ trợ các dữ liệu mới. Trong mỗi lần hàm này được gọi thì biến curFrame sẽ được tăng lên. Biến này điều khiển khung hình của hình sprite động sẽ được hiển thị. Khi số này lớn hơn số lượng khung hình của một hình sprite động (biến numFrames) thì biến curFrame này sẽ được xác lập lại thành giá trị 0. Phiên bản mới của hàm render có dạng như sau:

```
/*****************************************************************************
* Render(void)
*****************************************************************************/
void Render(void)
{
	// Con trỏ lưu địa chỉ bộ đệm
	IDirect3DSurface9* backbuffer = NULL;
	// Kiểm tra xem đối tượng D3DDevice đã chắc chắn được tạo chưa
	if( NULL = = pd3dDevice )
		return;
	// Xoá bộ đệm bằng màu đen
	pd3dDevice->Clear( 0,
			NULL,
			D3DCLEAR_TARGET,
			D3DCOLOR_XRGB(0,0,0),
			1.0f,
			0 );
	// Lấy về con trỏ chứa đia chỉ bộ đệm
	pd3dDevice->GetBackBuffer(0,0,D3DBACKBUFFER_TYPE_MONO, &backbuffer);
	// Vòng lặp chạy trên toàn bộ sprite
	for ( int i = 0; i < 10; i++ )
	{
		// Tăng biến lưu giá trị khung hình đang được hiện thị của sprite
		if (spriteStruct[ i ].curFrame < spriteStruct[ i ].numFrames)
			spriteStruct[ i ].curFrame++;
		else
		// Nếu giá trị khung hình lớn hơn số lượng khung hình cho phép thì gán thành 0
		spriteStruct[ I ].curFrame = 0;
		// Xác lập vị trí chính xác vùng ảnh lưu trữ khung hình trên ảnh gốc
		spriteStruct[ i ].srcRect.left = spriteStruct[ i ].curFrame * 64;
		spriteStruct[ i ].srcRect.right = spriteStruct[ i ].srcRect.left + 64;
		// Tạo một biến có kiểu RECT để lưu trữ dữ liệu tạm
		RECT destRect;
		// Xác lập dữ liệu cho biến tạm kiểu RECT này
		destRect.left = spriteStruct[i].posX;
		destRect.top = spriteStruct[i].posY;

		// Sprite hình con cá này có chiều cao là 23 pixels
		destRect.bottom = destRect.top + SPRITE_HEIGHT;
		// Bề ngang của sprite là 64 pixels
		destRect.right = destRect.left + SPRITE_WIDTH;
		// Đặt dữ liệu sprite vào bộ đệm
		pd3dDevice->StretchRect (spriteSurface,
					srcRect,
					backbuffer,
					destRect,
```

```
                                                D3DTEXF_NONE);
            }
    // Hiển thị dữ liệu từ bộ đệm lên màn hình
    pd3dDevice->Present( NULL, NULL, NULL, NULL );
    }
```

Nếu bạn biên dịch và chạy ứng dụng với đoạn mã đã được cập nhật ở trên, bạn sẽ thấy một đôi cá đang bơi lội dọc theo chiều ngang màn hình. Hinh 3.9 minh hoạ những con cá mà bạn có thể thấy. Bạn có thể tìm toàn bộ mã nguồn đã được cập nhật ở trên trong thư mục chapter3\example4 của CD-ROM.



Hình 3.9. Hình minh hoạ các hình sprite động là các chú cá

Tại sao các sprite chạy quá nhanh?

Trong quá trình chạy ví dụ trên, bạn có thể thấy rằng những chú cá được hiển thị thông qua 4 khung hình và nó được di chuyển khá nhanh trên màn hình. Nguyên nhân của tình trạng này là chúng ta đã sử dụng kỹ thuật hoạt hoạ để thể hiện các khung hình. Bởi vì không có cách nào để tăng hay giảm chuyển động hoạt hình của sprite, nên khả năng hiển thị phụ thuộc chủ yếu vào hệ thống. Trên một máy tính tốc độ cao, những chú cá có vẻ như bơi lội, di chuyển rất là nhanh, ngược lại trên các hệ thống quá chậm thì ta lại cảm thấy những di chuyển này có thể khá đứt quãng.

Trong phần tiếp theo, chúng ta sẽ đề cập tới vấn đề làm thế nào để giảm tốc độ của các hình sprite động cũng như đảm bảo các khung hình sẽ được hiển thị chính xác thông qua một bộ đếm thời gian timer.

Hiển thị một hình sprite động chính xác

Để tạo một hoạt cảnh có thể chuyển động mượt mà thì ứng dụng game của bạn phải là ứng dụng là ứng dụng được quyền ưu tiên nhất trên hệ thống. Bằng cách sử dụng các timer, các hoạt cảnh chuyển động có thể được xác lập để xuất hiện tại những thời điểm xác định. Ví dụ, nếu bạn muốn chạy hoạt cảnh với tốc độ 30 khung hình trong một giây (fps) nhưng tốc độ khung hình hiện tại ứng dụng game của bạn lại là 60 fps, bạn muốn

giảm tốc độ cập nhật của hoạt cảnh xuống nhằm tránh khả năng nó sẽ chạy hai lần. Trong trường hợp này, bạn sẽ sử dụng một timer để quản lý quá trình cập nhật của hoạt cảnh chậm hơn một nửa so với bình thường, kết quả bạn sẽ có tốc độ cập nhật là 30 fps.

Định thời gian trong Windows

Bạn có thể sử dụng hai hàm trong Windows hỗ trợ để xác định và quản lý chính xác thời gian trong ứng dụng: GetTickCount và hàm QueryPerformanceCounter.

Hàm GetTickCount, sử dụng bộ định giờ của hệ thống nên có đôi chút giới hạn về khả năng ứng dụng trong các ứng dụng game. Hàm này sẽ trả về số milli giây (milliseconds) đã qua kể từ thời điểm hệ thống bắt đầu được khởi động. Giới hạn của hàm này là nó chỉ được cập nhật sau mỗi 10 milli giây. Bởi vì sự giới hạn của hàm GetTickCount này, việc sử dụng một bộ định giờ chính xác hơn là cần thiết. Hàm QueryPerformanceCounter có thể là sự lựa chọn cho bạn.

Hàm QueryPerformanceCounter sử giải pháp hiệu quả hơn hàm GetTickCount. Hàm này được sử dụng trực tiếp bộ đếm thời gian của phần cứng thay cho giải pháp phần mềm hệ thống của hàm GetTickCount, nó cho phép bạn có thể định giời gian theo microseconds (micro giây - 10^{-6} giây). Nó rất là hữu dụng trong các ứng dụng game – rất cần những hàm quản lý thời gian thật chính xác để hiển thị các hoạt cảnh một cách chính xác nhất.

Sử dụng hàm QueryPerformanceCounter

Mẫu hàm của QueryPerformanceCounter có dạng dưới đây:

```
BOOL QueryPerformanceCounter(
                LARGE_INTEGER *lpPerformanceCount
);
```

Hàm trên yêu cầu duy nhất một tham số đầu vào: đó là con trỏ kiểu LARGE_INTEGER. Sau khi hàm này được thực hiện xong, tham số đầu vào lpPerformanceCount sẽ chứa giá trị trả về từ bộ đếm thời gian của phần cứng hệ thống.

Tiếp theo sẽ là một đoạn mã nhỏ ví dụ sử dụng hàm QueryPerformanceCount này.

```
LARGE_INTEGER timeStart;
QueryPerformanceCounter(&timeStart);
```

Ở ví dụ trên, biến timeStart sẽ lưu giá trị trả về từ hàm QueryPerformanceCount để xác định mốc thời điểm bắt đầu.

Lấy về giá trị thời gian tại các thời điểm hiển thị khung hình

Để thể hiện chính xác chuyển động của các sprite, bạn cần phải gọi tới hàm QueryPerformanceCount hai lần trong mỗi vòng lặp: một lần trước khi bạn bắt đầu vẽ và lần thứ hai là sau khi quá trình vẽ hoàn tất. Giá trị trả về của cả hai trường hợp đều là kết quả trả về của bộ đếm thời gian của hệ thống tại thời điểm hàm được gọi. Bởi vì khả năng phân biệt của bộ đếm thời gian ở mức micro giây nên chắc chắn giá trị trả về của 2 lần gọi này sẽ khác nhau. Từ đó bạn có thể xác định được giá trị chênh lệnh giữa hai lần gọi và sử dụng chúng làm thước đo trong quá trình hiển thị các khung hình tiếp theo.

Ví dụ, bạn có thể sử dụng đoạn mã nguồn minh hoạ dưới đây:

```
LARGE_INTEGER timeStart;
```

```
LARGE_INTEGER timeEnd;
QueryPerformanceCounter(&timeStart);
Render( );
QueryPerformanceCounter(&timeEnd);
LARGE_INTEGER numCounts = ( timeEnd.QuadPart – timeStart.QuadPart )
```

Sau khi các đoạn mã trên đã được thực hiện xong, biến numCounts sẽ chứa giá trị số xung nhịp của bộ đếm thời gian đã diễn ra giữa hai lần gọi tới hàm QueryPerformanceCounter. Biến QuadPart được khai báo với kiểu LARGE_INTEGER tương đương 64bit dữ liệu trên bộ nhớ và được dùng để nhận giá trị trả về của bộ đếm thời gian hệ thống.

Sau khi bạn đã có giá trị chênh lệch khoảng thời gian giữa hai lần gọi, bạn sẽ cần thực hiện một bước nữa trước khi bạn có được một giá trị hữu dụng trong quá trình hiển thị ảnh động của sprite. Đó là bạn cần phải chia giá trị numCounts này cho tần số hoạt động của bộ đếm thời gian.

Chú ý:

Tần số hoạt động của bộ đếm là giá trị đại diện cho số xung nhịp mà đồng hồ thực hiện trong một giây.

Hàm QueryPerformanceFrequency dùng để lấy về giá trị tần số của bộ đếm thời gian này của hệ thống.

Hàm QueryPerformanceFrequency này chỉ yêu cầu duy nhất một đối số: con trỏ đối tượng có kiểu LARGE_INTEGER để lưu giữ kết quả trả về của hàm. Mã nguồn minh hoạ quá trình gọi hàm này được liệt kê dưới đây:

```
LARGE_INTEGER timerFrequency;
QueryPerformanceFrequency(&timerFrequency);
```

Sau khi bạn có giá trị tần số hoạt động của bộ đếm, bạn có thể sử dụng kết hợp với giá trị của biến numCounts để tính toán tỷ lệ thời gian của quá trình di chuyển cũng như hiển thị ảnh động của sprite. Đoạn mã sau minh hoạ quá trình tính toán:

```
float anim_rate = numCounts / timerFrequency.QuadPart;
```

Bây giờ thì chúng ta đã có giá trị tỷ lệ cần thiết để thể hiện các hình ảnh động một cách mượt mà hơn.

Thay đổi cấu trúc dữ liệu của các Animation

Trong phần này chúng ta sẽ ứng dụng những kiến thức đã học ở trên để thay đổi lại mã nguồn ví dụ 4 có sử dụng kỹ thuật hiển thị hình động trên bộ định thời gian hệ thống.

Bước đâu tiên chúng ta cần thực hiện đó là thay đổi lại cấu trúc dữ liệu của Animation. Ở trong phần trước chúng ta đã khai báo các biến moveX và moveY là các biến kiểu nguyên. Chúng ta sẽ phải thay đổi kiểu dữ liệu này sang kiểu thực float để các hình ảnh của sprite sẽ được di chuyển chính xác hơn. Dưới đây là cấu trúc của sprite đã được cập nhật lại:

```
struct {
        RECT srcRect; // holds the location of this sprite
        // in the source bitmap
        float posX; // the sprite's X position
        float posY; // the sprite's Y position
```

```
        // movement
        float moveX;
        float moveY;
        // animation
        int numFrames; // the number of frames this animation has
        int curFrame; // the current frame of animation
} spriteStruct[MAX_SPRITES];
```

Như bạn có thể thấy, các biến posX và posY cũng đã được chuyển sang kiểu thực float để giúp quá trình thể hiện chúng được chính xác hơn.

Tiếp đến, bạn cần cập nhật lại giá trị đã được sử dụng trong hàm initSprite cho biến moveX. Biến moveX này trước đó đã được xác lập giá trị là 1, nhưng bạn phải phải thay đổi nó thành một giá trị mới tương ứng với giá trị tỷ lệ mà ta đã tính toán ở trên. Giá trị mới này là số lượng pixel mà bạn muốn sprite di chuyển trong một giây thể hiện các khung hình. Trong trường hợp này, chúng ta hãy xác lập nó là 30.0. Điều này cho phép những chú ca có thể bơi lội dọc theo màn hình với một tốc độ hợp lý.

Đoạn mã nguồn cuối cùng bạn cần phải thay đổi nằm bên trong hàm drawSprite. Trong hàm này, bạn sẽ thấy đoạn mã tương tự như sau:

```
spriteStruct[whichOne].posX += spriteStruct[whichOne].moveX;
```

Dòng lệnh này điều khiển quá trình di chuyển của mỗi sprite trên màn hình. Bạn sẽ thấy rằng toạ độ X – biến posX – sẽ được tăng lên bằng cách cộng với giá trị biến moveX lưu trữ. Để quá trình hiển thị ảnh được chính xác sau khi ta tiến hành cập nhật mã nguồn hỗ trợ bộ định thời tốc độ hiển thị, bạn cần phải thay đổi dòng mã nguồn trên về dạng như sau:

```
spriteStruct[whichOne].posX += spriteStruct[whichOne].moveX * anim_rate;
```

Trong dòng lệnh này, giá trị moveX được nhân với giá trị lưu trong biến anim_rate. Bởi vì biến anim_rate này được cập nhật mỗi lần một khung hình được hiển thị, nó sẽ cung cấp một hình sprite chuyển động rất mượt mà ngay cả trên một máy tính tốc độ cao.
Bây giờ thì bạn đã cập nhật xong mã nguồn cho sprite, tiếp đến bạn sẽ phải chèn thêm các mã lệnh của bộ định thời timer. Bộ định thời timer này yêu cầu ba biến toàn cục:

```
LARGE_INTEGER timeStart; // holds the starting count
LARGE_INTEGER timeEnd; // holds the ending count
LARGE_INTEGER timerFreq; // holds the frequency of the counter
```

Tiếp đến chúng ta sẽ thực hiện lời gọi tới hàm QueryPerformanceFrequence để lấy về tần số hoạt động của bộ đếm thời gian. Bạn cần phải gọi tới hàm này trước khi vòng lặp chính quản lý thông điệp được thực hiện:

```
QueryPerformanceFrequency(&timerFreq);
```

Cuối cùng, bạn cần phải thêm vào các lời gọi tới hàm QueryPerformanceCounter trước và sau khi gọi tới hàm render. Trước tiên là trước khi gọi hàm render:

```
QueryPerformanceCounter(&timeStart);
```

Lần gọi thứ hai phải được đặt sau lời gọi tới hàm render:

```
QueryPerformanceCounter(&timeEnd);
```

Ngay sau khi lời gọi cuối cùng tới hàm QueryPerformanceCounter, bạn cần phải tính toán lại ngay giá trị tốc độ cập nhật ảnh động của sprite.

```
anim_rate = ( (float)timeEnd.QuadPart - (float)timeStart.QuadPart ) /
            timerFreq.QuadPart;
```

Bạn có thể biên dịch ví dụ đã được chỉnh sửa và xem hết quả của những gì chúng ta vừa cập nhật, các hình động đã được thể hiện mượt mà hơn. Toàn bộ mã nguồn cập nhật này bạn có thể tìm thấy trong như mục chapter3\example5 trên CD-ROM.

Tổng kết chương

Tại thời điểm này, bạn đã được tiếp cận tới những kiến thức đơn giản về cách hoạt động của DirectX và làm thế nào để tạo và sử dụng các surface.

Bạn cũng đã được tiếp cận tới kiến thức về bộ định thời timer và làm thế nào để tạo một đối tượng sprite động chuyển động thật mượt mà. Bạn sẽ tiếp tục còn sử dụng những kiến thức về bộ định thời timer này trong suốt các phần tiếp theo của quyển sách, chính vì vậy bạn phải thực sự nắm chắc những kiến thức này.

Trong chương tiếp theo, bạn sẽ thực sự được tiếp cận tới thế giới đồ hoạ 3 chiều.

Những kiến thức đã học trong chương này

Trong chương này chúng ta đã đề cập tới các vấn đề sau đây:

- Làm thế nào để tải một ảnh bitmap thông qua các hàm trong thư viện D3DX
- Làm thế nào để hiển thị một ảnh lên màn hình bằng DirectX
- Sprite là gì và làm thế nào để sử dụng chúng
- Làm thế nào để tạo một sprite động bằng cách sử dụng kỹ thuật hiển thị các khung hình của sprite theo một bộ định thời timer.

Câu hỏi kiểm tra kiến thức

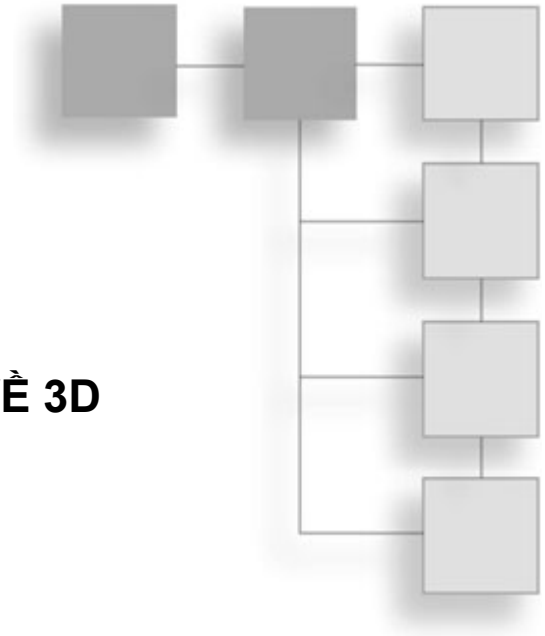
Bạn có thể tìm thấy câu trả lời của phần Kiểm tra kiến thức và Những bài tập tự làm trong phần Phụ lục A, “Trả lời các câu hỏi và bài tập” ở cuối quyển sách.

1. Hàm nào dùng để tạo đối tượng offscreen surface?
2. Chức năng của hàm StretchRect?
3. Các kiểu dữ liệu có thể lưu trữ trong offscreen surface?
4. Tại sao chúng ta nên xoá sạch bộ đệm sau mỗi lần hiển thị?
5. Sự khác biệt chủ yếu giữa hàm QueryPerformanceCounter và hàm GetTickCount?

Bài tập tự làm

1. Viết một ví dụ nhỏ sử dụng hàm StretchRect để thu nhỏ các phần của một bức ảnh.
2. Viết một chương trình sử dụng các kiến thức đã học trong chương này để di chuyển một thông điệp dọc theo màn hình thông qua các sprites.

CHƯƠNG 4



NHỮNG KIẾN THỨC CƠ BẢN VỀ 3D

Chắc các bạn cũng thấy game 2D đang dần bị tụt hậu trong một vài năm gần đây. Đa số các game bây giờ đều cố sử dụng được sức mạnh của các loại card 3D mới nhất, cố gắng làm cho game thật hơn. Direct3D là một thành phần quan trọng trong trào lưu này. Nó cho phép hàng triệu khách hàng của Microsoft Windows được thưởng thức những công nghệ game mới nhất.

Những gì bạn sẽ được học ở chương này:

- Không gian 3D được sử dụng thế nào.
- Hệ thống toạ độ là gì.
- Cách dựng những điểm của một đa giác.
- Khái niệm vecto trong Direct3D.
- Vertex buffer là gì.
- Khái niệm khung cảnh 3D (3D scene) .
- Những cấu trúc cơ bản bạn có thể sử dụng.

Không gian 3D

Phần trước, tôi đã nói về những game chỉ cho phép di chuyển theo 2 phương, tức là trong không gian phẳng. Khái niệm (sprites) mà bạn dùng ở trên chủ yếu là cho không gian với chiều rộng và chiều cao nhưng không có chiều sâu.

DIrect3D cho bạn khả năng đưa thêm một chiều không gian nữa vào thế giới game với sự bổ sung của chiều sâu. Chiều sâu là khả năng của vật thể có thể di chuyển ra xa hoặc lại gần người quan sát. Nhân vật ở trong thế giới 3D sẽ thật hơn nhiều bản sao của chúng trong không gian 2D.

Không gian 3D cho phép nhân vật di chuyển vòng quanh theo cách tương tự như thế giới thực. Trước khi bạn tận dụng được lợi thế của không gian 3D, bạn cần biết cách xây dựng nó, và cách đặt các vật thể vào đó.

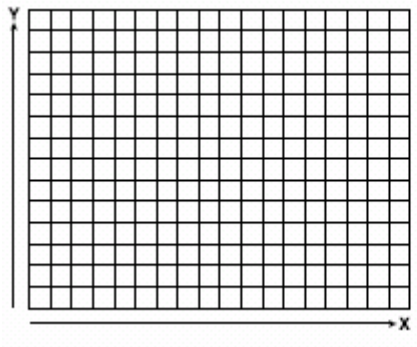
Hệ thống toạ độ

Hệ thống toạ độ là cách để định nghĩa điểm trong không gian. Nó bao gồm các đường thẳng vuông góc với nhau gọi là các trục toạ độ. Hệ toạ độ 2D chỉ gồm 2 trục toạ độ, còn

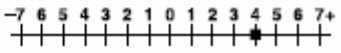
hệ 3D thì có thêm một trục nữa. Tâm điểm của hệ toạ độ, nơi mà các trục toạ độ giao nhau, được gọi là gốc toạ độ. Hình 4.1 biểu hiện hệ trục toạ độ 2D. Hai trục của hệ toạ độ 2D được kí hiệu là X và Y. X là trục nằm ngang, còn Y là trục thẳng đứng.

Xác định một điểm trong không gian 2D

Một điểm được xác định như một vị trí duy nhất trên một trục. Một điểm ở trong không gian 1D, (chỉ có duy nhất một trục), có thể được biểu diễn qua một giá trị. Hình 4.2 biểu diễn điểm trong không gian 1D. Gốc của đường thẳng có giá trị là 0. Hướng sang bên phải của gốc toạ độ là các giá trị dương, ngược lại, ở bên trái gốc toạ độ là các giá trị âm. Trong hình 4.2, điểm biểu diễn có giá trị là dương 4.



Hình 4.1

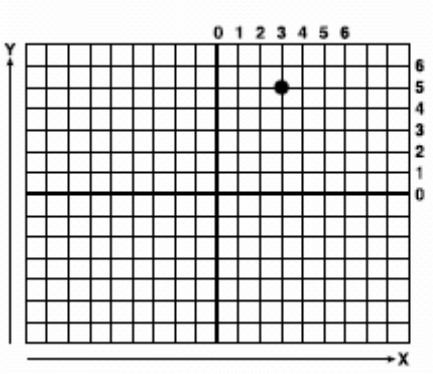


Hình 4.2

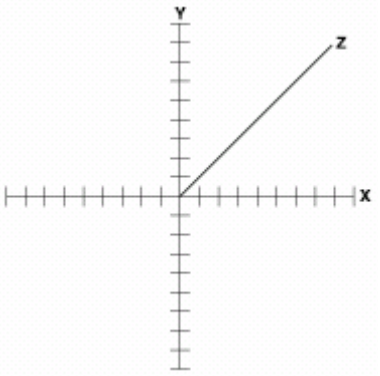
Hệ toạ độ 2D, vì nó có 2 trục toạ độ, nên đòi hỏi thêm một giá trị nữa để biểu diễn một điểm. Để biểu diễn một điểm trong không gian 2D, bạn cần xác định vị trí dọc theo trục X và Y của nó. Ví dụ, một điểm trong hệ toạ độ 2D có thể được xác định bằng 2 số là X và Y, mỗi số xác định một vị trí trên trục tương ứng. Giống như ví dụ 1D ở hình 4.2, những giá trị trên trục X tăng dần từ trái qua phải, nhưng những giá trị trên trục Y lại tăng dần từ dưới lên trên. Hình 4.3 cho thấy hệ toạ độ 2D với một điểm có toạ độ X=3 và Y=5, người ta thường viết dưới dạng (X, Y). Trong ví dụ này điểm đó được biểu diễn là (3, 5).

Xác định 1 điểm trong không gian 3D

Như đã đề cập ở phần trên, hệ toạ độ 3D có thêm một trục nữa, gọi là trục Z. Trục Z vuông góc với mặt phẳng tạo bởi trục X và Y. Hình 4.4 cho ta thấy vị trí của trục Z.
Chú ý rằng trong hệ trục toạ độ này, trục X và Y để thể hiện chiều rộng và chiều cao, còn trục Z thể hiện chiều sâu. Trục Z có cả giá trị âm và dương khi ta di chuyển so với gôc toạ độ tuỳ thuộc vào loại hệ toạ độ. Hệ toạ độ thường được sắp đặt theo cả kiểu tay trái lẫn kiểu tay phải.



Hình 4.3



Hình 4.4

Hệ toạ độ tay trái

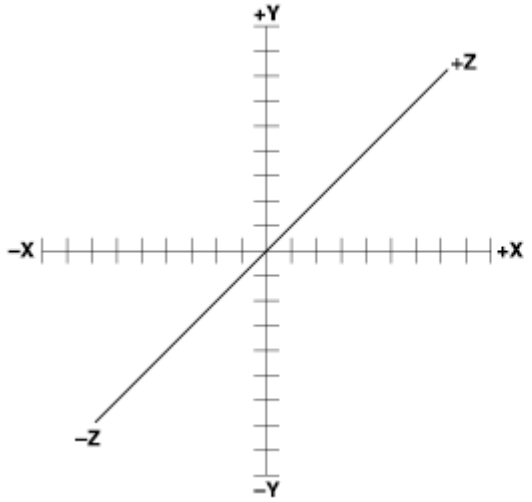
Hệ toạ độ tay trái: chiều dương trục X hướng về bên phải và chiều dương trục Y hướng lên trên. Sự khác nhau chủ yếu là ở trục Z. Trục Z trong hệ toạ độ này có chiều dương hướng ra xa người nhìn, và chiều âm hướng về phía người nhìn. Hình 4.5 biểu diễn hệ toạ độ tay trái. Đây là hệ toạ độ được sử dụng trong Direct3D.

Hệ toạ độ tay phải

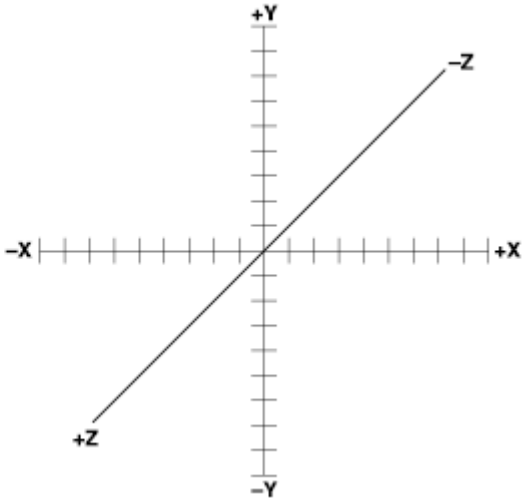
Hệ tọa độ tay phải được dùng trong OpenGL, có trục X và trục Y giống như hệ tọa độ tay trái, nhưng trục Z thì theo chiều ngược lại. Chiều dương của trục Z hướng về phía người nhìn, trong khi chiều âm thì đi ra xa. Hình 4.6 biểu diễn hệ tọa độ tay trái.

Khái niệm về vector

Một vecto tương tự như là một điểm. Vecto bao gồm các thông tin về tọa độ X, Y, Z và đồng thời cũng chứa đựng những thông tin khác nữa, ví dụ như là màu sắc hoặc texture.



Hình 4.5: hệ tọa độ tay trái



Hình 4.6: hệ tọa độ tay phải

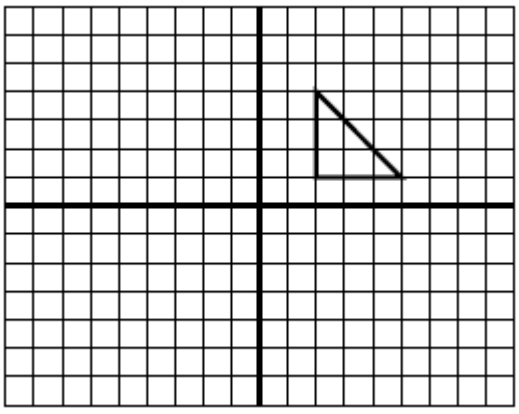
Cấu trúc để mô tả vecto:

```
struct {
        float x;
        float y;
        float z;
} vertex;
```

Cấu trúc vecto này gồm 3 biến kiểu float, miêu tả vị trí của vecto so với các trục tọa độ.

Tạo một hình

Bạn có thể tạo một hình nào đó bằng cách dùng 2 hoặc nhiều vecto. Ví dụ, để tạo một hình tam giác ta cần có ba vecto để xác định ba đỉnh của tam giác. Sử dụng vecto để thể hiện một hình giống như việc ta nối các điểm lại với nhau. Hình 4.7 cho thấy cách tạo ra một hình tam giác bằng ba vecto.



Hình 4.7 Tạo tam giác bằng 3 vector

Để tạo một hình tam giác cần có 3 vecto

```
struct {
        float x; // toạ độ X
        float y; // toạ độ Y
        float z; // toạ độ Z
} vertex [ 3 ];
```

Ở đây, tôi vừa khai báo một mảng gồm 3 vecto. Bước tiếp theo là xác định vị trí cho các vecto theo như hình 4.7.

```
// vecto thứ nhất
vertex[0].x = 2.0;          // gán tọa độ X
vertex[0].y = 4.0;          // gán tọa độ Y
vertex[0].z = 0.0;          // gán tọa độ Z
// vecto thứ hai
vertex[1].x = 5.0;          // gán tọa độ X
vertex[1].y = 1.0;          // gán tọa độ Y
vertex[1].z = 0.0;          // gán tọa độ Z
// vecto thứ ba
vertex[0].x = 2.0;          // gán tọa độ X
vertex[0].y = 1.0;          // gán tọa độ Y
vertex[0].z = 0.0;          // gán tọa độ Z
```

Chú ý là tọa độ Z của cả ba vecto đều được gán là 0. Do tam giác này không có chiều sâu, nên tọa độ Z giữ nguyên giá trị 0.

Chú ý:

Tam giác là hình khép kín đơn giản nhất khi dùng vecto để biểu diễn. Bạn có thể tạo được những hình phức tạp hơn như hình vuông, hình cầu… nhưng thực ra chúng cũng được chia nhỏ ra thành các hình tam giác trước khi vẽ.

Cho thêm màu sắc

Ở trên, cấu trúc vecto chỉ gồm thông tin liên quan đến vị trí của vecto. Tuy nhiên, vecto cũng có thể chứa thông tin về màu sắc. Thông tin về màu sắc này có thể chứa trong bốn biến được thêm vào là R, G, B và A.

+ R là thành phần đỏ của màu.
+ G là thành phần xanh lá cây của màu.
+ B là thành phần xanh nước biển của màu.
+ A hệ số alpha của màu.

Mỗi một giá trị trên giúp ta xác định màu của vecto. Cấu trúc vecto lúc này được bổ sung như sau:

```
struct {
        // thông tin về vị trí
        float x;
        float y;
        float z;
        // thông tin về màu sắc
        float R;
        float G;
        float B;
        float A;
} vertex;
```

Sử dụng các biến R, G, B và A, bạn có thể đặt màu cho vecto. Ví dụ, nếu bạn muốn vecto có màu trắng, thì các biến R, G và B đều được đặt là 1.0. Đặt màu vecto bằng màu nước biển thì R và G được gán là 0.0 trong khi B gán là 1.0.

Chú ý:

Thành phần alpha của màu quyết định độ trong suốt của nó. Nếu giá trị alpha là 0, thì màu được xác định bằng R, G và B sẽ là màu đặc. Nếu alpha lớn hơn 0, thì màu lúc này sẽ ở một mức độ trong nào đó. Giá trị của alpha là từ 0.0f đến 1.0f.

Vertex Buffers

Vertex buffers là những vùng nhớ chứa thông tin về vecto cần thiết để tạo ra các đối tượng 3D. Những vecto chứa trong buffer có thể chứa đựng nhiều dạng thông tin khác nhau, như thông tin về vị trí, hệ texture, màu sắc. Vertex buffers rất hữu dụng cho lưu trữ hình tĩnh (những thứ cần render lặp lại nhiều lần). Vertex buffers có thể tồn tại cả trong bộ nhớ hệ thống và trong bộ nhớ của thiết bị đồ họa.
Để tạo một vertex buffer ta cần khai báo một biến có cấu trúc IDirect3DVertexBuffer9. Nó chứa trỏ trỏ tới vertex buffer do DirectX tạo ra.
Bước tiếp theo, ứng dụng cần tạo một vertex buffer và lưu trữ nó ở trong biến vừa khai báo. Sau khi tạo thành công vertex buffer, ta có thể lưu dữ liệu vecto vào đó. Ta thực hiện điều đó bằng cách khóa vertex buffer và copy dữ liệu vecto vào đó.

Tạo một vertex buffer

Bạn có thể tạo vertex buffer thông qua lời gọi hàm CreateVertexBuffer. Hàm này, gồm sáu đối số, được định nghĩa như sau:

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

■ **Length**. Biến xác định chiều dài của vertex buffer tính theo byte.
■ **Usage**. Cờ quy định cách thể hiện của vertex buffer. Giá trị này thường gán là 0.
■ **FVF**. Định dạng mềm dẻo mà vertex buffer sử dụng.
■ **Pool**. Vùng nhớ chứa vertex buffer. Giá trị này có kiểu D3DPOOL.
■ **ppVertexBuffer**. Con trỏ có cấu trúc IDirect3DVertexBuffer9 trỏ tới vertex buffer vừa tạo ra.
■ **pHandle**. Giá trị này nên đặt là NULL.
Những vecto lưu trong 1 vertex buffer có cấu trúc rất mềm dẻo. Về cơ bản, điều này có nghĩa là những vecto chứa trong buffer có thể chỉ chứa thông tin về vị trí, hoặc có thể chứa cả thông tin về màu sắc hay texture. Kiểu dữ liệu của vecto được điều khiển thông qua cờ định dạng mềm dẻo của vecto (FVF - Flexible Vertex Format).

Định dạng mềm dẻo của vecto

Định dạng mềm dẻo của vecto cho phép sự tùy biến về thông tin chứa trong vertex buffer. Bằng cách sử dụng cờ FVF, ta có thể thay đổi buffer để chứa bất kì dạng vecto nào. Bảng 4.1 mô tả chi tiết về cờ FVF.

D3DFVF_XYZ	Định dạng gồm X, Y, Z của vecto chưa qua biến đổi.
D3DFVF_XYZRHW	Định dạng gồm X, Y, Z của vecto đã qua biến đổi.
D3DFVF_XYZW	Định dạng chứa dữ liệu vecto đã qua biến đổi, cắt xén.
D3DFVF_NORMAL	Định dạng chứa dữ liệu thông thường.

D3DFVF_PSIZE	Định dạng bao gồm cả kích thước điểm của vecto.
D3DFVF_DIFFUSE	Bao gồm cả màu có hướng (xem chương sau).
D3DFVF_SPECULAR	Bao gồm cả màu vô hướng (xem chương sau).
D3DFVF_TEX0	Texture 0
D3DFVF_TEX1	Texture 1
D3DFVF_TEX2	Texture 2
D3DFVF_TEX3	Texture 3
D3DFVF_TEX4	Texture 4
D3DFVF_TEX5	Texture 5
D3DFVF_TEX6	Texture 6
D3DFVF_TEX7	Texture 7
D3DFVF_TEX8	Texture 8

Direct3D có thể sử dụng được tới 8 loại texture khác nhau cho mỗi vecto.
Định dạng vecto ta sẽ dùng được tạo ra bằng cách định nghĩa một cấu trúc vecto bổ sung. Cấu trúc vecto sau đây định nghĩa một vecto chứa thông tin về vị trí chưa qua biến đổi và màu của vecto.

```
struct CUSTOMVERTEX
{
        FLOAT x, y, z, rhw; // vị trí 3D chưa qua biến đổi
        DWORD color; // màu của vecto
};
```

Cấu trúc CUSTOMVERTEX bao gồm tọa độ chuẩn X, Y và Z của vecto, đồng thời có cả thành phần RHW. Giá trị RHW tượng trưng cho (Reciprocal of Homogeneous W), thông báo cho Direct3D rằng những vecto đang được dùng nằm trong vùng thấy được của màn hình. Giá trị này thường được dùng tính toán về làm mờ và xén tỉa và nên gán giá trị là 1.0.

Chú ý:

Màu của vecto là giá trị kiểu DWORD. Direct3D cung cấp một vài lệnh hỗ trợ bạn trong việc tạo màu. Một trong những lệnh đó là **D3DCOLOR_ARGB**(a, r, g, b). Lệnh này có bốn đối số alpha, red, green, blue. Mỗi thành phần có giá trị nằm trong đoạn từ 0 đến 255. Lệnh này trả về một giá trị màu DWORD mà Direct3D có thể sử dụng.
D3DCOLOR_ARGB(0, 255, 0, 0) tạo ra màu đỏ.
Một số lệnh khác là **D3DCOLOR_RGBA** và **D3DCOLOR_XRGB**, đã được trình bày chi tiết trong tài liệu của DirectX.

Sau khi tạo được cấu trúc vecto, bước tiếp theo là quy định cờ FVF làm tham số cho hàm CreateVertexBuffer.
Bởi vì cấu trúc CUSTOMVERTEX đòi hỏi thông tin vị trí chưa qua biến đổi và thành phần về màu, nên cờ FVF cần dùng là D3DFVF_XYZRHW và D3DFVF_DIFFUSE.
Code ví dụ dưới đây thể hiện lời gọi hàm CreateVertexBuffer sử dụng cấu trúc trên:

```
// cấu trúc vecto bổ sung
struct CUSTOMVERTEX
{
        FLOAT x, y, z, rhw; // vị trí 3D chưa qua biến đổi của vecto
        DWORD color; // màu của vecto
};
// biến trỏ tới vertex buffer
LPDIRECT3DVERTEXBUFFER9 buffer = NULL;
// biến lưu giá trị trả về của hàm
HRESULT hr;
// tạo một vertex buffer
```

```
hr = pd3dDevice->CreateVertexBuffer(
                                3*sizeof( CUSTOMVERTEX ),
                                0,
                                D3DFVF_XYZRHW | D3DFVF_DIFFUSE,
                                D3DPOOL_DEFAULT,
                                &buffer,
                                NULL );
// Kiểm tra giá trị trả về
if FAILED ( hr)
        return false;
```

Như bạn thấy, cấu trúc CUSTOMVERTEX được tạo ra trước, thông báo cho Direct3D kiểu của vecto được dùng. Tiếp theo, lời gọi tới CreateVertexBuffer tạo ra một buffer và lưu trữ vào biến "buffer" khai báo ở trên.
Đối số đầu tiên cho CreateVertexBuffer, kích thước của buffer theo byte, tạo ra vùng nhớ đủ để chứa ba vecto kiểu CUSTOMVERTEX.
Đối số thứ ba, cờ FVF, quy định cờ D3DFVF_XYZRHW và D3DFVF_DIFFUSE sẽ được dùng.
Đối số thứ tư đặt vùng nhớ cho vertex buffer, Giá trị D3DPOOL_DEFAULT được dùng để tạo ra buffer có vùng nhớ thích hợp nhất với kiểu này.
Đối số cuối cùng là cái mà bạn cần quan tâm. Nó giúp ta tham chiếu tới buffer vừa được tạo ra.
Sau khi lời gọi tới CreateVertexBuffer hoàn thành, ta cần kiểm tra giá trị trả về để xác nhận rằng buffer đã được tạo ra thành công.

Nạp dữ liệu cho buffer

Sau khi bạn có vertex buffer, bạn cần đưa dữ liều vecto vào đó. Nhưng trước đó, bạn phải khóa vùng nhớ mà buffer đang dùng. Sau khi vùng nhớ này được khóa, bạn mới có thể nạp dữ liệu vào đó.

Khóa Vertex Buffer

Khóa vùng nhớ cho vertex buffer cho phép ứng dụng của bạn ghi dữ liệu lên đó. Tại thời điểm này, bạn đã định nghĩa xong vertex buffer và kiểu của vecto mà nó chứa. Bước tiếp theo là khóa buffer và nạp dữ liệu vecto. Khóa buffer thông qua lời gọi hàm:

```
HRESULT Lock(
        UINT OffsetToLock,
        UINT SizeToLock,
        VOID **ppbData,
        DWORD Flags
);
```

Hàm Lock function có bốn đối số:

- **OffsetToLock**. Vùng buffer bạn muốn khóa. Nếu bạn muốn khóa toàn bộ thì gán giá trị này là 0.
- **SizeToLock**. Kích thước dạng byte bạn muốn khóa. Nếu bạn muốn khóa toàn bộ buffer thì gán giá trị này là 0.
- **ppbData**. Con trỏ dạng void trỏ tới buffer chứa vecto.
- **Flags**. Cờ quy định kiểu khóa. Đưa vào một trong các giá trị sau:
  - D3DLOCK_DISCARD. Ghi đè toàn bộ buffer.
  - D3DLOCK_NO_DIRTY_UPDATE. Không ghi dữ liệu lên các vùng bẩn!!!
  - D3DLOCK_NO_SYSLOCK. Cho phép hệ thống tiếp tục xử lý các sự kiện trong suốt quá trình khóa.

- D3DLOCK_READONLY. Chỉ cho phép đọc.
- D3DLOCK_NOOVERWRITE. Không cho ghi đè dữ liệu cũ.

Đoạn code sau cho thấy cách gọi thông thường của hàm Lock:

```
HRESULT hr;
VOID* pVertices;
// Khóa vertex buffer
hr = g_pVB->Lock( 0, 0, ( void** ) &pVertices, 0 );
// Kiểm tra giá trị trả về
if FAILED (hr)
        return false;
```

Hàm Lock thừa nhận rằng bạn đã tạo thành công vertex buffer. Biến g_pVB trỏ tới buffer này.

Sao chép dữ liệu vào vertex buffer

Sau khi khóa vertex buffer, bạn có thể tự do copy dữ liệu vào buffer. Bạn vừa có thể copy các vecto mới vào buffer, vừa có thể sửa đổi những vecto đã nằm trong buffer. Ví dụ tiếp theo cho thấy cách sử dụng memcpy để copy một mảng vecto vào trong vertex buffer.

```
// cấu trúc CUSTOMVERTEX
struct CUSTOMVERTEX
{
        FLOAT x, y, z, rhw; // vị trí 3D đã qua biến đổi của vecto
        DWORD color; // màu vecto
};
// định nghĩa các vecto dùng trong vertex buffer
CUSTOMVERTEX g_Vertices [ ] =
        {
        {320.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 255, 0, 0),},
        {250.0f, 400.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 0, 255, 0),},
        {50.0f, 400.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 0, 0, 255),},
        };
// Copy dữ liệu vào vertex buffer
memcpy( pVertices, g_Vertices, sizeof( g_Vertices ) );
```

Đầu tiên ta khai báo cấu trúc CUSTOMVERTEX. Như đã đề cập trước đây, cấu trúc này chứa cả vị trí và màu của vecto. Tiếp theo, ta tạo ra một mảng vecto. Nó được trỏ đến bởi g_Vertices và nó chứa những vecto sẽ được copy vào buffer. Cuối cùng, lời gọi tới memcpy sẽ copy những vecto này vào buffer. Đối số thứ nhất cho memcpy là pVertices, là con trỏ kiểu void đã được tạo ra qua lời gọi Lock.

Mở khóa Vertex Buffer

Sau khi những vecto trên đã đươc copy vào buffer, bạn phải mở khóa buffer. Mở khóa buffer cho phép Direct3D tiếp tục quá trình bình thường. Bạn mở khóa buffer thông qua hàm Unlock được định nghía dưới đây:

```
HRESULT Unlock (VOID);
```

Hàm Unlock không đòi hỏi đối số và giá trị trả về của nó là D3D_OK nếu thành công. Sau khi nạp dữ liệu vào vertex buffer, ta có thể biểu diễn nó trên màn hình.
Hàm SetupVB dưới đây tổng hợp toàn bộ các bước đã nêu ở trên:

```
// con trỏ tới vertex buffer
LPDIRECT3DVERTEXBUFFER9 g_pVB = NULL;
/*****************************************************************************
* SetupVB
* Tạo và nạp dữ liệu vertex buffer
```

```
*****************************************************************************/
HRESULT SetupVB()
{
	HRESULT hr;
	// Khởi tạo giá trị cho 3 vecto của tam giác
	CUSTOMVERTEX g_Vertices[] =
		{
		{320.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 255, 0, 0), },
		{250.0f, 400.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 0, 255, 0), },
		{50.0f, 400.0f, 0.5f, 1.0f, D3DCOLOR_ARGB (0, 0, 0, 255), },
		};
	// tạo một vertex buffer
	hr = pd3dDevice->CreateVertexBuffer(
			3*sizeof(CUSTOMVERTEX),
			0,
			D3DFVF_XYZRHW|D3DFVF_DIFFUSE,
			D3DPOOL_DEFAULT,
			&g_pVB,
			NULL );
	// Kiểm tra xem vertex buffer đã được tạo ra chưa
	if FAILED ( hr )
		return NULL;
	VOID* pVertices;
	// Khóa vertex buffer
	hr = g_pVB->Lock( 0, sizeof(g_Vertices), (void**)&pVertices, 0 );
	// Kiểm tra xem lời gọi hàm có thành công không
	if FAILED (hr)
		return E_FAIL;
	// Copy các vecto vào buffer
	memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
	// Mở khóa vertex buffer
	g_pVB->Unlock();
	return S_OK;
}
```

Hàm SetupVB đòi hỏi biến chứa vertex buffer phải được khai báo bên ngoài phạm vi của hàm này. Biến g_pVB tham chiếu tới biến này. Nếu vertex buffer được tạo và nạp dữ liệu thành công, hàm SetupVB trả về một giá trị kiểu HRESULT là S_OK.

Hiển thị nội dung Buffer

Sau khi bỏ thời gian để tạo vertex buffer và nạp dữ liệu vecto cho nó, chắc bạn sẽ tự hỏi nó sẽ xuất hiện trên màn hình thế nào. Để render những vecto trong vertex buffer đòi hỏi ba bước. Bước thứ nhất là cài đặt luồng nguồn, tiếp theo là thiết lập cho chế độ shader (đổ bóng), cuối cùng là vẽ những vecto đó lên màn hình. Những bước trên được giải thích chi tiết ở phần sau.

Cài đặt luồng nguồn

Luồng trong Direct3D là mảng của những thành phần dữ liệu có bao gồm nhiều thành tố. Vertex buffer bạn tạo ra ở trên chính là một luồng. Trước khi Direct3D có thể render một vertex buffer, bạn phải gắn liền buffer đó với một luồng dữ liệu. Điều đó được thực hiện với hàm SetStreamSource được định nghĩa dưới đây:

```
HRESULT SetStreamSource(
	UINT StreamNumber,
	IDirect3DVertexBuffer9 *pStreamData,
	UINT OffsetInBytes,
```

```
        UINT Stride
);
```

SetStreamSource cần 4 đối số:

- **StreamNumber**. Số của luồng dữ liệu. Nếu bạn chỉ tạo một vertex buffer thì đối số này là 0
- **pStreamData**. Con trỏ tới biến chứa vertex buffer.
- **OffsetInBytes**. Số byte tính từ điểm bắt đầu của buffer nới chứa dữ liệu. Giá trị này thường để là 0.
- **Stride**. Kích thước của mỗi vecto trong.

Một ví dụ về lời gọi tới SetStreamSource:

```
pd3dDevice->SetStreamSource ( 0, buffer, 0, sizeof(CUSTOMVERTEX) );
```

Trong lời gọi tới hàm SetStreamSource, đối số thứ nhất biểu diễn số của luồng được gán là 0. Đối số thứ hai là con trỏ tới vertex buffer hợp lệ. Đối số thứ ba được gán là 0, thông báo cho Direct3D vị trí bắt đầu là từ đầu luồng. Đối số cuối cùng là bước của luồng. Nó được gán cho kích thước bằng byte của cấu trúc CUSTOMVERTEX. Hàm sizeof sẽ tính toán số byte cho bạn.

Cài đặt đổ bóng

Sau khi đặt nguồn cho luồng, bạn phải đặt chế độ đổ bóng vecto. Chế độ này thông báo cho Direct3D kiểu đổ bóng được dùng. Hàm SetFVF, định nghĩa dưới đây, cài đặt cho Direct3D sử dụng định dạng vecto bổ sung.

```
HRESULT SetFVF(
        DWORD FVF
);
```

Hàm SetFVF chỉ cần một đối số là FVF. Đối số FVF gồm các giá trị định nghĩa bởi D3DFVF. Đoạn code sau cho thấy cách dùng FVF.

```
HRESULT hr;
hr = pd3dDevice->SetFVF (D3DFVF_XYZRHW | D3DFVF_DIFFUSE);
// Kiểm tra kết quả trả về từ SetFVF
if FAILED (hr)
        return false;
```

Đoạn code trên đưa vào giá trị D3DFVF_XYZRHW và D3DFVF_DIFFUSE cho FVF. Khi cấu trúc CUSTOMVERTEX được xây dựng, nó dùng hai giá trị trên khi tạo vertex buffer. Bạn cũng phải tạo một (Direct3D device) hợp lệ. Nó được tham chiếu qua biến pd3dDevice.

Render Vertex Buffer

Sau khi bạn tạo luồng và gắn nó với một vertex buffer, bạn có thể render những vecto trong đó lên màn hình. Hàm cần dùng là DrawPrimitive được định nghĩa dưới đây.
Hàm DrawPrimitive sẽ duyệt qua vertex buffer và render dữ liệu của nó lên màn hình.

```
HRESULT DrawPrimitive(
        D3DPRIMITIVETYPE PrimitiveType,
        UINT StartVertex,
        UINT PrimitiveCount
);
```

Hàm DrawPrimitive cần ba đối số:

- PrimitiveType. Kiểu gốc dùng để vẽ vecto của luồng.
- StartVertex. Vị trí của vecto đầu tiên trong.

- PrimitiveCount. Số kiểu gốc cần render.

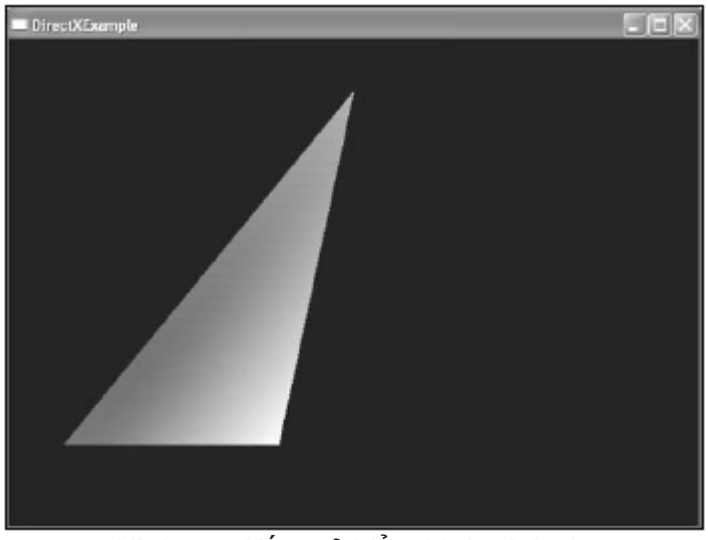
Kiểu gốc có thể là một trong các giá trị:

- D3DPT_POINTLIST. Vẽ các điểm riêng lẻ.
- D3DPT_LINELIST. Vẽ những đường thẳng riêng lẻ.
- D3DPT_LINESTRIP. Vẽ những line liên tiếp nối đuôi nhau.
- D3DPT_TRIANGLELIST. Vẽ những tam giác riêng lẻ gồm 3 vecto.
- D3DPT_TRIANGLESTRIP. Vẽ một loạt tam giác liên tiếp nối đuôi nhau (từ tam giác thứ 2 trở đi chỉ cần 1 vecto để xác định).
- D3DPT_TRIANGLEFAN. Vẽ một loạt tam giác có chung 1 vecto (đỉnh).

Đoạn code sau sử dụng DrawPrimitive ở chế độ vẽ D3DPT_TRIANGLESTRIP.

```
HRESULT hr;
// Gọi DrawPrimitive
hr = pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 1 );
// Kiểm tra kết quả trả về
if FAILED (hr)
        return false;
```

Đoạn code trên thông báo với Direct3D để render những vecto chứa trong vertex buffer sử dụng chế độ vẽ D3DPT_TRIANGLESTRIP cho tham số thứ nhất. Tham số thứ hai được gán là 0, thông báo cho DrawPrimitive bắt đầu với vecto đầu tiên trong buffer. Đối số cuối được gán là 1 bởi vì số vecto trong bộ đệm chỉ đủ để tạo ra một tam giác.
Ta cần có thiết bị một Direct3D được khai báo chuẩn. Nó được tham chiếu đến thông qua biến pd3dDevice.
Mã nguồn chi tiết cho phần tạo và render một vertex buffer có thể tìm thấy ở thư mục chapter4\example1 trên đĩa CD-ROM.



Hình 4.8: Kết quả hiển thị từ ví dụ 1.

Render một khung cảnh (scene)

Trước khi bạn có thể render, bạn phải chuẩn bị Direct3D để render. Hàm BeginScene thông báo cho Direct3D chuẩn bị cho render. Sử dụng hàm BeginScene, Direct3D cần

chắc chắn rằng vùng render hợp lệ và sẵn sàng. Nếu hàm BeginScene bị lỗi, code của bạn sẽ bỏ qua đoạn gọi render.
Sau khi render xong, bạn cần gọi tới hàm EndScene. Hàm EndScene thông báo với Direct3D rằng bạn đã kết thúc quá trình gọi render và khung cảnh (scene) đã sẵn sàng để chuyển sang back buffer.
Đoạn code sau đây xác nhận mã trả về từ khối BeginScene và EndScene.

```
HRESULT hr;
if ( SUCCEEDED( pDevice->BeginScene( ) ) )
{
        // Chỉ thực hiện render khi việc gọi BeginScene thành công
        // Đóng khung cảnh (scene)
        hr = pDevice->EndScene( );
        if ( FAILED ( hr ) )
                return hr;
}
```

Đoạn code ở trên xác nhận sư thành công của lời gọi tới BeginScene trước khi cho phép quá trình render diễn ra bằng cách sử dụng lệnh SUCCEEDED. Khi quá trình render hoàn thành, bạn gọi tới hàm EndScene.
Đoạn code sau là một ví dụ:

```
/*******************************************************************************
* render
*******************************************************************************/
void render()
{
        // Xóa back buffer bằng màu đen
        pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0,0,0), 1.0f, 0 );
        // Thông báo bắt đầu scene tới Direct3D
        pd3dDevice->BeginScene();
        // Vẽ những vecto chứa trong vertex buffer
        // Trước hết cần đặt luồng dữ liệu
        pd3dDevice->SetStreamSource( 0, buffer, 0, sizeof(CUSTOMVERTEX) );
        // Đặt định dạng vecto cho luồng
        pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
        // Vẽ những vecto trong vertex buffer dùng triangle strips
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 1 );
        // Thông báo với Direct3D rằng quá trình vẽ đã kết thúc
        pd3dDevice->EndScene();
        // chuyển từ back buffer sang front buffer
        pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Hàm render là tập hợp tất cả các bước đã nói ở trên. Biến pd3dDevice thể hiện một thiết bị Direct3D hợp lệ được tạo ra ở bên ngoài hàm này.

Những kiểu cơ bản

Ở trên, bạn phải lựa chọn cài đặt kiểu cơ bản mà DrawPrimitive sẽ sử dụng để render những vecto trong vertex buffer. Và như vậy trong ví dụ trước ta đã chọn kiểu triangle strip với mục đích đơn giản hóa việc vẽ và tạo khả năng cho thêm các tam giác một cách

dễ dàng. Phần tiếp theo ta sẽ giải thích chi tiết hơn một chút về sự khác nhau giữa các kiểu cơ bản.

Point list

Một point list bao gồm một loạt các điểm không nối với nhau. Hình 4.9 biểu diễn một lưới tọa độ chứa bốn điểm riêng biệt. Mỗi điểm được xác định qua các tọa độ X, Y và Z. Ví dụ như điểm cao nhất ở phía trái được xác định qua (1, 6, 0).

Line List

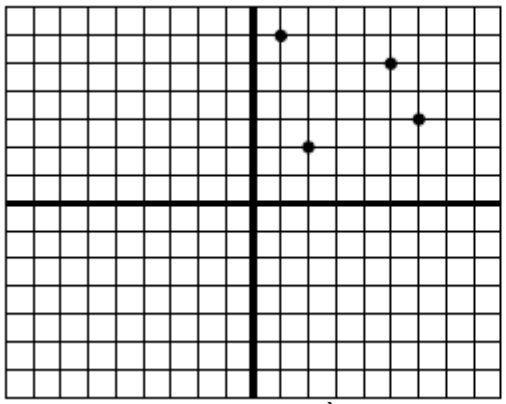
Một line list bao gồm các đoạn thẳng đi qua hai điểm. Những đoạn thẳng thuộc danh sách đoạn thẳng thì không nối liền với nhau. Hình 4.10 biểu diễn hai đoạn thẳng render bằng (line list). Nó được xây dựng từ bốn vecto. Đoạn thẳng phía bên trái có điểm phía trên là (-6, 5, 0) và điểm phia dưới là (-4, 2, 0).

Line Strip

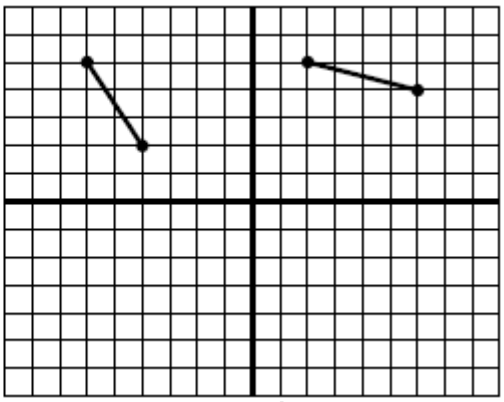
Một line strip là một loạt các đoạn thẳng nối nhau, trong đó mỗi đoạn thẳng thêm vào được xác định chỉ bằng một vecto. Mỗi vecto trong line strip nối với một vecto liền trước tạo thành một đoạn thẳng. Hình 4.11 biểu diễn một line strip, nó được xây dựng thông qua 6 vecto và sinh ra 5 đoạn thẳng.

Triangle List

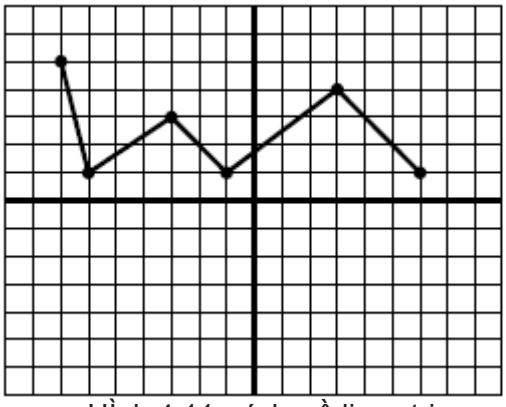
Một triangle list bao gồm các tam giác không nối với nhau và xuất hiện tự do bất cứ đâu trong hệ tọa độ. Hình 4.12 biểu diễn hai tam giác được tạo ra từ 6 vecto. Mỗi tam giác cần 3 vecto.



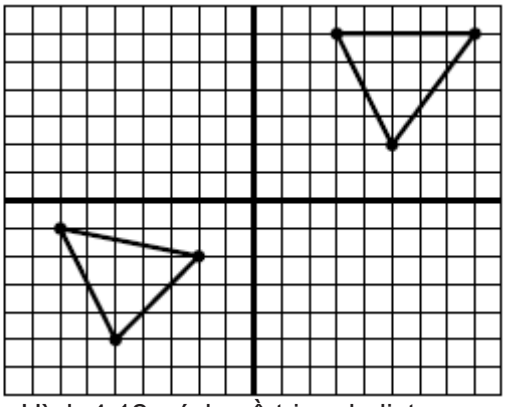
Hình 4.9: ví dụ về point list



Hình 4.10: ví dụ về line list



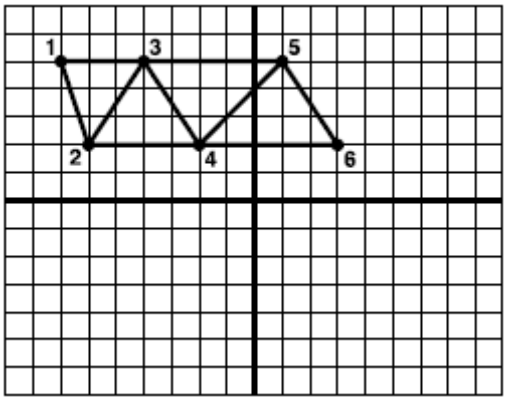
HÌnh 4.11: ví dụ về line strip



Hình 4.12: ví dụ về triangle list

Triangle Strip

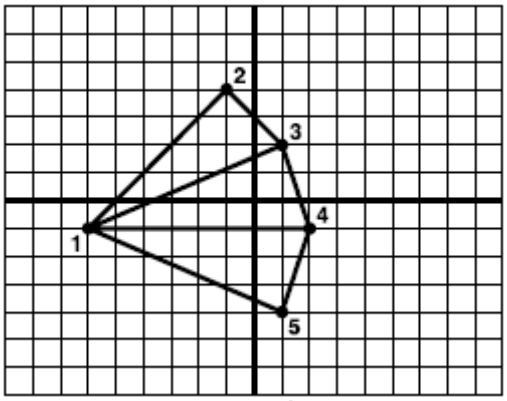
Một triangle strip là một loạt các tam giác nối với nhau trong đó mỗi tam giác thêm vào được xác định chỉ bằng một vecto. Hình 4.13 biểu diễn 4 tam giác được tạo ra chỉ bằng 6 vecto.
Những triangle strip được xây dưng bằng cách đưa vào 3 vecto đầu tiên để định nghĩa một tam giác. Sau đó, mỗi khi một vecto được thêm vào, thì sẽ sinh ra hai đoạn thẳng nối 2 vecto được thêm vào sau cùng với vecto vừa thêm tạo thành một tam giác mới. Trong hình 4.13, biểu diễn cả thứ tự của các vecto được thêm vào.



Hình 4.13: ví dụ về triangle strip

Triangle Fan

Một triangle fan là một loạt các tam giác có chung nhau một vecto. Sau khi tam giác đầu tiên được tạo thành, mỗi một vecto thêm vào sẽ tạo thành một tam giác mới có một đỉnh là vecto được thêm vào đầu tiên, một đỉnh là vecto được thêm vào cuối cùng, đỉnh còn lại là vecto vừa thêm vào.
Hình 4.14 biểu diễn ba tam giác được tạo thành từ 5 vecto. Thứ tự của các vecto sẽ quy định hình dạng của triangle fan. Hình 4.14 cũng biểu diễn cả trình tự đưa các vecto vào để tạo thành một triangle fan như như bạn thấy.



Hình 4.14: ví dụ về triangle fan

Tổng kết chương

Chương này, chỉ mới nêu ra những khái niệm cơ bản về 3D. Trong các chương tiếp theo, bạn sẽ được học thêm một số chủ đề nâng cao hơn, nhưng ngay từ bây giờ bạn cần phải hiểu một cách rõ ràng về công dụng của vertex buffers.

Giờ đây bạn đã có những khái niệm cơ bản về cách làm việc của không gian 3D và làm thế nào để định nghĩa một vật thể ở trong nó, đã đến lúc để học cách len lỏi sâu hơn vào cái giới này. Trong chương tiếp theo, bạn sẽ được học cách xoay và di chuyển một vật thể.

Những kiến thức bạn đã được học

Trong chương này bạn đã được học:

- Sự khác nhau giữa không gian 2D và 3D.
- Vecto là gì và cách xác định chúng
- Cách tạo và sử dụng vertex buffer
- Cách render các vecto.
- Các kiểu cơ bản khác nhau được đề cập trong Direct3D.

Các câu hỏi ôn tập

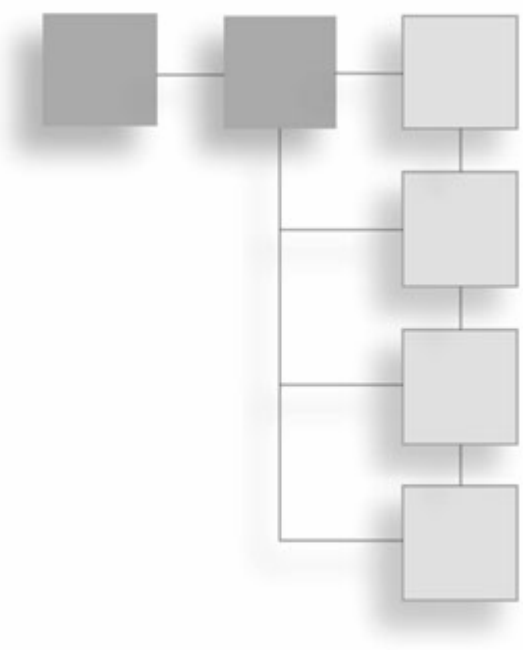
Bạn có thể tìm thấy đáp án của các câu hỏi và bài tập ở phần phụ lục A, "Đáp án phần bài tập cuối chương".

1. Cách xác định một điểm trong hệ tọa độ 3D?
2. Trục tọa độ nào thường dùng để diễn tả chiều sâu?
3. Hàm SetFVF dùng để làm gì?
4. Kiểu cơ bản nào dùng để xây dựng một loạt các đoạn thẳng liên tiếp nhau?
5. Cần có bao nhiêu vecto để xây dựng một triangle strip gồm 5 tam giác?

Bài tập

1. Viết một hàm để render một line list bao gồm 4 đoạn thẳng.
2. Viết một hàm để render một số tam giác sử dụng kiểu triangle list.

CHƯƠNG 5



MA TRẬN, PHÉP BIẾN ĐỔI VÀ PHÉP XOAY

Phần lớn những người mới bắt đầu đều tin rằng ma trận và toán học 3D là phần khó nhất trong lập trình đồ họa. Điều này có thể đúng trong một vài năm trước đây, nhưng bây giờ thì không. Direct3D đã có nhiều cải tiến trong suốt thời gian qua và loại bỏ được rất nhiều công việc cồng kềnh, phức tạp, giúp người lập trình có điều kiện để tập trung hơn vào cái mà họ muốn.
Chương này sẽ giới thiệu cho bạn về ma trận và cho thấy nó giúp bạn giải quyết công việc đơn giản như thế nào.
Những phần mà bạn sẽ được học trong chương này:

- Mô hình 3D là gì và cách tạo ra nó
- Cách tối ưu thao tác render bằng cách sử dụng “index buffers”
- Khái niệm về “geometry pipeline” và các giai đoạn của nó
- Ma trận là gì và nó có tác dụng gì với thế giới 3D
- D3DX giúp gì cho công việc của bạn
- Tác động lên các vật thể 3D trong một scene.
- Cách tạo một camera ảo.

Tạo một mô hình 3D

Giờ đây khi bạn đã biết cách vẽ một tam giác, đã đến lúc để mở rộng kiến thức và tạo ra một mô hình 3D đầy đủ. Hầu hết mọi thứ trong game đều được biểu diễn bằng các đối tượng 3D, từ nhân vật bạn điều khiển cho đến môi trường mà nhân vật đó tác động lên. Một đối tượng 3D có thể được tạo ra từ một đa giác đơn lẻ cho đến hàng ngàn các đa giác, tùy thuộc vào cái mà mô hình biểu diễn. Những thành phố đầy ô tô, các tòa nhà và người có thể được biểu diễn theo cách này.
Một đối tượng 3D dù rất đáng sợ, nhưng hãy nghĩ rằng chúng chỉ là một tập hợp những hình tam giác được liên kết với nhau mà thôi. Bằng cách chia nhỏ một mô hình ra thành các hình cơ bản, ta có thể nắm bắt được nó dễ dàng hơn.

Tôi sẽ chỉ cho bạn các bước cần thiết để có thể tạo ra và render một hình hộp. Một hình hộp không phải là một đối tượng phức tạp, nhưng nó sẽ cho bạn những nền tảng cần thiết để xây dựng bất kỳ mô hình 3D nào.

Định nghĩa một Vertex Buffer

Ở chương 4, "cơ bản về 3D", bạn đã được giới thiệu về vertex buffer như là một nơi sạch sẽ và dễ dùng để lưu trữ các vecto. Khi mà các vật thể ngày càng trở lên phức tạp, sự tiện lợi của vertex buffer lại càng rõ ràng hơn. Vertex buffer là một chỗ lý tưởng để lưu trữ các vecto của một đối tượng, cho phép bạn dễ dàng truy cập và render chúng bằng các phương thức rất đơn giản.
Phần trước bạn chỉ dùng vertex buffer lưu trữ ba vecto để tạo một hình tam giác. Khi muốn tạo một đối tượng phức tạp hơn, bạn sẽ cần lưu trữ nhiều vecto hơn.
Khi ta định nghĩa những vecto cho một vật thể cố định, hãy coi như ta lưu trữ chúng trong một mảng. Mảng này có kiểu là CUSTOMVERTEX, như đã đề cập ở chương 4, nó cho phép bạn định nghĩa một layout cho dữ liệu vecto của bạn. Mỗi thành phần của mảng chứa những thông tin mà Direct3D cần để mô tả một vecto. Đoạn code sau sẽ định nghĩa các vecto cho một hình hộp.

```
// Cấu trúc CUSTOMVERTEX
struct CUSTOMVERTEX
{
        FLOAT x, y, z; // vị trí 3D chưa qua biến đổi của vecto
        DWORD color; // màu của vecto
};
CUSTOMVERTEX g_Vertices[] =
{
        // 1
        { -64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { -64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        // 2
        { -64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { -64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        // 3
        { -64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { -64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        // 4
        { -64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { -64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        // 5
        { 64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        { 64.0f, -64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        // 6
        {-64.0f, 64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        {-64.0f, -64.0f, -64.0f, D3DCOLOR_ARGB(0,0,0,255)},
        {-64.0f, 64.0f, 64.0f, D3DCOLOR_ARGB(0,0,0,255)},
```

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

Việc đầu tiên mà đoạn code thực hiện là khai báo cấu trúc CUSTOMVERTEX. Cấu trúc này gồm có hai phần: thứ nhất là vị trí thông qua X, Y và Z; thứ hai là màu.
Sau khi định nghĩa cấu trúc này, mảng g_Vertices được tạo ra và nạp dữ liệu đủ để mô tả một hình hộp. Dữ liệu vecto được phân ra thành sáu phần, mỗi phần biểu diễn một mặt của hình hộp.. Ở phần trước, bạn đã luôn luôn gán giá trị 1.0f cho biến Z, tức là tạo ra một đối tượng phẳng. Nhưng với hình hộp bạn cần tạo ra một mô hình 3D thật sự, giá trị Z lúc này sẽ được dùng để xác định khoảng cách của các vecto trong không gian.
Bước tiếp theo là tạo và nạp dữ liệu cho vertex buffer trên cơ sở dữ liệu vecto khai báo ở phần vừa rồi. Đoạn code sau thực hiện điều đó:

```
// tạo một vertex buffer
HRESULT hr;
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
// tạo ra một vertex buffer chứa dữ liệu mô tả hình hộp
hr = pd3dDevice->CreateVertexBuffer(sizeof(g_Vertices) * sizeof(CUSTOMVERTEX),
                                    0,
                                    D3DFVF_CUSTOMVERTEX,
                                    D3DPOOL_DEFAULT,
                                    &vertexBuffer,
                                    NULL );
// Kiểm tra giá trị trả về của CreateVertexBuffer
if FAILED (hr)
        return false;
// chuẩn bị để nạp dữ liệu cho vertex buffer
VOID* pVertices;
// khóa vertex buffer
hr = vertexBuffer->Lock(0, sizeof(g_Vertices), (void**) &pVertices, 0);
// Kiểm tra xem vertex buffer đã được khóa chưa
if FAILED (hr)
        return false;
// copy dữ liệu vào vertex buffer
memcpy ( pVertices, g_Vertices, sizeof(g_Vertices) );
// Mở khóa vertex buffer
vertexBuffer->Unlock();
```

Sử dụng lời gọi tới CreateVertexBuffer để tạo một vertex buffer; đồng thời xác định luôn kích thước và kiểu của nó. Thay vì chỉ ra kích thước của vertex buffer ngay từ đầu, ta đã sử dụng hàm sizeof để tính toán nó lúc biên dịch. Nhân kích thước của mảng g_Vertices với kích thước của cấu trúc CUSTOMVERTEX ta có chính xác kích thước của vertex buffer dùng để lưu trữ toàn bộ các vecto cần dùng.
Sau đó ta tiến hành khóa buffer, và copy những vecto chứa trong mảng g_Vertices vào đó qua hàm memcpy.
Sau khi ta nạp đầy dữ liệu cho vertex buffer, đã đến lúc để bạn vẽ đối tượng 3D của mình.

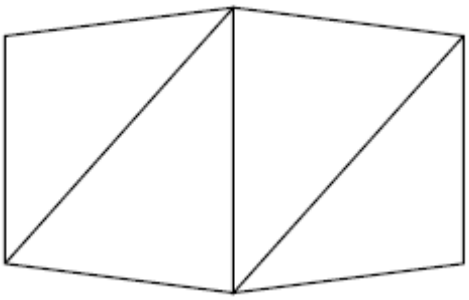
Render hình hộp

Render một hình hộp cũng chỉ giống như vẽ những đối tượng khác từ một vertex buffer, bất kể là nó phức tạp thế nào. Sự khác nhau chủ yếu phân biệt giữa hình hộp, tam giác, ô tô là ở số vecto được dùng. Sau khi đối tượng được lưu vào vertex buffer, thật dễ dàng để render nó.

Hàm render dưới đây nêu chi tiết quá trình render một hình hộp xác định thông qua mảng g_Vertices.

```
/*****************************************************************************
* Render
*****************************************************************************/
void Render(void)
{
        // Xóa back buffer bởi màu trắng
        pd3dDevice->Clear( 0,
                           NULL,
                           D3DCLEAR_TARGET,
                           D3DCOLOR_XRGB(255,255,255),
                           1.0f,
                           0 );
pd3dDevice->BeginScene();
// Cài đặt luồng
pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );
// Cài đặt định dạng cho vecto
pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
// Gọi DrawPrimitive để vẽ một hình hộp
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 4, 2 );
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 8, 2 );
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );
pd3dDevice->EndScene();
// Hiển thị back buffer
pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Để render hình hộp đầu tiên ta cài đặt nguồn luồng và định dạng vecto. Sự khác biệt lớn nhất giữa việc vẽ một tam giác với việc render hình hộp 3D bằng nhiều tam giác là ở chỗ ta đã sử dụng nhiều lời gọi tới DrawPrimitive.
Mỗi lệnh gọi tới DrawPrimitive trong 6 lênh ở trên sẽ render một mặt của hình hộp bằng cách sử dụng kiểu triangle strip.
Hình 5.1 là hình hộp mà ta nhận được. Hình hộp này được render ở dạng khung dây (wire frame), bạn có thể thấy được các tam giác tạo lên nó.



Hình 5.1: Hình hộp 3D đầy đủ

Index Buffer (bộ đệm chỉ mục)

Index buffer là những vùng nhớ lưu trữ dữ liệu về chỉ mục. Mỗi chỉ mục trong Index buffer đại diện cho một vecto trong vertex buffer. Sử dụng các chỉ mục này giúp giảm lượng dữ liệu cần chuyển tới card đồ họa vì ta chỉ cần gửi một giá trị duy nhất đại diện cho mỗi vecto thay vì dữ liệu đầy đủ về vecto như X, Y, Z… Như vậy dữ liệu về vecto nằm trong vertex buffer và được tham chiếu đến thông qua index buffer.
Bạn có thể tạo ra một Index buffer có kiểu IDirect3DIndexBuffer9 thông qua hàm CreateIndexBuffer, được định nghĩa như sau:

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

Hàm CreateIndexBuffer có 6 đối số:

- **Length**. Kích thước của index buffer theo byte.
- **Usage**. Giá trị kiểu D3DUSAGE quy định cách dùng index buffer.
- **Format**. Định dạng cho các phần tử của index buffer (các chỉ số). Có 2 lựa chọn: D3DFMT_INDEX16 hoặc D3DFMT_INDEX32.
  D3DFMT_INDEX16 nghĩa là mỗi phần tử có 16 bit, và D3DFMT_INDEX32 nghĩa là mỗi phần tử có 32 bit.
- **Pool**. Vùng nhớ được dùng cho index buffer.
- **ppIndexBuffer**. Địa chỉ của vùng nhớ nơi chứa index buffer được tạo ra.
- **pHandle**. Giá trị này thường để là NULL.

Đoạn code ví dụ về CreateIndexBuffer.

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

Lời gọi tới CreateIndexBuffer tương tự như với CreateVertexBuffer ở phần trước. Sự khác biệt chủ yếu giữa hai hàm này là ở đối số thứ ba, nó định dạng cho các phần tử (các chỉ số) chứa trong index buffer. Bạn có 2 lựa chọn 16 hoặc 32 bit tương ứng với các chỉ số có kiểu WORD hay DWORD.
Ở phần trước, chúng ta đã tạo một hình hộp với vertex buffer. Hình hộp này cần 24 vecto (trong đó có nhiều vecto trùng nhau) để tạo 12 mặt tam giác. Sử dụng index buffer, bạn có thể tạo ra một hình hộp tương tự như vậy mà chỉ cần 8 vecto. Phần tiếp theo sẽ trình bày cách để thực hiện điều đó.

Tạo một hình hộp với Index Buffer

Bước thứ nhất để tạo một hình hộp với index buffer là định nghĩa các vecto và chỉ số. Ở đây ta định nghĩa các vecto có cấu trúc CUSTOMVERTEX giống như phần trước. Mỗi vecto bao gồm các thành phần X, Y, Z và màu.

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

```
};
```

Sau khi đã định nghĩa các vecto, bước tiếp theo là phát sinh mảng chỉ số. Chỉ số, cũng giống như vecto, được lưu vào trong một mảng. Như đã đề cập ở trên các chỉ số có thể có định dạng là 16 hoặc 32 bit. Đây là lúc ta sử dụng nó.
Đoạn code sau khai báo mảng các chỉ số tương ứng với các vecto dùng để tạo hình hộp.

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

Mảng IndexData ở trên chia 36 chỉ số thành 12 nhóm, mỗi nhóm bao gồm 3 giá trị cần dùng để tạo 1 mặt tam giác. Như vậy ta có 12 tam giác, cứ 2 tam giác xác định một mặt của hình hộp.

Chú ý:

Hãy nhớ rằng: Nếu bộ nhớ hạn hẹp và mô hình của bạn không cần đến kiểu chỉ số DWORD, thì bạn nên dùng kiểu WORD.

Tạo và nạp dữ liệu cho Index Buffer

Sau khi định nghĩa xong dữ liệu cần thiết cho index buffer, bạn cần copy dữ liệu này vào index buffer. Bước này tương tự như copy vecto vào vertex buffer.
Đầu tiên, bạn khóa buffer bằng hàm Lock. Sau đó, bạn copy các các chỉ số định nghĩa ở trên vào trong index buffer bằng hàm memcpy và kết thúc bằng việc mở khóa cho buffer. Kết quả nhận được là một index buffer chứa các chỉ số bạn cần để render hình hộp ta cần.
Đoạn code sau thực hiện quá trình tạo và nạp dữ liệu cho index buffer.

```
// index buffer
LPDIRECT3DINDEXBUFFER9 iBuffer;
HRESULT hr;
// tạo index buffer
hr = pd3dDevice->CreateIndexBuffer(sizeof(IndexData)*sizeof(WORD),
                                   D3DUSAGE_WRITEONLY,
                                   D3DFMT_INDEX16,
                                   D3DPOOL_DEFAULT,
                                   &iBuffer,
                                   NULL);
// kiểm tra kết quả tạo index buffer
if FAILED(hr)
```

```
            return false;
// chuẩn bị copy copy các chỉ số vào index buffer
VOID* IndexPtr;
// khóa index buffer
hr = iBuffer ->Lock(0, 0, (void**)& IndexPtr, D3DLOCK_DISCARD);
// kiểm tra xem index buffer đã được khóa chưa
if FAILED (hr)
            return hr;
// thực hiện quá trinh copy vào buffer
memcpy( pVertices, IndexData, sizeof(IndexData) );
// mở khóa index buffer
iBuffer->Unlock();
```

Sau khi nạp dữ liệu xong cho index buffer, bạn có thể dùng kết hợp giữa vecto và chỉ số để render đối tượng ta cần.

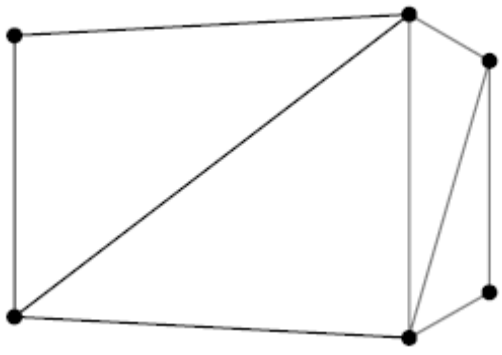
Rendering hình hộp với Index Buffer

Phần trước, khi thực hiện vẽ với vertex buffer, ta dùng hàm DrawPrimitive. Hàm DrawPrimitive sử dụng dữ liệu có trong vertex buffer để tạo các đối tượng cơ bản như các tam giác nối nhau hoặc các tam giác riêng lẻ. Bạn có thể vẽ bằng cách tương tự như vậy với index buffers và hàm DrawIndexedPrimitive.
Hàm DrawIndexedPrimitive sử dụng index buffer như là nguồn dữ liệu và render các hình cơ bản để tạo ra đối tượng 3D. Hàm index buffer được định nghĩa như sau:

```
HRESULT DrawIndexedPrimitive(
                D3DPRIMITIVETYPE Type,
                INT BaseVertexIndex,
                UINT MinIndex,
                UINT NumVertices,
                UINT StartIndex,
                UINT PrimitiveCount
);
```

Hàm DrawIndexedPrimitive có 6 đối số :
- **Type**. Kiểu cơ bản được sử dụng khi render
- **BaseVertexIndex**. Chỉ số đầu tiên trong vertex buffer
- **MinIndex**. Chỉ số nhỏ nhất trong lời gọi.
- **NumVertices**. Số lượng vecto trong lời gọi.
- **StartIndex**. Vị trí đầu tiên để đọc dữ liệu từ mảng vecto
- **PrimitiveCount**. Số hình cơ bản cần vẽ.

Hình 5.2 biểu diễn một hình hộp render ở chế độ khung dây và tô đậm ở các đỉnh. Các vecto ở đỉnh biểu thị cho các vecto được đại diện trong index buffer qua các chỉ số.



Hình 5.2: Hình hộp

```
// cài đặt chỉ số
m_pd3dDevice->SetIndices( m_pDolphinIB );
// gọi hàm DrawIndexedPrimitive để vẽ thông qua các chỉ số
```

```
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                    0, // chỉ số đầu tiên trong vectex buffer
                                    0, // chỉ số nhỏ nhất
                                    m_dwNumDolphinVertices, // số vecto
                                    0, // chỉ số bắt đầu
                                    m_dwNumDolphinFaces ); // số hình
```

Trước khi ta gọi tới DrawIndexedPrimitive ta cần gọi hàm SetIndices trước. Hàm SetIndices, được định nghĩa ở dưới, thông báo với Direct3D rằng index buffer nào sẽ được dùng làm dữ liệu vẽ. Hàm SetIndices hoạt động giống như hàm SetStreamSource khi ta sử vertex buffer.

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

Hàm SetIndices chỉ có một đối số: con trỏ tới một index buffer hợp lệ.

Hệ thống chuyển đổi hình học (the Geometry Pipeline)

Ở các phần trên, ta đã sử dụng hệ tọa độ được quy đổi trước để vẽ các vật thể lên màn hình. Điều đó có nghĩa là vị trí của đối tượng về cơ bản đã được định nghĩa trước ở trên màn hình. Việc đó làm hạn chế không gian của ta và sự chuyển động của các vật thể ở trong nó.

Phần lớn mô hình 3D không được tạo ra thông qua code. Ví dụ như, nếu bạn làm một game đua xe, chắc hẳn bạn sẽ tạo mô hình ô tô bằng các phần mềm dựng 3D. Trong suốt quá trình đó, ta có thể làm việc với mô hình một cách độc lập với hệ tọa độ tổng thể. Tức là các đối tượng sẽ được tạo ra từ tập hợp các vecto mà không cần quan tâm đến vị trí chính xác ở đâu và cách đặt như thế nào trong môi trường game. Chính vì lý do này mà bạn cần phải tự mình di chuyển và xoay các mô hình theo ý của bạn.

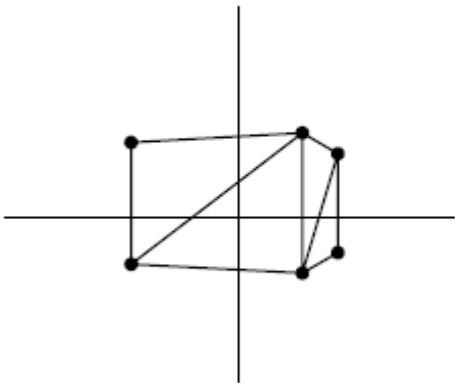
Bạn có thể thực hiện điều đó qua hệ thống chuyển đổi hình học. Hệ thống này là là một quá trình cho phép bạn biến đổi các đối tượng từ hệ tọa độ này sang hệ tọa độ khác.

Khi một mô hình được khởi dựng, nó thường được đặt vào trung tâm ở gốc tọa độ. Nó làm cho mô hình được đặt vào trung tâm của môi trường theo một hướng mặc định.

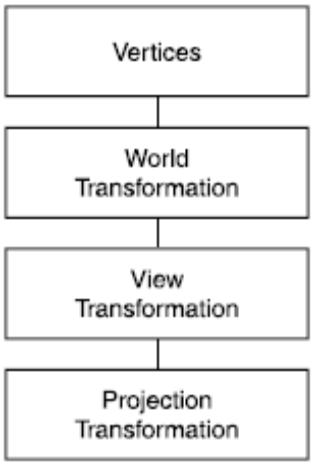
Không phải tất cả mô hình bạn nạp vào đều nằm ở gốc tọa độ, vậy làm thế nào để đặt mô hình vào đúng vị trí? Câu trả lời là sử dụng các phép biến đổi.

Hình 5.3 biểu diễn một hình hộp được đặt ở trung tâm tại gốc tọa độ.

Các phép biến đổi bao gồm dịch chuyển, xoay, tỉ lệ. Thực hiện chúng với mô hình, bạn có thể đặt nó theo ý muốn. Các phép biến đổi đó được thực hiện thông qua hệ thống chuyển đổi hình học.



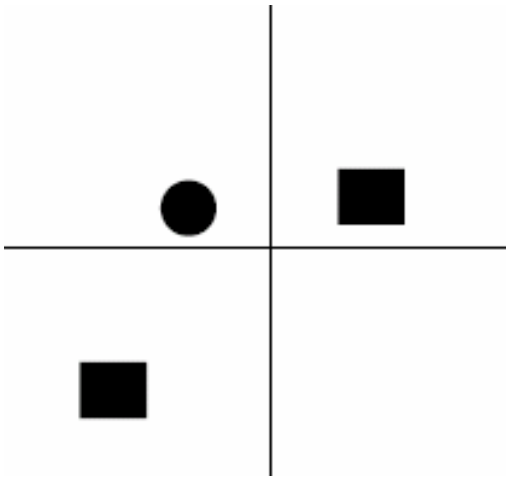
Hình 5.3: Hình hộp được đặt ở trung tâm gốc tọa độ.



Khi bạn nạp vào một mô hình, các vecto của nó được đặt trong một hệ tọa độ địa phương gọi là không gian mô hình. Không gian mô hình thì liên hệ với hệ tọa độ tổng thể nơi mà mô hình được đặt vào. Ví dụ như, trong lúc tạo đối tượng, các vecto của đối tượng sẽ liên quan đến một điểm gốc nằm ở gần chúng. Một hình hộp kích thước 2 đơn vị được đặt ở trung tâm thì các vecto của nó sẽ cách gốc tọa độ là 1 đơn vị theo các trục. Nếu bạn muốn đặt hình hộp này ở đâu đó khác, bạn cần phải biến đổi các vecto của nó từ hệ tọa độ địa phương sang hệ tọa độ tổng thể. Hệ tọa độ tổng thể này, được gọi là (world space) và quá trình biến đổi các vecto sang hệ thống này gọi là (world transformation).

World Transformation

Giai đoạn (world transformation) của hệ chuyển đổi hình học sẽ đưa một đối tượng với hệ toạ độ địa phương của nó sang hệ toạ độ tổng thể.
Hệ toạ độ tổng thể là một hệ thống mà các vật thể được đặt ở đúng vị trí của nó trong không gian 3D. Các mô hình sau khi được chuyển đổi về hệ tổng thể này sẽ được gắn với một điểm gốc duy nhất của hệ. Hình 5.5 biểu diễn các đối tượng 3D được gắn với một gốc toạ độ .
Giai đoạn tiếp theo của hệ chuyển đổi hình học là (view transformation). Bởi vì tất cả các đối tượng ở thời điểm này đã được gắn với một điểm gốc duy nhất, do đó bạn chỉ có thể quan sát chúng từ điểm này. Để có thể quan sát khung cảnh từ một điểm bất kì, chúng ta cần đến phép biến đổi view transformation.



Hình 5.5: Các đối tượng gắn chung với 1 điểm gốc

View Transformation

View transformation biến đổi tọa độ từ không gian thực sang không gian camera. Không gian camera được gắn với một hệ tọa độ để xác định vị trí của nó. Khi ta đặt một điểm nhìn cho camera (ảo) thì hệ tọa độ của không gian thực sẽ thay đổi tương ứng với camera đó.

Chú ý:

Ở đây tôi nói là "camera ảo" thay cho "camera" vì khái niệm camera trong không gian 3D thực ra không tồn tại. Bằng cách di chuyển camera(ảo) dọc lên theo trục Y hoặc di chuyển toàn bộ không gian thực dọc xuống theo trục Y, ta đều đạt được kết quả như nhau.

Với góc quay và điểm nhìn của camera, bạn đã có thể biểu diễn mọi thứ lên màn hình.

Projection Transformation

Giai đoạn tiếp theo trong hệ thống chuyển đổi hình học là projection transformation. projection transformation tác động đến chiều sâu của không gian. Khi một vật thể nằm gần camera thì trông nó sẽ lớn hơn so với khi nó ở cách xa camera, điều đó tạo ra cảm giác về độ sâu.
Các vecto theo đó sẽ được chuyển về dạng 2D. Và kết quả là một ảnh 2D mô phỏng cho khung cảnh 3D được kết xuất ra màn hình. Bảng 5.1 cho thấy các kiểu biến đổi trong hệ chuyển đổi hình học và kiểu của các không gian tương ứng mà nó tác dụng lên.

Table 5.1 Coordinate System Transformations

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

Ma trận là gì?

Ma trận là một mảng chia thành các hàng và các cột. Dưới đây là ma trận 4x4 chứa các giá trị từ 1 đến 16.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Ma trận được dùng trong 3D cho các phép biến đổi. Các giá trị chứa trong ma trận được dùng để dịch chuyển, xoay, tỉ lệ đối tượng. Mỗi hàng trong ma trận biểu diễn một trục trong hệ tọa độ. Hàng thứ nhất chứa tọa độ trên trục x, hàng thứ hai chứa tọa độ trên trục y, hàng thứ 3 chứa tọa độ trên trục z. Mỗi phần tử trong ma trận biểu diễn một thành phần của phép biến đổi. Ví dụ như, các phần tử13, 14, 15 chứa vị trí X, Y, Z hiện tại của một vecto. Các phần tử 1, 6 và 11 chứa các hệ số tỉ lệ. Đoạn code sau định nghĩa một ma trận:

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

Hàng cuối cùng của ma trận trên biểu diễn một đối tượng có tọa độ là X=2.0f, Y=3.0f, Z=2.0f.

Ma trận đồng nhất

Ma trận đồng nhất là ma trận mặc định, nó đặt vật thể ở gốc tọa độ với tỉ lệ thu phóng là 1. Giá trị của các phần tử 1, 6, 11 được gán là 1, nhằm tạo ra đối tượng với hệ số thu phóng là 1. Các phần tử 13, 14, 15 thì có giá trị là 0. Ma trận đồng nhất được định nghĩa như dưới đây:

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

Nếu bạn muốn đưa một vật thể trở về gốc tọa độ, bạn có thể biến đổi các vecto của nó thông qua ma trận đồng nhất. Vật thể qua đó sẽ được đưa trở về gốc tọa độ mà không hề bị xoay, thu phóng. Sau đó bạn có thể tự do dịch chuyển vật thể đến bất cứ chỗ nào bạn muốn.

Initializing a Matrix

Tạo mới hay thay đổi một ma trận cũng đơn giản như việc thay đổi các phần tử ở trong một mảng. Ví dụ như, nếu bạn có một ma trận đồng nhất và bạn muốn biến nó thành một ma trận biến đổi có thể dịch chuyển một vật thể 5 đơn vị theo trục X và 3 đơn vị theo trục Y, bạn có thể thay đổi nội dung của ma trận đó như sau:

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

Và khi đó ma trận sẽ có dạng:

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

Kết quả là ta có một chứa các thông số cần thiết để có thể dịch chuyển vật thể như yêu cầu ở trên.

Phép nhân ma trận

Chắc hẳn bạn đang thắc mắc ma trận tác dụng như thế nào lên các vecto của vật thể. Tốt thôi, ta chỉ cần nhân từng vecto của vật thể với ma trận để có được các vecto biến đổi. Về mặt toán học thì điều này khá đơn giản. Thành phần X của vecto sẽ được biến đổi khi ta nhân hàng đầu tiên của ma trận với vecto. Khi đó vecto được biến đổi có thể nhận được thông qua việc tổng hợp kết quả của các phép nhân đó. Công thức cho phép nhân đó là:

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

Ma trận đầu tiên là vecto cần biến đổi. Ma trận tiếp theo là ma trận biến đổi. Ma trận thứ ba là vecto sau khi đã biến đổi.
Bạn có thể thực hiện phép nhân ma trận bằng cách nhân hàng của ma trận này với cột của ma trận kía.

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

Lấy hàng của ma trận bên trái nhân với cột của ma trận bên phải. Ví dụ ta lấy hàng 1 của ma trận trái nhân cột 1 của ma trận phải ta sẽ có:

A × 1 + B × 5 + C × 9 + D × 13

Chú ý:

phép nhân ma trận không có tính hoán vị, nếu ta đổi chỗ hai ma trận này với nhau thì ta có thể sẽ có một ma trận kết quả khác. Vì vậy, thứ tự của các ma trận là yếu tố rất quan trọng trong phép nhân ma trận.

Cách định nghĩa một ma trận trong Direct3D

Direct3D đơn giản hóa việc tạo ma trận thông qua cấu trúc D3DMATRIX như sau:

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

Sử dụng cấu trúc D3DMATRIX mà Direct3D cung cấp, bạn còn được cung cấp thêm nhiều hàm tiện lợi khác khi thao tác trên ma trận ví dụ như hàm khởi tạo ma trận.

D3DX làm cho ma trận đơn giản hơn

Phần trên, ta đã được giới thiệu về cấu trúc D3DMATRIX mà Direct3D cung cấp. Nó giúp đơn giản hóa việc định nghĩa và quản lý ma trận nhưng vẫn bắt bạn phải tự thực hiện tính toán trên đó, tuy vậy thư viện D3DX sẽ giúp bạn điều này.
Thư viện D3DX cũng có một cấu trúc là D3DXMATRIX. Các thành phần trong D3DXMATRIX cũng giống như của D3DMATRIX, nhưng D3DXMATRIX cung cấp thêm

một vài tiện ích khác nữa. Nó cung cấp một vài hàm cho phép bạn thực hiện tính toán, so sánh trên đó.
Cấu trúc D3DXMATRIX được định nghĩa như sau:

```
typedef struct D3DXMATRIX : public D3DMATRIX {
    public:
    D3DXMATRIX() {};
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );
    // access grants
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT operator () ( UINT Row, UINT Col ) const;
    // casting operators
    operator FLOAT* ();
    operator CONST FLOAT* () const;
    // assignment operators
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );
    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;
    // binary operators
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;
    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );
    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;
} D3DXMATRIX, *LPD3DXMATRIX;
```

Điều bạn có thể thấy đầu tiên là D3DXMATRIX là một cấu trúc kế thừa từ D3DMATRIX và bổ xung thêm một số hàm khiến cho nó giống như một lớp (class) ở trong C++. Theo cách định nghĩa này, bạn chỉ có thể truy cập tới nó thông qua C++, và nó chỉ được sử dụng như là một lớp (class) thật sự với các thành phần public.
Nhìn qua cấu trúc này, bạn có thể thấy nó viết chồng (overload) rất nhiều toán tử dùng cho việc tính toán trên ma trận. Bởi vì, cấu trúc D3DXMATRIX rất hữu dụng, nên chúng ta sẽ sử dụng nó nhiều ở các phần tiếp theo.

Điều khiển các đối tượng 3D thông qua ma trận

Như vậy, ta đã biết sơ qua về khái niệm ma trận, ta sẽ tiếp tục xem xét tính hữu dụng của nó. Ta sử dụng ma trận khi muốn kiểm soát các đối tượng trong không gian. Cho dù bạn muốn di chuyển đối tượng lòng vòng hay chỉ đơn giản là xoay nó, bạn đều cần phải thực hiện thông qua ma trận. D3DX cung cấp hàng loạt các hàm để bạn có thể kiểm soát các đối tương dễ dàng hơn thông qua ma trận. Dưới đây là một trong số đó:

- **D3DXMatrixIdentity**. Làm sạch ma trận (biến nó về lại ma trận đồng nhất).

- **D3DXMatrixRotationX**. Xoay đối tượng quanh trục X.
- **D3DXMatrixRotationY**. Xoay đối tượng quanh trục Y.
- **D3DXMatrixScaling**. Phép tỉ lệ (thu phóng) một đỗi tượng.
- **D3DXMatrixTranslation**. Dịch chuyển đối tượng theo các trục.

Di chuyển đối tượng

Để di chuyển đối tượng lòng vòng trong game, bạn cần phải thực hiện phép tịnh tiến. Phép tịnh tiến sẽ tác động làm cho đối tượng di chuyển theo các trục tọa độ. Nếu bạn muốn di chuyển đối tượng sang phải chẳng hạn, bạn có thể tịnh tiến nó dọc theo trục X về hướng dương.
Tịnh tiến các vật thể được thực hiện thông qua hàm D3DXMatrixTranslation như dưới đây:

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

Hàm D3DXMatrixTranslation cần 4 đối số.

- **pOut**. Ma trận đầu ra. Con trỏ đến đối tượng D3DXMATRIX.
- **x**. Khoảng dịch chuyển theo trục X. Chấp nhận cả âm hoặc dương.
- **y**. Khoảng dịch chuyển theo trục Y.
- **z**. Khoảng dịch chuyển theo trục Z.

Đoạn code sau cho thấy cách sử dụng hàm D3DXMatrixTranslation

```
D3DXMATRIX matTranslate;
D3DXMATRIX matFinal;
// Đưa ma trận matFinal về ma trận đồng nhất
D3DXMatrixIdentity(&matFinal);
// tịnh tiến đối tượng 64 đơn vị về bên phải theo trục X.
// Ma trận kết quả lưu vào matTranslate
D3DXMatrixTranslation(&matTranslate, 64.0f, 0.0f, 0.0f);
// Nhân ma trận tịnh tiến và ma trận đồng nhất để có ma trận biến đổi
// ma trận biến đổi lưu trữ trong finalMat
D3DXMatrixMultiply(&finalMat, &finalMat, & matTranslate);
// thực hiện phép biến đổi đối tượng trong không gian thực
pd3dDevice->SetTransform(D3DTS_WORLD, &finalMat);
```

Hàm D3DXMatrixTranslation được sử dụng để tịnh tiến đối tượng 64 đơn vị về bên phải trục X. Để thực hiện phép biến đổi này với đối tượng, ta nhân ma trận tịnh tiến với ma trận đồng nhất, sau đó đối tượng được quy đổi về không gian thực.

Xoay đối tượng

Phép tịnh tiến đối tượng thật tuyệt, nhưng bạn vẫn chưa thực sự làm được những thứ mà game của bạn đòi hỏi. Sẽ còn gì là thú vị nếu như trong một trò đua xe bạn không thể lượn chiếc xe quanh một vòng cua … bởi lý do xe của bạn chỉ có thể di chuyển trên một đường thẳng? Đó là lý do mà bạn cần đến phép xoay. Nếu chiếc xe có khả năng xoay thì thì nó có thể ngoặt, và lượn theo các khúc cua.

Kết hợp giữa phép xoay và tịnh tiến các đối tượng 3D cho phép nhân vật của bạn có thể di chuyển tự do trong không gian game. Phép xoay cho phép những chiếc bánh xe ô tô có thể quay, cánh tay có thể cử động, quả bóng chày có thể xoay tròn khi bay…
Phép xoay là một quá trình quay một đối tượng xung quanh các trục tọa độ. Bởi vì phép xoay được thực hiện thông qua ma trận, nên thư viện D3DX đã cung cấp một vài hàm hữu ich để đơn giản hóa quá trình đó.
Phép xoay có thể thực hiện trên các trục vào bất kì thời điểm nào và trên bất kì trục nào trong 3 trục tọa độ. D3DX cung cấp cho mỗi trục tọa độ một hàm xoay. Ví dụ như, nếu bạn muốn xoay đỗi tượng quanh trục X, bạn có thể dùng hàm D3DXMatrixRotationX, như định nghĩa dưới đây:

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

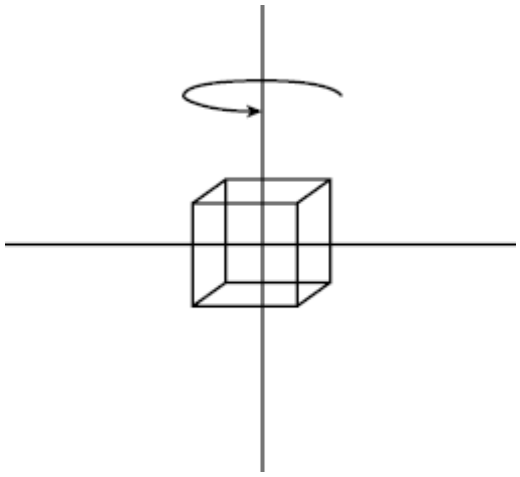
Hàm D3DXMatrixRotationX chỉ có 2 đối số:

- **pOut**. Con trỏ đến đối tượng kiểu D3DXMATRIX. Chứa ma trận trả về.
- **Angle**. Góc xoay đối tượng (dạng radian).

Sử dụng hàm D3DXMatrixRotationX hay bất kì gì có liên quan là rất đơn giản. Đầu tiên, ta định nghĩa một cấu trúc D3DXMATRIX chứa ma trận kết quả, sau đó đưa vào góc xoay là xong. Đoạn code sau cho thấy cách sử dụng hàm này:

```
D3DXMATRIX matRotate; // ma trận kết quả
D3DXMatrixRotationX(&matRotate, D3DXToRadian(45.0f));
```

Bạn định nghĩa một ma trận đầu ra và gọi hàm D3DXMatrixRotationX. Bạn có thể thấy, với tham số thứ hai ta đã sử dụng một lệnh trợ giúp là D3DXToRadian. Lệnh này nhận vào một góc trong khoảng 0 đến 360 độ và chuyển nó thành dạng radian. Trong ví dụ trên, góc xoay là 45 độ.
Kết quả của phép xoay là một đối tượng được xoay quanh trục X 45 độ.
Hình 5.6 biểu diễn một hình hộp được xoay quanh trục Y.
Đoạn code sau cho thấy những bước cần làm để xoay hình hộp quanh trục Y. Phép xoay sẽ thực hiện dựa trên bộ đếm thời gian và hình hộp sẽ được xoay liên tục.



Hình 5.6 Hình hộp được xoay quanh trục Y

```
/*****************************************************************************
* render
*****************************************************************************/
void render(void)
{
        // xóa back buffer về màu đen
        pd3dDevice->Clear( 0,
                        NULL,
                        D3DCLEAR_TARGET,
```

```
                    D3DCOLOR_XRGB(255,255,255),
                    1.0f,
                    0 );
        pd3dDevice->BeginScene();
        // đặt luồng vecto
        pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );
        // định dạng vecto
        pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
        // gán meshMat về ma trận đơn vị
        D3DXMatrixIdentity(&objMat);
        // cài đặt ma trận xoay
        D3DXMatrixRotationY(&matRotate, timeGetTime()/1000.0f);
        // nhân ma trận tỉ lệ và ma trận xoay để có ma trận objMat
        D3DXMatrixMultiply(&finalMat, &objMat, &matRotate);
        // thực hiện phép biến đổi trong không gian thực
        pd3dDevice->SetTransform(D3DTS_WORLD, &finalMat);
        // Render hình hộp bằng kiểu triangle strips
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 4, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 8, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );
        pd3dDevice->EndScene();
        // biểu diễn ra front buffer
        pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Có 3 biến được khai báo ở đầu của hàm render là objMat, matRotate, và finalMat. Những biến này là các ma trận lưu trữ thông tin về hình hộp. Phần trước bạn đã được học cách đưa một ma trận về dạng đồng nhất , và ở đây ma trận objMat cần được làm như vậy mỗi lần hàm render được gọi. Điều đó nhằm mục đích làm cho phép xoay luôn thực hiện ở gốc tọa độ. Như vậy objMat biểu diễn vị trí thực của hình hộp.

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

Ma trận thứ hai là matRotate, chứa thông tin về phép xoay hình hộp. Bởi vì hình hộp cần được xoay liên tục, do đó bạn cần cập nhật mới ma trận matRotate cho mỗi lần xoay với một vị trí mới của hình hộp. Phép xoay được thực hiện thông qua D3DXMatrixRotationY, là một hàm trong thư viện D3DX. Những hàm xoay trong thư viện D3DX sẽ viết đè thông tin trên ma trận qua mỗi lần xoay, vì thế mà bạn không cần gọi D3DXMatrixIdentity để đồng nhất hóa ma trận này.

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

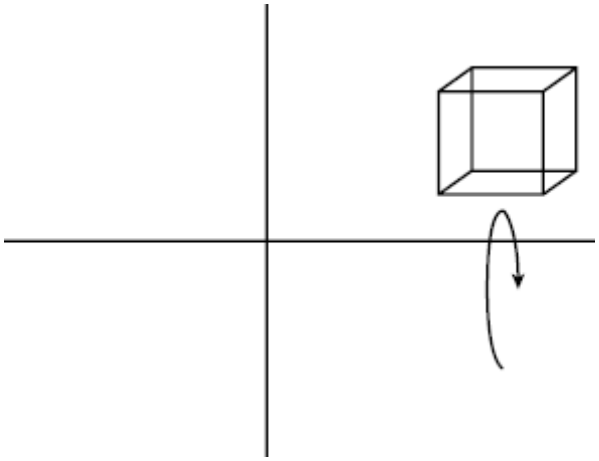
Hàm timeGetTime sử dụng thời gian hiện tại và chia cho 1000.0f để xoay hình hộp trơn chu hơn.
Tóm lại, bạn có hai ma trận, một cái biểu diễn vị trí của đối tượng và cái kia biểu diễn sự vận động của đối tượng, bạn cần nhân hai ma trận này với nhau để có được ma trận cuối cùng biểu diễn qua finalMat.
Ma trận kết quả quy đổi hình hộp về không gian thực qua hàm SetTransform dưới đây:

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

Kết quả trả về từ SetTransform là một hình hộp được đặt ở vị trí mới và định hướng trong không gian thực. Hàm render sẽ vẽ hình hộp qua các lời gọi DrawPrimitive.
Bạn có thể tìm thấy mã nguồn về các phép xoay đối tượng trong thư mục chapter5\example2 trên đĩa CD.

Tâm của phép xoay

Tâm của phép xoay một đối tượng phụ thuộc vào trục mà nó xoay quanh. Nếu một đối tượng, ví như hình hộp trong hình 5.6 đã xoay , tâm xoay của nó làm cho nó quay tròn xung quanh gốc tọa độ. Nếu một đối tượng được tịnh tiến ra xa gốc tọa độ và dọc theo một trục nào đó, thì tâm xoay của cũng dịch chuyển dọc trục đó làm cho đối tượng bị dịch chuyển sang vị trí khác trong quá trình thực hiện phép xoay.
Nhìn vào hinh 5.7, ta thấy một hình hộp được tịnh tiến dọc theo trục X và Y trước khi bị xoay. Khi hình hộp đó
được xoay quanh trục X, thì nó cũng bị tịnh tiến trong suốt quá trình xoay này.
Để thay đổi tâm phép xoay, bạn cần phải tịnh tiến đối tượng ra xa gốc tọa độ trước khi tiến hành phép xoay.



Hình 5.7: Hình hộp được xoay quanh trục X sau khi được tịnh tiến ra xa gốc tọa độ

Đoạn code sau cho thấy cách mà tâm xoay thay đổi khi tịnh tiến đối tượng.

```
/******************************************************************************
* render
******************************************************************************/
void render(void)
{
        // xóa back buffer về màu đen
        pd3dDevice->Clear( 0,
                           NULL,
                           D3DCLEAR_TARGET,
                           D3DCOLOR_XRGB(255,255,255),
                           1.0f,
                           0 );
        pd3dDevice->BeginScene();
        pd3dDevice->SetStreamSource( 0, vertexBuffer, 0, sizeof(CUSTOMVERTEX) );
        pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
        // tịnh tiến đối tượng ra xa gốc tọa độ
        D3DXMatrixTranslation(&matTranslate, 64.0f, 0.0f, 0.0f);
        // cài đặt phép quay
        D3DXMatrixRotationY(&matRotate, timeGetTime()/1000.0f);
        // Nhân ma trận tịnh tiến và ma trận xoay để có ma trận objMat
        D3DXMatrixMultiply(&objMat, &matTranslate, &matRotate);
        // thực hiện phép biến đổi trong không gian thực
        pd3dDevice->SetTransform(D3DTS_WORLD, &objMat);
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 4, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 8, 2 );
        pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 12, 2 );
```

```
            pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 16, 2 );
            pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 20, 2 );
            pd3dDevice->EndScene();
            // đưa kết quả ra front buffer
            pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```

Sự thay đổi lớn nhất trong hàm render là hàm D3DXMatrixTranslation. Hàm này dịch chuyển hình hộp ra xa gốc tọa độ 64 đơn vị.
Trong trường hợp này, hình hộp sẽ được tịnh tiến ra xa gôc tọa độ dọc theo trục X và sau đó mới xoay. Hai ma trận được dùng ở đây là : matTranslate và matRotate. Hai ma trận này được nhân với nhau để có ma trận objMat, chứa vị trí sau cùng của hình hộp. Kết quả là ta có một hình hộp được xoay ở cách xa gốc tọa độ

Phép tỉ lệ

Phép tỉ lệ cho phép bạn thay đổi kích thước đối tượng bằng cách nhân các vecto của đối tượng với một lượng nào đó. Để thực hiện phép tỉ lệ trên đối tượng, bạn cần tạo một ma trận chứa các hệ số tỉ lệ. Những hệ số này cho biết các vecto sẽ được phóng to hay thu nhỏ bao nhiêu. Như đã đề cập ở trên, các vị trí 1, 6, 11 là các hệ số tỉ lệ theo các phương X, Y và Z. Mặc định, các số đó là 1.0f nghĩa là các đối tượng có kích thước như nó vốn có. Thay đổi bất kì hệ số nào sẽ làm thay đổi kích thước của đối tượng. Nếu các hệ số này lớn hơn 1.0f thì đối tượng sẽ được phóng to ra, ngược lại, nếu hệ số này nhỏ hơn 1.0f thì đối tượng sẽ bị thu nhỏ lại.

$$\begin{bmatrix} X & 2 & 4 & 4 \\ 5 & Y & 7 & 8 \\ 9 & 10 & Z & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Như đã đề cập ở trên, phép tỉ lệ được điều khiển thông qua các giá trị trong ma trận. Để tạo một ma trận tỉ lệ, ta chỉ cần định nghĩa một ma trận đồng nhất và thay đổi các hệ số như đã nói ở trên. Bạn vừa có thể tự mình thay đổi các giá trị này vừa có thể sử dụng thông qua hàm D3DXMatrixScaling như định nghĩa dưới đây:

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

Hàm D3DXMatrixScaling cần 4 đối số:

- **pOut**. Con trỏ đến đối tượng D3DXMATRIX chứa ma trận tỉ lệ
- **sx**. Hệ số tỉ lệ theo phương X
- **sy**. Hệ số tỉ lệ theo phương Y
- **sz**. Hệ số tỉ lệ theo phương Z

Đoạn code sau cho thấy cách dùng hàm D3DXMatrixScaling để tăng gấp đôi kích thước một vật thể.

```
D3DXMATRIX matScale;
// đặt hệ số tỉ lệ
```

```
D3DXMatrixScaling(&matScale, 2.0f, 2.0f, 2.0f);
// Nhân ma trận đối tượng với ma trận tỉ lệ
D3DXMatrixMultiply(&objMat, & objMat, &matScaling);
```

Biến objMat ở trên biểu diễn ma trận gốc của đối tượng. nhân ma trận của đối tượng với ma trận tỉ lệ ta sẽ có thể thu phóng đối tượng khi vẽ.

Thứ tự trong các phép tính toán ma trận

Thứ tự trong các phép toán là rất quan trọng. Ví dụ như, nếu bạn muốn xoay một đối tượng quanh tâm của nó và sau đó dịch chuyển nó đi đâu đó, thì trước tiên bạn cần tính toán với ma trận xoay trước tiếp đó là đến ma trận tịnh tiến. Nếu hai phép tính này được đổi chỗ cho nhau, thì đối tượng sẽ được tịnh tiến đến 1 vị trí khác sau đó mới được xoay quanh gốc tọa độ. Điều này có thể làm cho đối tượng được đặt ở một ví trí khác và hướng theo một hướng khác trong không gian. Đoạn code sau chỉ ra trình tự đúng để thực hiện phép xoay và tịnh tiến đối tượng:

```
D3DXMATRIX objRotate;
D3DXMATRIX objTranslation;
D3DXMATRIX objFinal;
// phép xoay
D3DXMatrixRotationY(&objRotate, D3DXToRadian(45));
// phép tịnh tiến
D3DXMatrixTranslation(&objTranslation, 1.0f, 0.0f, 0.0f);
// nhân ma trận xoay và ma trận tịnh tiến với nhau
D3DXMatrixMultiply(&objFinal, &objRotate, &objTranslation);
// thực hiện phép biến đổi trong không gian thực
pd3dDevice->SetTransform(D3DTS_WORLD, &objFinal);
```

Bước thứ nhất là tạo ma trận xoay đối tượng, objRatate. Sử dụng hàm D3DXMatrixRotationY như trên, đối tượng sẽ được quay một góc 45 độ quanh trục Y.
Tiếp theo ta tịnh tiến đối tượng đã xoay một đơn vị về bên phải bằng cách sử dụng hàm D3DXMatrixTranslation.
Cuối cùng, ta tạo ra ma trận biến đổi bằng cách nhân ma trận xoay và ma trận tịnh tiến với nhau với hàm D3DXMatrixMultiply. Nếu ma trận xoay và ma trận được đảo chỗ cho nhau (hoán vị) trong lời gọi hàm D3DXMatrixMultiply, thì phép tịnh tiến sẽ được thực hiện trước phép xoay, và đối tượng bị rời sang vị trí khác.

Tạo một camera bằng các phép chiếu

Bạn có thể tạo ra một camera trong Direct3D bằng cách định nghĩa một ma trận cho bước (projection transformation). Ma trận này định nghĩa một vùng nhìn cho camera, hệ số co, mặt phẳng clipping xa và gần.
Sau khi bạn đã tạo được ma trận chiếu, bạn thiết lập nó thông qua hàm SetTransform. Bạn có thể thấy rằng, hàm SetTransform đã được sử dụng ở ví dụ trước. SetTransform, định nghĩa ở dưới đây, cho phép thiết lập ma trận cho các bước của hệ thống chuyển đổi hình học. Ví dụ như, khi bạn thiết lập ma trận cho một camera, cũng có nghĩa là bạn đang quy định cho scene sẽ được quan sát ra sao trong suốt giai đoạn chiếu. Giai đoạn này cũng là giai đoạn cuối cùng của hệ chuyển đổi hình học, nó quy định cách mà khung cảnh 3D sẽ được render dưới dạng 2D.

```
HRESULT SetTransform(
```

```
        D3DTRANSFORMSTATETYPE State,
        CONST D3DMATRIX *pMatrix
);
```

Hàm SetTransform cần 2 đối số:

- **State**. Xác định giai đoạn của hệ chuyển đổi hình học cần chỉnh sửa.
- **pMatrix**. Con trỏ có cấu trúc D3DMATRIX được dùng để thiết lập cho giai đoạn trên.

Đoạn code sau cho thấy cách tạo và định nghĩa một ma trận cho giai đoạn chiếu.

```
D3DXMATRIX matProj; // ma trận chiếu
/*****************************************************************************
* createCamera
* tạo camera ảo
*****************************************************************************/
void createCamera(float nearClip, float farClip)
{
        // Xác định vùng nhìn, hệ số co và mặt phẳng clipping xa, gần
        D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, 640/480, nearClip, farClip);
        // thiết lập ma trận matProj cho giai đoạn chiếu
        pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj);
}
```

Thay vì tự mình tạo một ma trận chiếu, ta đã dùng hàm D3DXMatrixPerspectiveFovLH do D3DX cung cấp. Hàm này tạo một ma trận đầu ra chứa trong matProj được khai báo ở trên, nó cho phép bạn xác định góc nhìn phối cảnh, hệ số co, và mặt phẳng clipping chỉ trong một lời gọi hàm.
Sau khi bạn đã tạo được ma trận chiếu, bạn có thể thiết lập nó cho hệ chuyển đổi hình học thông qua hàm SetTransform. Bởi vì ma trận này tác dụng lên giai đoạn chiếu, cho nên tham số đưa vào là D3DTS_PROJECTION.

Vị trí và hướng của camera

Đến thời điểm này, bạn đã có thể sử dụng được camera rồi. Camera tác động lên mọi thứ trong khung cảnh cũng giống như là các đối tượng được chuyển qua giai đoạn chiếu trong hệ chuyển đổi hình học vậy. Chỉ có một điều là, camera của ta đang được đặt ở gốc tọa độ. Bởi vì camera trong thế giới thực là một vật thể có thể chuyển động được, nên ta cũng cần làm cho camera ảo có thể làm được giống như vậy. Nó cần có khả năng chuyển động trong khung cảnh và cũng có thể thay đổi hướng nhìn. Để đạt được 2 tiêu chí này, bạn cần thay đổi ma trận tương ứng với giai đoạn View transformation. Mặc định, ma trận này được đặt là ma trận đồng nhất cố định camera ảo ở gốc tọa độ. Để thay đổi vị trí và hướng của camera, bạn cần tạo một ma trận mới. Cách đơn giản nhất để làm điều đó là sử dụng hàm trợ giúp D3DXMatrixLookAtLH của D3DX.
Hàm D3DXMatrixLookAtLH cho phép bạn chỉ ra vị trí của camera (định nghĩa như một vecto dạng D3DXVECTOR3), nơi mà camera nhìn vào (cũng dạng D3DXVECTOR3), và hướng của camera (cũng là dạng D3DXVECTOR3).
Đoạn code sau chỉ ra cách tạo ma trận quan sát (view).

```
D3DXMATRIX matView; // ma trận view
/*****************************************************************************
* pointCamera
* đặt điểm nhìn cho camera thông qua 1 vecto
*****************************************************************************/
```

```
void pointCamera(D3DXVECTOR3 cameraPosition, D3DXVECTOR3 cameraLook)
{
        D3DXMatrixLookAtLH ( &matView,
                             &cameraPosition, // vị trí camera
                             &cameraLook, // điểm nhìn
                             &D3DXVECTOR3 (0.0f, 1.0f, 0.0f)); // hướng lên trên
        // thiết lập ma trận này cho giai đoạn view
        pd3dDevice->SetTransform (D3DTS_VIEW, &matView);
}
```

Hàm pointCamera cần hai đối số: cameraLook và cameraPosition.
Biến cameraPosition chứa vị trí hiện tại của camera. Ví dụ như, nếu camera được đặt ở cách gốc tọa độ 2 đơn vị dọc theo trục Y, cameraPosition sẽ có dạng (0.0f, -2.0f, 0.0f).
Biến cameraLook thông báo cho camera nơi cần hướng tới và nó có quan hệ với vị trí đặt camera. Ví dụ như, nếu coi camera đặt ở vị trí10 đơn vị dương dọc theo trục Y,10 đơn vị âm theo trục Z và tưởng tượng rằng ta muốn camera hướng về phía gốc tọa độ. Bởi vì lúc này camera đã được đặt ở phía trên so với gốc tọa độ, do đó muốn thấy được gốc tọa độ nó cần phải nhìn xuống dưới. Ta cần thiết lập cho vecto cameraLook giá trị là (0.0f, -10.0f; 0.0f), điều này có nghĩa là camera hướng thẳng theo trục Y và hướng xuống dưới. Camera lúc đó có thể nhìn các thấy đối tượng đặt ở gốc tọa độ và từ phía trên đầu của chúng.
Ma trận cuối cùng mà D3DXMatrixLookAtLH tạo ra được lưu trữ trong matView và được thiết lập cho giai đoạn view của hệ chuyển đổi. Giá trị D3DTS_VIEW được truyền cho hàm SetTransform thông báo cho Direct3D rằng ma trận view cần được cập nhật lại.

Tổng kết chương

Trong chương này, ta đã giới thiệu những khái niệm chung cần thiết khi xây dựng một ứng dụng 3D. Khi bạn muốn tạo một chương trình quan sát mô hình, hay một game nhập vai góc quay thứ nhất, ma trận và các phép biến đổi chính là nền móng mà game của bạn được xây dựng trên đó.

Những gì bạn đã được học

Trong chương này, bạn đã được học:

- Cách chuyển các đối tượng 3D qua hệ chuyển đổi hình học.
- Ma trận là gì khi nào và làm thế nào sử dụng chúng
- Cách dịch chuyển và xoay các đối tượng.
- Tại sao thứ tự trong phép nhân ma trận lại quan trọng đến vậy.
- Cách tạo và sử dụng camera để quan sát các đối tượng 3D.

Câu hỏi kiểm tra

Bạn có thể tìm thấy đáp án cho phần này và phần bài tập tự làm trong phụ lục "Đáp án phần bài tập cuối chương".

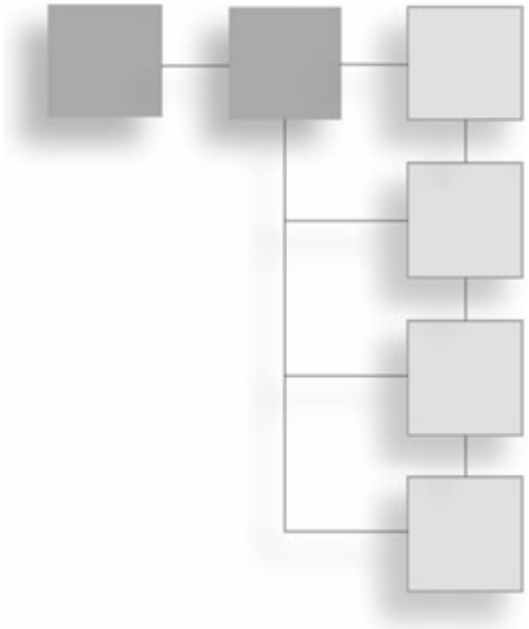
1. Chỉ mục của đối tượng được lưu trữ trong bộ đệm (buffer) nào ?
2. Ma trận là gi?
3. Những bước trong hệ chuyển đổi hình học?
4. Ma trận đồng nhất dùng để làm gì?
5. Thay đổi hệ số co của camera tác động đến phần nào của hệ chuyển đổi hình học?

Bài tập tự làm

1. Sử dụng hàm D3DXMatrixMultiply, để xoay đối tượng và sau đó tịnh tiến nó 5 đơn vị theo trục X.
2. Viết một hàm render xoay liên tục một đối tượng quanh trục Y.

(Chuong 6...)

CHƯƠNG 7



CHIA NHỎ VÀ LÀM MỊN CÁC ĐỐI TƯỢNG

Bạn đã học được cách làm thế nào để tạo các object 3D bằng code và biểu diễn nó lên màn hình. Có lẽ bạn đang nghĩ rằng đây là một quá trình thật nhàm chán và chưa có cách nào để có thể tạo tất cả các object bằng code. Bạn nghĩ rất đúng. Hiện nay 3D đã nhập cuộc. Nó có thể mô tả mọi thứ trong game của bạn rất giống với thực thể. Những mô hìnhcó thể thể hiện vật thể và đặc điểm xung quanh bạn và cũng có thể là chính chính nó. Với những đặc điểm này bạn có thể đưa những mô hình này vào game, bạn có thể biểu diễn nó với đối tượng Mesh và dịch chuyển hoặc điều khiển chúng.

Đây là các phần mà bạn sẽ học trong chương này:

- Direct3D điều khiển mesh như thế nào ?
- Cần những gì để hợp lý hóa một model 3D?
- Định dạng file X là gì?
- Làm thế nào để tạo và lưu giữ mesh?
- Làm thế nào để load một model 3D vào game?

Xây dựng một thế giới 3D

Mô hình 3D giúp chúng bạn thể hiện thế giới ảo mà bạn muốn tạo. Những mô hình này được bổ xung bởi gamer và đối thủ của gamer trong môi trường này. Những mô hình này được lấy từ đâu? Nếu bạn có một package thiết kế 3D giống như Max hoặc Maya, bạn đã có những công cụ cần thiết để tạo mọi thứ cho game của bạn khi cần. Nếu những chương trình trên bạn không có thì bạn có thể dùng những package khác như MilkShape 3D, nó cũng có thể làm việc tốt.

Sau khi đã tạo các model, bạn đưa chúng vào một trong những định dạng file 3D hiện có. Nhưng bạn cần biết làm thế nào để load một file định dạng 3D vào game của mình. Mục đích của cuốn sách này là giúp bạn làm việc với những định dạng file mà Microsoft đã tạo ra.

Chú ý:

Bạn có thể tìm MilkShape 3D tại trang http://www.swissquake.ch/chumbalum-soft/index.html.

Mesh là gì?

Code của bạn điều khiển những mô hình 3D được load vào trong game cũng được xem như là một Mesh. Một Mesh là một code container mà chứa mọi thứ liên quan đến đối tượng 3D. Nó bào gồm các vectơ, tọa độ texture và các dữ liệu khác. Bằng cách sử dụng dữ liệu có trong mesh object bạn có thể biểu diễn các mô hình 3D lên màn hình .

Chú ý:

Thư viện tổng hợp D3DX chứa tất cả mọi thứ mà bạn cần để sử dụng mesh trong Direct3D

Direct3D định nghĩa một Mesh như thế nào?

Hầu hết các mesh trong Direct3D đều dựa trên ID3DXBaseMesh interface. Interface này cung cấp kho dự trữ dành cho các model của bạn, giúp các methods có hiệu lực để tăng tốc độ truy cập tới dữ liệu trong mesh. Ví dụ method GetVertexBuffer luôn sẵn có trong ID3DXBaseMesh interface để giúp bạn truy cập trực tiếp tới vectơ buffer của đối tượng mesh.
Dưới đây là một số kiểu Mesh khác nhau:

- ID3DXMesh – Đây là dạng mesh interface chuẩn mà bạn sẽ sử dụng.
- ID3DXPMesh – Interface này cho phép bạn sử dụng mesh tiến hành.
- ID3DXSPMesh – interface này điều khiển sự đơn giản hóa các mesh object.
- ID3DXPatchMesh - Interface này cung cấp Patch mesh theo chức năng.

Mỗi một kiểu mesh có thể lưu giữ tất cả các vectơ của một model trong vectơ buffer và cung cấp cho bạn thông tin về model, ví dụ như số phần tử hoặc là số vectơ.

Tạo Mesh

Bước đầu tiên khi sử dụng các mesh trong game đó là sự khởi tạo mesh object. Mesh object là kho dự trữ, cất giữ tất cả các thông tin cần thiết dùn để mô tả model của bạn trong Direct3D. Sau khi bạn đã tạo ra mesh, bạn dễ dàng copy tất cả thông tin mà model của bạn cần.
Hai hàm trong Direct3D dùng để tạo mesh: D3DXCreaetMesh và D3DXCreateMeshFVF. Vì mỗi hàm này tạo mesh bằng các cách khác nhau, chúng ta sẽ làm rõ cả hai hàm dưới đây.

D3DXCreateMesh

Giao diện ID3DXMesh là một giao diện đơn giản nhất trong mesh interface, dễ dàng tổ chức và thực hiện một cách nhanh chóng. Trong phần này, bạn sẽ học cách làm thế nào để tạo mesh trong ID3DXMesh interface bằng hàm D3DXCreateMesh.

```
D3DXCreateMesh(
        DWORD NumFaces,
        DWORD NumVertices,
        DWORD Options,
        CONST   LPD3DVERTEXELEMENT9 *pDeclaration,
        LPDIRECT3DDEVICE9  pDevice,
        LPD3DXMESH  *ppMesh
);
```

Hàm D3DcreateMesh có 6 tham số:

- **NumFaces**. Số phần tử mà mesh sẽ chứa.
- **NumVertices**. Số vectơ mà mesh sẽ chứa.
- **Options**. Giá trị từ bảng liệt kê D3DXMESH .
- **pDeclaration**. Ma trận thuộc đối tượng D3DVERTEXELEMENT9. Những object này mô tả FVF cho mesh.
- **pDevice**. Một Direct3D device hợp lệ.
- **ppMesh**. Con trỏ trỏ tới đối tượng ID3DMesh hợp lệ.

Đoạn code sau sẽ chỉ ra cách làm thế nào để tạo một đối tượng mesh mà chứa đầy đủ bộ vectơ, dùng để tạo một khối lập phương.

```
HRESULT hr;
// biến giữ một mesh được tạo mới
LPD3DXMESH boxMesh;
// ma trận D3DVERTEXELEMENT9
D3DVERTEXELEMENT9 Declaration [MAX_FVF_DECL_SIZE];
// tạo  khai báo cần thiết cho hàm D3DXCreateMesh
D3DXDeclaratorFromFVF (D3DFVF_CUSTOMVERTEX, Declaration);
hr= D3DXCreateMesh(12,                          //số phần tử của mesh
                   8,                           //số vectơ
                   D3DXMESH_MANAGED,            //sử dụng bộ nhớ cần thiết cho mesh
                   Declaration,         //ma trận kiểu đối tượng D3DVERTEXELEMENT9
                   pd3dDevice,                  //the Direct3D device
                   &boxMesh);                   //biến giữ mesh

//kiểm tra  giá trị trả về để chắc chắn rằng bạn đã cómột đối tượng mesh hợp lệ.
if FAILD (hr)
        Return false;
```

Như bạn đã thấy, đoạn code trên tạo ra một mesh chứa 12 phần tử và 8 vectơ và tất cả chúng được đưa vào trong biến boxMesh. Biến thứ ba là D3DXMESH_MANAGED thông báo Direct3D là cần sử dụng bộ nhớ cần thiết cho cả vectơ và index buffer để tạo mesh.
Bạn nên chú ý là hàm D3DXDeclaratorFromFVF phải được gọi trước hàm D3DXCreateMesh. D3DXDeclaratorFromFVF tạo đối tượng cần thiết D3DVERTEXELEMENT9 cho tham số thứ 4 bằng cách sử dụng Flexible Vertex Format (định dạng véctơ linh hoạt) mà model của bạn sử dụng.
Khi bạn sử dụng hàm D3DXDeclaratorFromFVF, bạn không cần thiết phải tạo ngay các đối tượng D3DVERTEXELEMENT9.

D3DXCreatMeshFVF

Hàm D3DXCreateMeshFVF không giống với D3DcreateMesh ở một điểm là nó dựa vào sự kiến tạo mesh trên Định Dạng Véctơ Linh Hoạt (Flexible Vertex Format) thay vì dùng Declarator. Mesh object này cũng giống với mesh object được tạo bởi hàm D3DXCreateMesh ở trong phần trước.
Hàm D3DXCreatMeshFVF được định nghĩa như sau:

```
HRESULT D3DXCreateMeshFVF(
        DWORD Numface,
        DWORD NumVertices,
        DWORD Options,
        DWORD FVF,
```

```
        LPDIRECT3DDEVICE9 pDevice,
        LPD3DXMESH *ppMesh
);
```

Hàm D3DXCreatMeshFVF cần 6 tham số:

- **Numface** - số phần tử mà mesh sẽ có
- **NumVertices** – số véctơ mà mà mesh sẽ có
- **Options** – các giá trị từ bảng liệt kê D3DXMESH
- **FVF** – Flexible Vertex Format của các véctơ.
- **pDevice** – Direct3D device hợp lệ.
- **ppMesh** – con trỏ trỏ tới đối tượng ID3DXMESH

Dưới đây là một chương trình mẫu gọi hàm D3DXCreateMeshFVF.

```
//biến giữ giá trị trả về
HRESULT hr;
//biến giữ mesh đựơc tạo mới
LPD3DXMESH boxMesh;
//tạo mesh bằng cách gọi gàm D3DXCreateMeshFVF
hr= D3DXCreateMeshFVF (12,                          //Numfaces
                       8,                           // NumVertices
                       D3DFVF_MANAGED,              // Options
                       D3DFVF_CUSTOMVERTEX,         // FVF
                       pd3dDevice,                  // pDevice
                       &boxMesh);                   // ppMesh
//kiểm tra giá trị trả về có hợp lệ không
if FAILED (hr)
        return false;
```

Một lần nữa bạn tạo mesh bằng cách sử dụng bộ nhớ quản lý cho véctơ, index buffer và việc chỉ định giá trị D3DXMESH_MANAGED. Khi việc gọi hàm kết thúc, biến boxMessh sẽ giữ đối tượng mesh hợp lệ.
Hàm D3DXCreateMeshFVF là hàm dễ sủ dụng nhất trong hai hàm tạo mesh.

Filling the Mesh.

Bạn đã tạo được mesh object, bạn cần bổ xung thêm dữ liệu để mô tả model mà bạn muốn thể hiện. Ở đây, bạn có một kho rỗng với kích thước thích hợp để chứa dữ liệu cần thiết cho việc tạo khối lập phương.
Để xác định một khối lập phương, trước tiên bạn cần lock véctơ buffer lại và bổ xung vào nó 8 véctơ mà khối lập phương cần. Tiếp theo bạn cần lock index buffer và copy toàn bộ index vào trong đó.
Hàm SetupMesh chỉ ra dưới đây sẽ giúp bạn từng bước hoàn thành mesh với các thông tin cần thiết để tạo một khối lập phương.

```
/*******************************************************************************
* SetupMesh
* Set up the vertex buffer and index buffer of a mesh
*******************************************************************************/
HRESULT SetupMesh()
{
        HRESULT hr;                     //biến giữ giá trị trả về
        ////////////////////////////////////////////////////////////////////////
        //các véctơ của vertex buffer
        CUSTOMVERTEX g_Vertices[ ]={
```

```
                // X        Y        Z
                {-1.0f, -1.0f, -1.0f, D3DCOLOR_ARGB(0,255,0,0)},        //0
                        {-1.0f,  1.0f, -1.0f, D3DCOLOR_ARGB(0,0,0,255)},        //1
                { 1.0f,  1.0f, -1.0f, D3DCOLOR_ARGB(0,0,255,0)},        //2
                { 1.0f, -1.0f, -1.0f, D3DCOLOR_ARGB(0,0,0,255)},        //3
                {-1.0f, -1.0f,  1.0f, D3DCOLOR_ARGB(0,0,255,0)},        //4
                { 1.0f, -1.0f,  1.0f, D3DCOLOR_ARGB(0,0,0,255)},        //5
                { 1.0f,  1.0f,  1.0f, D3DCOLOR_ARGB(0,0,255,0)},        //6
                {-1.0f,  1.0f,  1.0f, D3DCOLOR_ARGB(0,0,0,255)}                 //7
        };
        //Copy các véctơ vào trong vertex buffer
        VOID* pVertices;
        // lock vertex buffer
        hr = boxMesh->LockVertexBuffer(D3DLOCK_DISCARD, (void**)&pVertices);
        //kiểm tra lại để chắc chắn là các vertex buffer đã lock
        if FAILED (hr)
                return hr;
        ////////////////////////////////////////////////////////////////////////////
        //index buffer data
        //index buffer xác định các phần tử của khối lập phương,
        //hai phần tử ở mỗi mặt của cube
        WORD IndexData[ ] = {
                0,1,2,          //0
                2,3,0,          //1
                4,5,6,          //2
                6,7,4,          //3
                0,3,5,          //4
                5,4,0.          //5
                3,2,6,          //6
                6,5,3,          //7
                2,1,7,          //8
                7,6,2,          //9
                1,0,4,          //10
                4,7,1,          //11
        }
        //copy các véctơ vào buffer
        memcpy(pVertices, g_Vertices, sizeof(g_Vertices) );
        //mở khóa véctơ buffer
        boxMesh->UnlockVertexBuffer();
        //chuẩn bị copy các index vào index buffer
        VOID* IndexPtr;
        //khóa index buffer
        hr-boxMesh->LockIndexBufffer( 0, &IndexPtr);
        //kiểm tra xem index buffer đã khóa chưa
        if FAILED (hr)
                return hr;
        //copy các index vào buffer
        memcpy(IndexPtr, IndexData, sizeof( IndexData)*sizeof(WORD));
        //mở khóa buffer
        boxMesh->UnlockIndexBuffer();
        return S_OK;
}
```

Điều đầu tiên mà hàm SetupMesh làm đó là tạo ma trận g_Vertices. Ma trận này chứa các véctơ và véctơ màu mà bạn cần để xác định một khối lập phương. Tiếp đó, vectơ buffer bị locked như đã thấy ở ví dụ trên. Việc gọi hàm memcpy là để copy tất cả các véctơ vào trong vertex buffer và sau đó thực hiện unlock buffer.

Tiếp theo bạn cần điền vào index buffer. Giống như vectơ buffer, bạn phải lock nó trước khi dữ liệu được copy vào nó. Các index mà index buffer sử dụng sẽ được xác định trong ma trận IndexData. Chú ý rằng ma trận IndexData có kiểu WORD, điều đó có nghĩa là chúng là những value 16-bit.

Sau khi bạn đã có các giá trị xác định, bạn lock index buffer và copy vào các index này bằng việc gọi hàm memcpy rồi unlock index buffer.

Chú ý:

Cả vectơ và index buffer đều cần thiết để tạo một đối tượng mesh hợp lệ

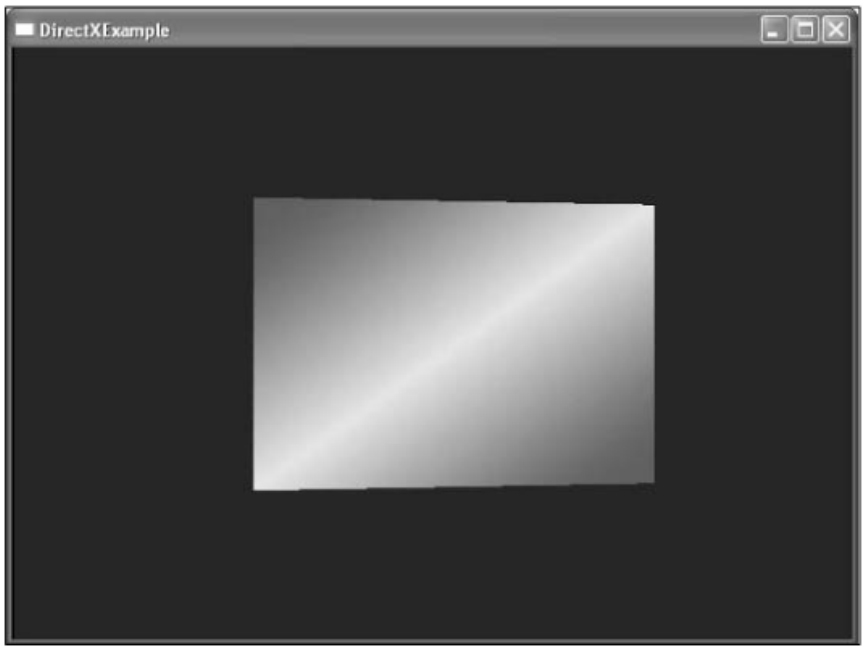
Hiển thị Mesh

Bạn đã tạo ra một mesh và đưa dữ liệu vào trong đó, như vậy bạn đã sẵn sàng đưa nó ra màn hình. Hàm drawMesh dưới đây sẽ chỉ ra việc biểu diễn một mesh sẽ cần những gì. Hàm này sẽ làm khối lập phương quay trên màn hình.

```
/*******************************************************************************
*void drawMesh (LPD3DXMESH mesh, D3DMATERIAL9 material)
* Draws the mesh
*******************************************************************************/
void drawMesh (LPD3DXMESH mesh, D3DMATERIAL9 *marterial)
{
//quay mesh
D3DXMATRIX matRot;
D3DXMATRIX matView;
D3DXMATRIX matWorld;
//tạo ma ma trận quay
D3DXMatrixMultiply(&matWorld, &matRot, &matView);
//lấy sự biến đổi vào kết quả của ma trận matWorld
pd3dDecice ->SetTransform( D3DTS_WORLDM, &matWorld);
//đưa giữ liệu vào sử dụng
pd3dDevice->SetMaterial(material);
//vẽ mesh
mesh->DrawSubset(0);
}
```

Phần đầu tiên của hàm drawMesh sẽ là quay và dịch chuyển khối lập phương trên màn hình. Sau đó giữ liệu mà khối lập phương sẽ sử dụng được nạp vào qua việc gọi SetMaterial. Phần quan trọng nhất của hàm drawMesh là gọi DrawSubset.
Hàm DrawSubset thông báo Direct3D phần nào của mesh bạn muốn hiện thị lên màn hình. Vì mesh mà bạn tạo ra ở phần trước chỉ chứa một nhóm đơn thuần, giá trị 0 đã được truyền cho DrawSubset.

Trên hình 7.1 sẽ chỉ ra kết quả mesh được hiển thị trên màn hình. Bạn sẽ tìm thấy đầy đủ source code trong chapter7\example1 trên đia CD-ROM.



Hình 7.1 hình ảnh mesh của cube trên màn hình.

Chú ý:

Mesh có thể chứa rất nhiều nhóm đặc tính khác. Những nhóm khác đó lưu giữ một tập hợp tất cả các phần tử được dùng để mô tả sự phân chia của mesh. Ví dụ như nếu bạn có 2 vùng mesh mà cần 2 texture khác nhau, mỗi vùng trong đó sẽ đặt vào một nhóm đặc tính riêng rẽ. Bằng cách này, bạn có thể điều chỉnh lại texture và giữ liệu mà Dicrect3D sử dụng trước khi kết xuất mỗi nhóm

Tối ưu hóa Mesh.

Khi một mesh được tạo, nó không có định dạng tốt ưu để Direct3D vẽ. Ví dụ mesh của bạn phải chứa các véctơ bị lặp lại và được sử dụng cho nhiều phần tử, hoặc các véctơ và các phần tử không được nằm trong một thứ tự có hiệu quả. Khi tối ưu hóa mesh bằng cách sử dụng các "véctơ được dùng chung" và cho phép các véctơ và các phần tử sắp xếp lại, bạn có thể tăng quá trình thực hiện khi kết xuất một mesh.

Thư viện tổng hợp D3DX cung cấp 2 hàm để tối ưu mesh: Optimize và OptimizeInplace. Mỗi một hàm này về cơ bản thực hiện cùng một công việc nhưng với các key khác nhau. Optimize tạo ra output mesh, ngược lại OptimizeInplace làm thay đổi ở input mesh. Tôi sẽ giải thich cụ thể hai hàm này được sử dụng với mesh như thế nào.
Hàm Optimize định nghĩa nhu sau, lấy một input mesh, tối ưu nó và khởi tạo một output mesh.

```
HRESULT  Optimize(
        DWORD Flags,
        CONST DWORD *pAdjacencyIn,
        DWORD *pAdjacencyOut,
        DWORD *pFaceRemap,
        LPD3DXBUFFER *ppVerTexRemap,
        LPD3DXMESH *ppOptMesh
);
```

Hàm Optimize có 6 tham số:

- **Flags** – là dấu hiệu chỉ ra dạng tối ưu cho việc thi hành. Bạn càn tìm flag cho tham số này trong bảng liệt kê D3DXMESHOPT.

- **pAdjacencyIn** – con trỏ trỏ tới ma trận đang lưu dữ liệu liền kề hiện tại cho input mesh.
- **pAdjacencyOut** – con trỏ trỏ tới ma trận đang lưu giữ dữ liệu liền kề cho output mesh được tối ưu.
- **pFaceRemap** – con trỏ trỏ tới buffer đang lưu dữ index mới cho output mesh.
- **ppVertexRemap** – địa chỉ gửi đến con trỏ của giao diện *ID3DXBuffer* dành cho output mesh.
- **ppOptmesh** – một giao diện ID3DXMesh đang lưu giữ output mesh được tạo mới.

Hàm OptimizeInplace làm thay đổi input mesh, được xác định như sau:

```
HRESULT  Optimize(
        DWORD Flags,
        CONST DWORD *pAdjacencyIn,
        DWORD *pAdjacencyOut,
        DWORD *pFaceRemap,
        LPD3DXBUFFER *ppVerTexRemap,
);
```

Hàm OptimizeInplace có 5 tham số:

- **Flags** – là dấu hiệu chỉ ra dạng tối ưu để thi hành. Bạn có thể tìm thấy dấu hiệu cho tham số này ở trong bảng liệt kê D3DXMESHOPT.
- **pAdjacencyIn** – con trỏ trỏ tới ma trận đang lưu giữ dữ liệu liền kề hiện tại cho mesh
- **pAdjacencyOut** – con trỏ trỏ tới buffer đang lưu giữ dũ liệu liền kề cho mesh được tối ưu. Nếu bạn không muốn tập trung dữ liệu liền kề, bạn có thể chuyển giá trị NULL cho tham số này.
- **pFaceRemap** – con trỏ trỏ tới buffer đang lưu giũ index mới của dũ liệu mỗi phần tử. Nếu bạn không muốn chọn các thông tin này thì bạn có thể gán NULL cho tham số này.
- **ppVerTexRemap** – con trỏ trỏ tới giao diện ID3DXBuffer dùng để lưu giũ index mới của mỗi véctơ.

Bảng 7.1 Các tham số Flags

Giá trị	Mô tả
D3DXMESHOPT_COMPACT	Sắp xếp lại các bề phần tử để bỏ đi các véctơ và các phần tử không sử dụng
D3DXMESHOPT_ATTRSORT	Sắp xếp lại các bề phần tử để giảm số trạng thái giũ liệu thay đổi
D3DXMESHOPT_VERTEXCACH	Sắp xếp lại các bề phần tử để giúp đưa ra hình vẽ
D3DXMESHOPT_STRIPREORDER	Sắp xếp các bề phần tử để phóng to chiều dài của hình tam giác gần kề.
D3DXMESHOPT_IGNOREVERTS	Chỉ giúp các bề phần tử được tối ưu, còn các véctơ thì không.
D3DXMESHOPT_DONOTSPLIT	Ngăn ngừa sự cắt nhỏ cảu các véctơ được chia sẻ trong nhóm.
D3DXMESHOPT_DIVICEINDEPENDENT	Giúp véctơ cất giữ kích thước vào một tập hợp mà kích thước đó làm việc tốt trong phần cứng đươc thừa kế.

Getting the Mesh Details

Trong suốt quá trình tối ưu hóa mesh, bạn đang tò mò muốn biết các details của mesh mà bạn đang làm việc có những gì. Ví dụ bạn phải tự hỏi mình có bao nhiêu vécto hoặc bao nhiêu bề phần tử mà mesh chứa trước khi tối ưu hóa. Giao diện ID3DXMesh cung cấp 2 hàm có ích cho mục đích này:

- GetNumVertices – Trả về số véctơ có trong mesh
- GetNumfaces – Trả về số phần tử có trong mesh.

Đoạn chương trình sau là một mẫu mà chỉ ra cách làm thế nào để sử dụng những hàm này và biểu diễn một cửa sổ MessageBox chứa số phần tử và số véctơ.

```
//biểu diễn số véctơ và số phần tử vào mesh
std::string numVertices;
sprintf ( (char*) numVertices.c_str (), “message”, MB_OK);
//biểu diễn số phần tử trong mesh
std::string numFaces;
sprintf ( (char*) numFaces.c_str(),
“numFaces = %d”,
pMeshSysMem->GetNumFaces());
MessageBox (NULL, numFaces.c_str (), “mesage”, MB_OK);
```

Biến pMeshSysMem phải chứa một mesh hợp lệ trước khi GetNumVertices hoặc là GetNumFaces được gọi.

The Attribute Table

Trong suốt quá trình tối ưu mesh, bạn có sự tùy chọn khởi tạo một bảng đặc tính. Bảng này bao gồm những thông tin về đặc điểm buffer của mesh. Attribute buffer sẽ chứa các đặc điểm của mỗi một véctơ trong mesh.

Như đã nói trước, mỗi mesh có thể chứa nhiều nhóm đặc tính. Mỗi nhóm đều chứa một danh sách các véctơ mà thực hiện chức năng của nhóm đó. Khi sử dụng nhiều nhóm đặc tính bạn có thể chia cắt một cách có lựa chọn mesh thành các phần riêng rẽ. Ví dụ mesh của một xe hơi có thể chia thành nhóm chứa thân xe và nhóm khác chứa bánh xe. Nếu nhóm chứa thân xe đánh dấu là nhóm 0 và nhóm chứa bánh xe là nhóm 1 thì bạn có thể sử dụng 2 cách gọi hàm để gọi hàm DrawSubset để vẽ toàn bộ xe hơi.

```
cafMesh->DrawSubset(0);          // vẽ thân
cafMesh->DrawSubset(1);          // vẽ bánh xe
```

Vì mỗi nhóm có thể cần các dữ liệu khác nhau, vì vậy việc gọi hàm SetMaterial phải được gọi trước hàm DrawSubset.

Để phân biệt các nhóm riêng rẽ trong mesh, bạn cần tạo bảng đặc tính. Bạn có thể chỉ tạo một bảng đặc tính bằng cách gọi một trong hai hàm tối ưu. Khi bạn gọi hàm tối ưu và xắp xếp lại các bề phần tử thì một bảng đặc tính sẽ được khởi tạo lại. Theo mặc định nếu như mesh cần nhiều hơn một loại vật liệu, một tập hợp con sẽ được khởi tạo cho mỗi nhóm. Trong trường hợp này khối lập phương bạn đã tạo trong ví dụ trước chứa chỉ 1 loại vật liệu. Hàm OptimizeMesh sau đây sẽ đưa khối lập phương chứa trong biến *boxMesh* và chia nó thành 2 nhóm nhỏ riêng. Một nửa của khối lập phương sẽ được biểu diễn bằng một phần vật liệu, còn phần thứ 2 sẽ biểu diễn bằng phần vật liệu khác.

```
/*****************************************************************************
*OptimizeMesh
*****************************************************************************/
void OptimizeMesh (void)
{
    //gọi hàm OptimizeInplace để khỏ tạo bảng đặc tính
    boxMesh-> OptimizeInplace(D3DXMESHOPT_ATTRSORT, 0, NULL, NULL, NULL);
    DWORD numAttr;
    D3DXATTRIBUTERANGE *attribTable = D3DXATTRIBUTERANGE [2];
    // lấy chỉ số của vật thể trong bảng
    boxMesh->GetAttributeTable(attributeTable, &numAttr);
    //chuyền đặc tính cho nhóm thứ nhất
    attributeTable[0].AttriId = 0;
    attributeTable[0].FaceStart = 0;
    attributeTable[0].FaceCount = 6;
    attributeTable[0].VertexStart = 0;
    attributeTable[0] .VertexCount = 8;

    //chuyển đặc tính cho nhóm thứ 2
    attributeTable[1]. AttriId = 1;
    attributeTable[1].FaceStart = 6;
    attributeTable[1].FaceCount = 6;
    attributeTable[1].VertexStart = 0;
    attributeTable[1] .VertexCount = 8;
    // viết bảng đặc tính lên mesh
    boxMesh->SetAttributeTable(attributeTable, 2);
}
```

Đoạn chương trình trên gọi hàm OptimizeInplace trong mesh của khối lập phương chưa biến boxMesh. Vì tôi sử dụng OptimizeInplace, nên tôi tiếp tục sử dụng mesh của khối lập phương nguyên bản.
Sau đó vì tôi tạo 2 nhóm đặc tính riêng rẽ nên tôi sẽ xây dựng 1 ma trận có hai phần từ kiểu D3DXATTRIBUTERANGE.

```
D3DXATTRIBUTERANGE *attribTable = D3DXATTRIBUTERANGE [2];
```

Mỗi phần tử cấu trúc D3DXATTRIBUTERANGE đều chứa thông tin mà Direct3D cần để xác định bảng đặc tính.
Cấu trúc D3DXATTRIBUTERANGE sẽ được chỉ ra dưới đây:

```
Typedef struct_ D3DXATTRIBUTERANGE {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount
} D3DXATTRIBUTERANGE;
```

Cấu trúc D3DXATTRIBUTERANGE có 5 biến:

- **AttribId** – chỉ số của nhóm hiện hành
- **FaceStart** –chỉ số phần tử đầu tiên trong nhóm này
- **FaceCount** – số phần tử sẽ có trong nhóm này
- **VertexStart** – chỉ số của véctớ đầu tiên trong nhóm này
- **VertexCount** – số véctơ mà nhóm này chứa

Sau khi bạn đã tạo ma trận cấu trúc D3DXATTRIBUTERANGE, bạn phải truy cập dữ liệu ở bảng đặc tính. Bạn có thể truy cập bảng đặc tính thông qua việc gọi hàm GetAttributeTable. Hàm GetAttributeTable có thể sử dụng trong hai cách:
Cách thứ nhất cho phép bạn xác định chỉ số của item mà hiện tại có trong bảng đặc tính. Khi bạn chuyển giá tri NULL cho tham số đầu tiên vào hàm GetAttributeTable và cho con trỏ trỏ đến biến có kiểu DWORD ở tham số thứ 2 thì mesh sẽ chỉ ra chỉ số của item hiện tại trong bảng. Item lấy được sẽ trả về một biến được gán cho tham số thứ 2. Một mẫu gọi hàm GetAttributeTable làm việc theo cách trên được chỉ ra dưới đây.

```
DWORD numAttr;        // biến giữ chỉ số của item trong bảng
// sử dụng hàm GetAttributeTable để chọn chỉ số của item trong bảng đặc tính.
boxMesh->GetAttributeTable (NULL, &numAttr);
```

Như bạn có thể thấy ở phần gọi hàm ở trên, biến *numAttr* được truyền cho tham số thứ 2. Khi hoàn thành việc gọi hàm này, numAttr sẽ chứa chỉ số của item hiện tại trong bảng đặc tính.
Bây giờ bạn đã có chỉ số của item, bạn cần truy cập vào dữ liệu trong bảng. Bạn có thể sử dụng hàm GetAttributeTable bằng cách khác để thu thập thông tin. Trước đây bạn đã tạo ma trận cấu trúc có 2 phần tử kiểu D3DXATTRIBUTERANGE. Khi bạn đưa ma trận attrriTable vào tham số đầu tiên và biến numAttr vào tham số thứ 2 thì hàm GetAttributeTable sẽ nhập ma trận attribTable với nội dung các dữ liệu vào trong bảng. Việc gọi hàm GetAttributeTable sẽ được sử dụng trong cách được chỉ ra dưới đây:

```
boxMesh->GetAttributeTable (attribTable, &numAttr);
```

Ở đây, bạn dễ dàng điều khiển và thay đổi dữ liệu với ma trận attribTable. Nếu xem lại hàm OptimizeMesh, bạn sẽ thấy rằng tôi đã thay đổi biến ở trong mỗi phần tử cấu trúc D3DXATTRIBUTERANGE. Tôi đã đổi cấu trúc đầu tiên ở phần tử đứng ở vị trí đầu tiên và phần tử thứ 6 trong mesh, cái này mô tả một nửa phần tử của khối lập phương.

```
//truyền đặc tính cho biến ở  nhóm thứ 1
attributeTable[0].AttriId = 0;              // chỉ số của nhóm
attributeTable[0].FaceStart = 0;            // phần tử đầu tiên trong nhóm
attributeTable[0].FaceCount = 6;            //số bề phần tử trong nhóm
attributeTable[0].VertexStart = 0;          //vécto đầu tiên trong nhóm
attributeTable[0] .VertexCount = 8;         //số vécto trong nhóm
```

Nhóm thứ 2 bắt đầu với phần tử thứ 6 và bắt đầu lại với 6 phần tử. Sự thay đổi khác trong cấu trúc này là sự chuyển AttribId cho 1.

```
//chuyển đặc tính cho nhóm thứ 2
attributeTable[1]. AttriId = 1;
attributeTable[1].FaceStart = 6;
attributeTable[1].FaceCount = 6;
attributeTable[1].VertexStart = 0;
attributeTable[1] .VertexCount = 8;
```

Bước cuối cùng cần làm là cắt khối lập phương thành 2 nhóm đặc tính riêng lẽ để làm thay đổi tính chất của bảng đặc tính và cập nhật lại mesh của khối lập phương với những thay đổi này. Sự cập nhật các đặc tính của bảng trong mesh đã được thực hiện qua hàm SetAttributeTable dưới đây:

```
HRESULT SetAttributeTable(
CONST D3DXATTRIBUTERANGE *pAttribTable,
DWORD cAttribTableSize
);
```

hàm SetAttributeTable chỉ có 2 tham số:

- **pAttribTable** – con trỏ trỏ tới bảng đặc tính để cập nhật lại mesh

- **cAttribTableSize** – giá trị chỉ rõ kích thước của bảng đặc tính

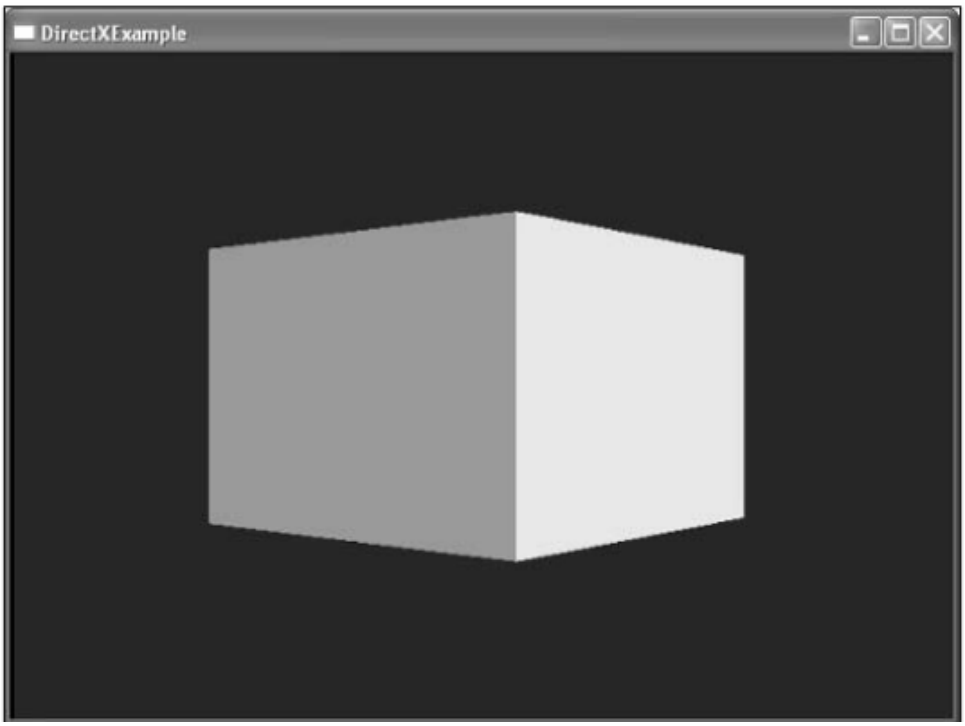
Đoạn chương trình sau sẽ chỉ ra bằng cách nào hàm SetAttributeTable có thể cập nhật bảng trong mesh. Biến attribTable biểu diễn ma trận các đặc tính sẽ được truyền cho tham số thứ nhất. Vì tôi muốn thay đổi khối lập phương trong cả 2 nhóm đặc tính nên tôi truyền giá trị 2 cho tham số thứ 2.

```
boxMesh->SetAttributeTable(attribTable, 2);
```

Bây giờ mesh của khối lập phương đã cắt thành 2 nhóm riêng, bạn phải thay đổi làm sao để khối lập phương được hiển thị. Để làm điều này cần hai lần gọi hàm DrawSubset, mỗi hàm chỉ rõ giá trị của từng nhóm để vẽ. Đoạn chương trình sau sẽ chỉ ra cách gọi cập nhật. Hơn nữa việc gọi hàm SetMaterial trước hàm mỗi lần gọi DrawSubset sẽ làm cho dữ liệu thay đổi trước mỗi lần kết xuất một nửa khối lập phương.

```
//lấy dữ liệu
pd3dDevice->SetMaterial (&materials[0].MatD3D);
//vẽ phần đầu tiên của mesh
mesh->DrawSubset(0);

//lấy dữ liệu thứ 2
pd3dDevice->SetMaterial (&materials[1].MatD3D);
//vẽ phần thứ hai của mesh
mesh->DrawSubset(1);
```

Hình 7.2 sẽ chỉ ra cube đã cập nhật được biểu diễn với dữ liệu riêng của mỗi nhóm.



Hình 7.2 Khối lập phương và sự áp dụng hai loại material. Material thứ nhất là màu xanh, và thứ hai là màu đỏ.

Predefined Mesh.

Sự kiến tạo mesh thủ công là một công việc nhàm chán và không dự đoán được tất cả các chi phí. May mắn thay, việc thiết kế chương trình thường loại trừ việc thiết kế thủ công. Ngoài ra DirectX có cung cấp một số hàm để hỗ trợ việc tạo đối tượng.

D3DX Object Creation

Chúng ta đã đi được một chặng đường khá xa, tôi đã chỉ ra sự hình thành phức tạp của mô hình 3D bằng phương pháp thủ công. Và tôi chỉ sử dụng những đối tượng đơn giản, ví dụ khối lập phương, ta dễ dàng biểu diễn. Trong DirectX cung cấp phương pháp phức tạp hơn để tạo các đối tượng đơn giản trong thư viện tổng hợp D3DX (D3DX Utility Library).
Nhứng hàm sau trong D3DX sẽ giúp bạn tạo một đối tượng đơn giản 3D như khối lập phương, khối cầu, khối trụ.

```
D3DXCreatBox – tạo  khối lập phương
D3DXCreatSphere – tạo khối cầu
D3DXCreatCylinder – tạo khối lăng trụ
D3DXCreatTeapot – tạo mô hình 3D ấm pha trà.
```

Tạo Khối Lập Phương

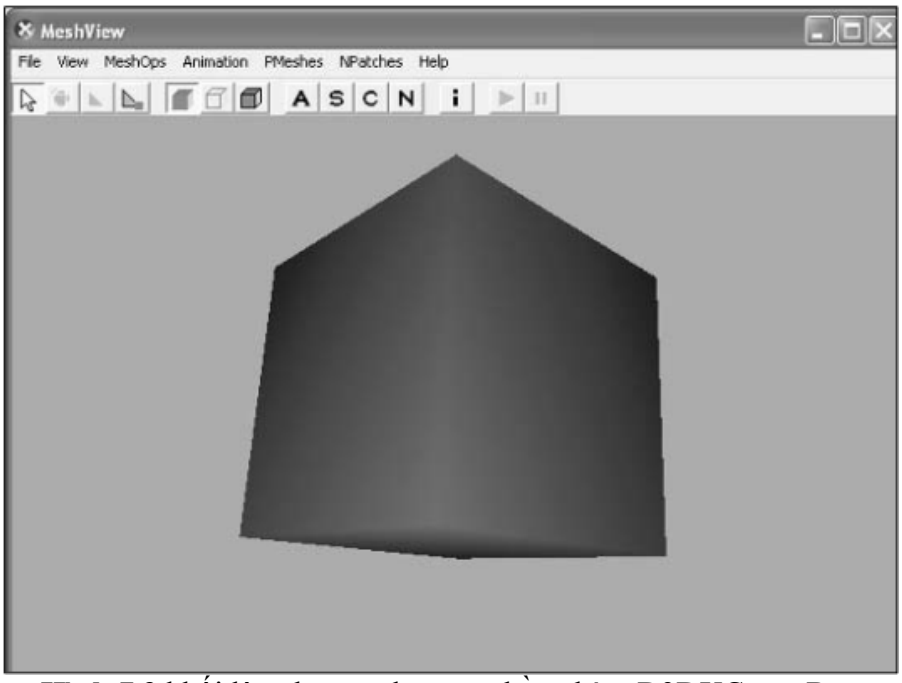
Bạn có t hể sử dụng hàm D3DXCreatBox được dịnh nghĩa dưới đây khi bạn muốn tạo một khối lập phương đơn giản. Kết quả khối lập phương sẽ là một đối tượng ID3DXMesh hoàn chỉnh mà bạn có thể tối ưu hoặc điều khiển theo ý muốn.

```
HRESULT D3DXCreateBox(
        LPDIRECT3DDEVICE9 pDevice,
        FLOAT Width,
        FLOAT Height,
        FLOAT Depth,
        LPD3DXMESH **ppMesh,
LPD3DXBUFFER *ppAdjacency
);
```

Hàm D3DXCreateBox có 6 tham số:

- **pDevice** – con trỏ trỏ tới Direct3D device hợp lệ
- **Width** – bề rộng của khối lập phương tính dọc theo trục X
- **Height** – chiều cao của khối lập phương tính dọc theo trục Y
- **Depth** – chiều sâu của khối lập phương tính theo trục Z
- **ppMesh** – địa chỉ trỏ tới con trỏ ID3DXMesh. Biến này chứa mesh của khối lập phương
- **ppAdjacency** –adjacency buffer. Nếu bạn không muốn giữ thông tin này , bạn có thể truyền NULL cho tham số này.

Khối được tạo với hàm này sẽ xem như một khối mà bạn đã sử dụng trong phần trước. Hình 7.3 sẽ chỉ ra một khối lập phương được tạo với hàm D3DXCreateBox.



Hình 7.3 khối lập phương được tạo bằng hàm D3DXCreateBox

Tạo hình khối ấm trà

Hình khối ấm trà được sử dụng rộng rãi trong các ví dụ về mô hình hình học 3D và nó cũng có thể được tạo dễ dàng trong Direct3D. Bạn đã kết xuất nó vì bạn đã sử dụng nó như một mô hình trong chương 6, "Vertex Colors, Texture Mapping, and 3D Lighting". Để tạo hình khối 3D ấm trà, bạn cần sử dụng hàm D3DXCreateTeapot được định nghĩa dưới đây:

```
HRESULT D3DXCreateTeapot(
        LPDIRECT3DDEVICE9 pDevice,
        LPD3DXMESH **ppMesh,
        LPD3DXBUFFER *ppAdjacency
);
```

Hàm D3DXCreateTeapot có 3 tham số cần thiết:

- pDevice – đối tượng Direct3D hợp lệ
- ppMesh – đối tượng ID3DXMesh trong đó sẽ đưa mesh được tạo vào
- ppAdjacency – adjacency buffer. Nếu bạn không muốn giữ thông tin này , bạn có thể truyền NULL cho tham số này.

Điều không may là hàm này không cho phép bạn thay đổi kích thước của ấm trà mà bạn muốn tạo. Dòng code đơn giản sau sẽ tạo ra một ấm trà cho bạn:

```
D3DXCreateTeapot (pd3dDevice, &teapotMesh, NULL);
```

Tạo hình khối cầu

Hình khối cầu rất có ích trong 3D. sử dụng chỉ những khối cầu, bạn có thể tạo một mô hình tượng trưng cho hệ phần tử trời. Nếu bạn thấy cần tạo khối cầu, bạn có thể sử dụng hàm D3DXCreateSphere được chỉ ra dưới đây:

```
HRESULT D3DXCreateSphere (
        LPDIRECT3DDEVICE9 pDevice,
        FLOAT Radius,
        UINT Slices,
        UINT Stacks,
        LPD3DXMESH **ppMesh,
        LPD3DXBUFFER *ppAdjacency
);
```

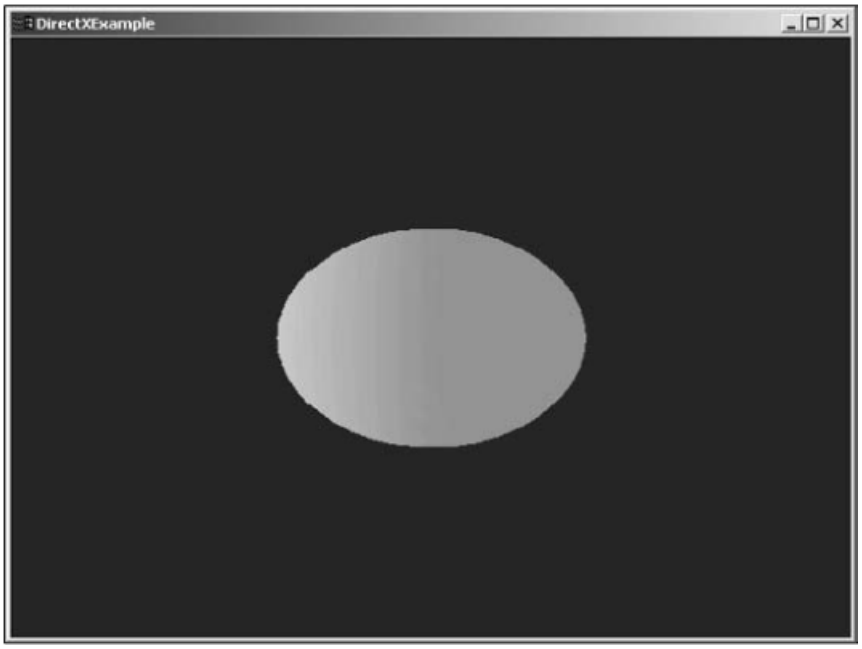
Hàm D3DXCreateSphere có 6 tham số:

- pDevice – Direct3D device hợp lệ
- Radius – bán kính của khối cầu có kiểu float
- Slices – số đoạn nối chiều dọc được chỉ ra
- Stacks - số đoạn nối chiều ngang được vẽ ra
- ppMesh – đối tượng ID3DXMesh lưu giữ khối cầu được tạo
- ppAdjacency - adjacency buffer. Nếu bạn không muốn giữ thông tin này , bạn có thể truyền NULL cho tham số này.

Đoạn chương trình nhỏ dưới đây sẽ chỉ ra cách làm thế nào để sử dụng hàm D3DXCreateSphere:

```
//tạo khối cầu
loat  sphereRadius=30;
int numSlices = 20;
int numStacks = 20;
D3DXCreateSphere ( pd3dDevice,
        sphereRadius,
        numSlices,
        numStacks,
        SphereMessh,
        NULL
);
```

Hình 7.4 sẽ chỉ ra một khối cầu được tạo bằng hàm D3DXCreateSphere. Độ lớn của giá trị biểu diễn qua tham số Slices va Stacks sẽ là độ nhẵn của khối cầu.



Tạo khối trụ

Đối tượng cuối cùng mà chúng ta sẽ biểu diễn là một khối trụ. Hàm D3DXCreateCylinder được chỉ ra dưới đây sẽ chỉ ra một cách rõ ràng các đặc điểm của khối trụ, ví dụ như chiều dài và bán kính của mỗi đáy.

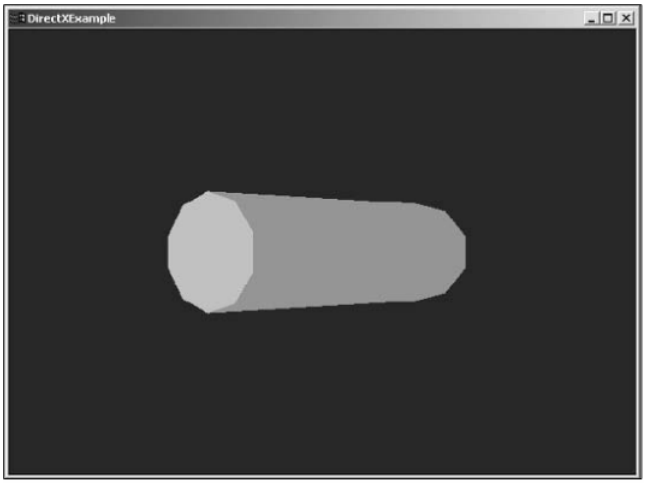
```
HRESULT D3DXCreateCylinder (
        LPDIRECT3DDEVICE9 pDevice,
        FLOAT Radius1,
        FLOAT Radius2,
        FLOAT Lenght,
        UINT Slices,
        UINT Stacks,
        LPD3DXMESH **ppMesh,
        LPD3DXBUFFER *ppAdjacency
);
```

Hàm D3DXCreateCylinder có 8 tham số:

- **pDevice** - Direct3D device hợp lệ
- **Radius1** – bán kính đáy nổi của khối trụ tính dọc theo trục Z và có kiểu float
- **Radius2** - bán kính đáy chìm của khối trụ tính dọc theo trục Zvà có kiểu float
- **Length** – chiều cao của khối trụ
- **Slices** – số phần tử tạo hình theo chiều dài của khối trụ
- **Stacks** – sô phần tử tạo hình theo chu vi đường tròn
- **ppMesh** - đối tượng ID3DXMesh lưu giữ khối trụ được tạo
- **ppAdjacency**- adjacency buffer. Nếu bạn không muốn giữ thông tin này , bạn có thể truyền NULL cho tham số này.

Ví dụ sau chỉ ra cách tạo một khối trụ:

```
//xác dịnh đặc điểm của khối trụ
float cylRadius1=2.0;
float cylRadius2= 2.0;
float cylLength = 7.0;
int cylSlices = 10;
int cylStack = 10;
//tạo khối trụ
D3DXCreateCylinder (pd3dDevice,
                    cylRadius1,
                    cylRadius2,
                    cylLength,
                    cylSlices,
                    cylStack,
                    &cylMesh,
                    NULL);
```



Hình 7.5 chỉ ra khối trụ được tạo bởi hàm *D3DXCreateCylinder*

Bạn có thể tìm source code của ví dụ về cách sử dụng các hàm *D3DX* tại thư mục Chapter7\example2 trong của đĩa CD-ROM đi kèm.

Định dạng của mesh trong Direct3D: File X

Tạo mesh bằng code không phải là một cách hay để tạo một thế giới 3D. Hầu hết các game cần những model dạng complex và detail có chất lượng tốt, chúng có thể được tạo bằng tay. Như tôi đã nói ở trên, các thanh công cụ thiết kế hiện có trên thị trường có thể

giúp bạn tạo các model 3D. Sau khi tạo các model, bạn có thể xuất chúng vào các định dạng file.
Microsoft đã tạo một dạng file chuyên dùng cho việc xuất các model 3D và nó được gọi là file X. File X có thể một trong hai dạng: text hoặc là binary. Đoạn chương trình sau sẽ chỉ ra một phần nhỏ của File X ở định dạng text.

```
MeshVertexColors{
        8,
        0;0.000000; 0.000000; 1.000000; 0.000000;;,
        1;1.000000; 1.000000; 1.000000; 0.000000;;,
        2;0.000000; 0.000000; 1.000000; 0.000000;;,
        3;0.000000; 1.000000; 1.000000; 0.000000;;,
        4;0.000000; 0.000000; 1.000000; 0.000000;;,
        5;0.000000; 1.000000; 1.000000; 0.000000;;,
        6;0.000000; 0.000000; 1.000000; 0.000000;;,
        7;0.000000; 1.000000; 1.000000; 0.000000;;,
}
```

Đây là mẫu mô tả định dạng trong đó dữ liệu nằm trong sự điều khiển của mỗi section của X file. Một mẫu có cùng tên được chỉ ra dưới đây sẽ làm việc với cấu trúc MeshVertexColors:

```
template MeshVertexColors\
{ \<1630B821- 7842-11cf-8F52-0040333594A3>\
        DWORD nVertexColors;\
        Array IndexColor vertexColors[nVertexColors];\
}
```

Mỗi template chứa 2 phần: nhận dạng số đơn nhất, kèm trong trong hai dấu ngoặc, và template có thể chứa khai báo dữ liệu. Trong ví dụ này, template MeshVertexColors khai báo hai dạng dữ liệu: kiểu DWORD dùng để chứa số véctơ colors, và một ma trận thuộc kiểu IndexColor.

File X được tạo là file như thế nào?

File X thường được tạo trong các phần mềm thiết kế 3D. Bạn có thể mã hóa một file X bằng phương pháp thủ công, nhưng đối với các complex model, việc này có thể sẽ rất rắc rối. Thường thì một thợ đồ họa hay tạo model và sau đó xuất chúng vào định dạng file X.
Microsoft đã thực hiện hai chương trình có thể chuyển model thành định dạng thích hợp. Chương trình đầu tiên là conv3ds.exe, là một ứng dụng command-line, chuyển một model 3D input vào định dạng file 3DS. File 3DS là dạng thường được tạo trong 3ds max, các packages thiết kế khác cũng hỗ trợ định dạng file này.
Chương trình thứ hai là xSkipExp.dle là một chuyển thể của 3ds max, cho phép bạn xuất trực tiếp các model từ trong môi trường thiết kế. Cả hai ứng dụng này đều có trong DirectX Software Development Kit (SDK).

Lưu Mesh vào file X

Vì đã có sẵn hai chương trình để tạo file X, nên có lẽ bạn thấy lạ vì sao tôi lại chỉ cho bạn cách tạo bằng phương pháp lập trình các file cho mình. Ồ, không phải tất cả các lập trình game đều viết về mã hóa đồ họa engine và trí tuệ nhân tạo AI; bạn cần phải tạo các công cụ để hoạt động các lập trình viên trở nên dễ dàng hơn. Ví dụ bạn được đề nghị tạo một ứng dụng mà có thể đọc mọi định dạng file 3D và xuất ra file X. May mắn thay D3DX Utility Library đã có để cứu bạn bằng các hàm tạo file X.

D3DXSaveMeshToX

D3DX chứa một hàm gọi D3DXSaveMeshToX mà bạn có thể sử dụng để tạo file X. Trước tiên bạn có thể sử dụng hàm này, mặc dù dữ liệu của bạn phải tập trung vào một ID3DXMesh object. Như bạn mong muốn từ trước, tôi đã dẫn dắt bạn xuyên suốt quá trình tạo và biểu diễn mesh. Trong phần này, tôi sẽ quay lại phần trước và chỉ cho bạn cách làm thế nào để lấy mesh đã tạo trước đó và lưu chúng vào file X.

Nhắc lại một số thứ mà bạn đã học về mesh trước kia: Mesh được tạo bằng cách sử dụng một trong hai hàm: D3DXCreateMesh hoặc là D3DXCreateMeshFVF. Sau khi tạo mesh, bạn bổ xung véctơ và index buffer cùng với thông tin gắn liền với model bạn đang tạo. Ở đây bạn nên có một ID3DXMesh object hợp lệ tuyệt đối để chuẩn bị tạo một file X từ chúng.
Hàm D3DXSaveMeshToX được định nghĩ dưới đây sẽ lấy mesh bạn tạo và lưu nó vào đĩa ở dạng file X.

```
HRESULT D3DXSaveMeshToX(
        LPCTSTR pFilename,
        LPD3DXMESH pMesh,
        CONST DWORD* pAdjacency,
        CONST D3DXMATERIAL* pMaterials,
        CONSTD3DXEFFECTINSTANCE* pEffectInstances,
        DWORD NumMaterials,
        DWORD Format
)
```

hàm D3DXSaveMeshToX có 6 tham số:

- **pFilename** – là một biến kiểu string dùng để lưu tên của file X .
- **pMesh** – con trỏ trỏ tới biến ID3Dmesh
- **pAdjacency** – con trỏ trỏ tới ma trận có 3 phần tử kiểu DWORD .
- **pMaterials** – con trỏ trỏ tới ma trận kiểu cấu trúc D3DXMATERIAL
- **pEffectInstances** – con trỏ trỏ tới ma trận kiểu effect instances
- **NumMaterials** – số cấu trúc D3DXMATERIAL trong biến pMaterials
- **Format** – định dạng của file X sẽ được lưu. Ba định dạng có thể có:
- **DXFILEFORMAT_BINARY** – file X được lưu trong định dạng binary
- **DXFILEFOMAT_TEXT** – File X được lưu ở định dạng text-viewable
- **DXFILEFORMAT_COMPRESSED** – File X được lưu ở dạng nén.

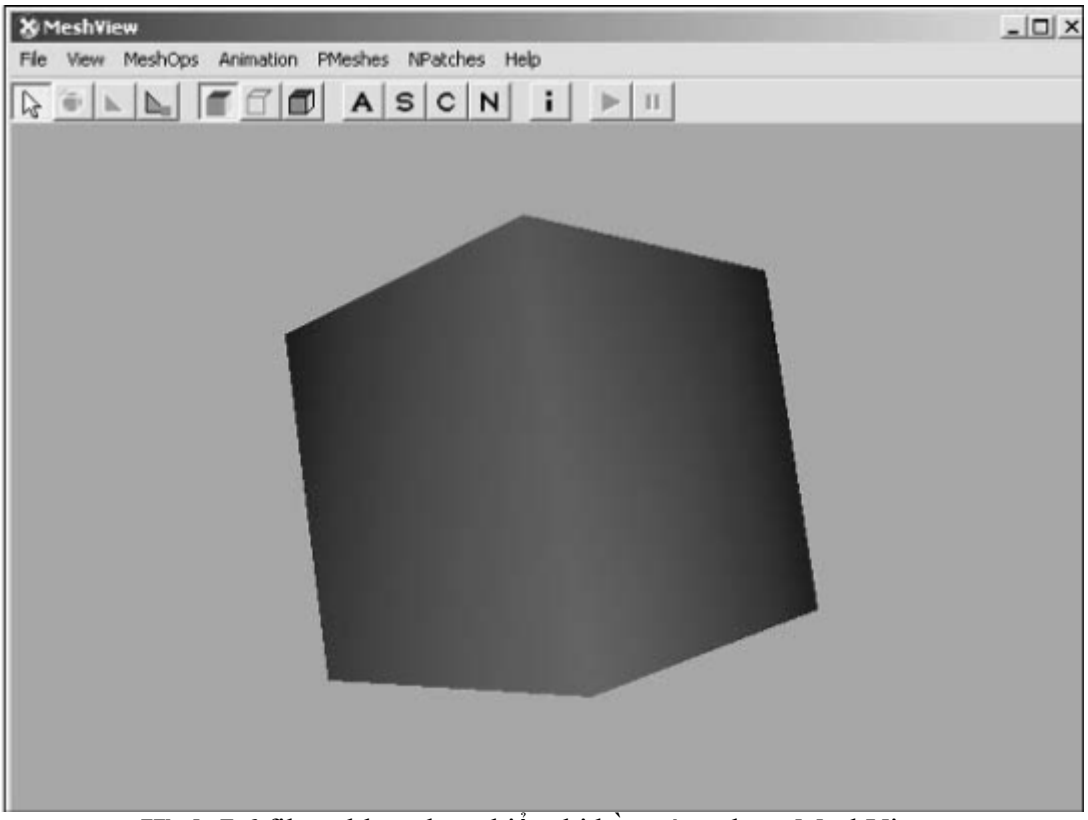
Nguồn chương trình chỉ ra dưới đây là một ví dụ về cách sử dụng hàm D3DXSaveMeshToX

```
LPD3DXMESH cubeMesh;        // con trỏ trỏ tới ID3DXMESH
D3DXMATERIAL material ;     //cấu trúc đơn D3DXMATERIAL

HRESULT HR;
//tạo mesh được dùng để lưu model khối lập phương
hr= D3DXCreateMeshFVF( 12,
                       8,
                       D3DXMESH_MANAGED,
                       D3DFVF_CUSTOMVERTEX,
                       pd3dDevice,
                       &cubeMesh);
//tạo véctơ và index buffer
//hàm này không được định nghĩa ở đây, nhưng nó điền véctơ và index buffers của mesh
//với thông tin cần thiết cho một khối lập phương. Bạn có thể xem phần tạo mesh từ code
//để có thể mô tả một cách đầy đủ về hàm này
```

```
Setup VBIB();
//Lấy một đối tượng mesh hợp lệ và lưu nó vào file X
D3DXSaveMeshToX ("cube.x",                  //tên file sẽ lưu
                 cubeMesh,                  //giao diện   ID3DXMESH
                 NULL,                      //dữ liệu liền kề( không sử dụng)
                 &material,                 //ma trận kiểu cấu trúc D3DXMATERIAL
                 NULL,                      //ma trận kiểu effect instances(không sử dụng)
                 1,                         // số cấu trúc kiểu D3DXMATERIAL
                 DXFILEFORMAT_TEXT);        // lưu file X vào dạng text
```

Phần quan trọng trong đoạn chương trình này là việc gọi hàm D3DXSaveMeshToX. Hàm này trình bày tỉ mỉ tên file đực lưu, mesh object được lưu và định dạng file X được lưu.

Hình 7.6 sẽ chỉ ra mesh của khối lập phương *cube.x* , nó giống với hình khi xem từ ứng dụng MeshView trong thư mục DXSDK\Bin\DXUtils . Ứng dụng này được cài đặt khi bạn cài đặt DirectX SDK.
Bạn có thể tìm thấy đầy đủ source của ví dụ về lưu file X chỉ ra cách sử dụng hàm D3DX được mô tả trong chương này ở chapter7\exemple3 trong của CD-ROM.



Hình 7.6 file cuble.x được hiển thị bằng ứng dụng MeshView.

Load một Mesh từ file X

Giờ bạn đã biết làm thế nào để lưu file X, vậy câu hỏi tiếp theo là: làm thế nào để load một file X từ đĩa? Bạn có thể viết một loader cho mình, việc này rất cần thiết để bạn hiểu biết rộng các định dạng file X, hoặc bạn có thể xem thêm trong thư viện *D3DX* .
Vì file X là định dạng file mà Microsoft cung cấp cho DirectX nên Microsoft cũng đã cung cấp hàm để loading những file này vào ứng dụng của bạn.

Sử dụng hàm D3DXLoadMeshFromX

Hàm *D3DXLoadMeshFromX* định nghĩ dưới đây được sử dụng để load file X từ đĩa:

```
HRESULT D3DXLoadMeshFromX(
        LPCTSTR pFilename,
        DWORD Options,
        LPDIRECT3DDEVICE9 pDevice,
        LPD3DXBUFER* ppAdjacency,
        LPD3DXBUFER* ppMaterials,
        LPD3DXBUFER* ppEffectInstances,
        DWORD* pNumMaterials,
        LPD3DXMESH* ppMesh
);
```

Hàm D3DXLoadMeshFromX có 8 tham số:

- **pFilename** – một chuỗi dùng để lưu file X
- **Options** – cờ hiệu chỉ định mesh sẽ được load như thế nào. Bạn có thể tìm thấy những giá trị này trong bảng liệt kê D3DXMESH
- **pDevice** – Direct3D device hợp lệ
- **ppAdjacency** – adjacency buffer.
- **ppMaterials** – con trỏ trỏ tới buffer chứa giữ liệu
- **ppEffectInstances** – con trỏ của buffer chứa ma trận effect instances
- **pNumMaterials** – số material mà bạn tìm thấy trong biến ppMaterials
- **ppMesh** – đối tượng ID3DXMesh sẽ lưu giữ mesh khi hàm này kết thúc xong.

Đoạn code sau chỉ ra cách làm thế nào để sử dụng hàm D3DXLoadMeshFromX cho việc load một file X từ đĩa.

```
//biến giữ giá trị trả về
HRESULT hr;
//biến giữ mesh được load lên
LPD3DXMÉH pMeshSysMem;
//buffer giữ dữ liệu liền kề
LPD3DXBUFFER ppAdjacencyBuffer;
// buffer giữ materials
LPD3DXBUFER pD3DXMtrBuffer;

//load mesh từ đĩa
hr= D3DXLoadMeshFromX(
        “cube.x”,
        D3DXMESH_SYSTEMMEM,
        pd3dDevice,
        &ppAdjacencyBuffer,
        &pD3DXMtrBuffer,
        NULL,
        &m_dwNumMaterrials,
        &pMeshSysMem
);
// kiểm tra mesh đã load thành công chưa
if (FAILDED (hr))
        return false;
```

Đoạn code trên load mesh trong file cube.x và đưa nó vào biến pMeshSysMem. Trong suốt quá trình gọi hàm D3DXLoadMeshFromX, biến m_dwNumMaterials đã được gán bởi số materials trong mesh. Sử dụng giá trị này, bạn có thể lấy materials từ mesh và đưa chúng vào material buffer để sử dụng sau này. Mỗi buffer được sử dụng khi kết xuất mesh làm thay đổi material chung của các nhóm nhỏ.

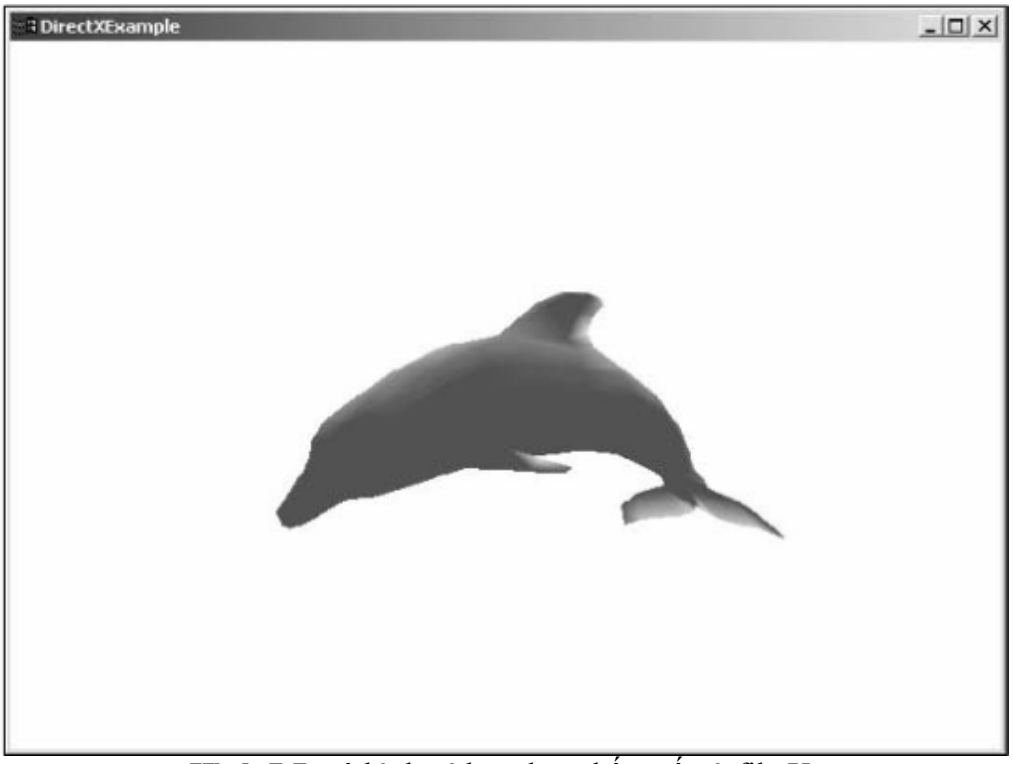
Vì biến pD3DXMtrBuffer chứa tất cả thông tin về material của mesh, nên bạn cần tách mỗi nhóm material thành các cấu trúc riêng kiểu D3DMATERIAL9. Đoạn code sau sẽ lấy một con trỏ của material trong mesh và đưa nó vào trong kiểu cấu trúc D3DMATERIAL9 bằng vòng lặp for.

```
//lấy con trỏ trỏ tới material buffer trong mesh
D3DXMATERIAL* matMaterials = (D3DXMATERIAL*)pD3DXMtrBuffer->GetBufferPointer();
//khai báo ma trận của material
D3DXMATERIAL9* m_pMeshMaterials;

//tạo hai phần tử cấu trúc D3DXMATERIAL9
m_pMeshMaterials = new D3DXMATERIAL9 [m_dwNumMaterials];
//vòng lặp

for( DWORD i=0;i< m_dwNumMaterials; i++)
{
//copy material
m_dwNumMaterials[i]=matMaterials[i].MatD3D;

//lấy màu xung quanh cho material (D3DX không làm việc này)
m_ dwNumMaterials[i].Ambient – m_ dwNumMaterials[i].Diffuse;
}
```

Bây giờ mesh đã được load, bạn có thể dễ dàng kết xuất chúng lên màn hình. Hàm drawMesh sau đây chỉ ra một cách thường dùng để kết xuất mesh từ file X.

```
/*****************************************************************************
*drawMesh
*****************************************************************************
void dxManager::drawMesh(void)
{
        D3DXMATRIX meshMatrix, scaleMatrix, rotaMatrix;
        createCamera(1.0f, 750.0f);                     //
        moveCamera(D3DXVECTOR3(0.0F, 0.0F, -450.0F));
        pointCamera(D3DXVECTOR3(0.0F, 0.0F, 0.0F));
        //lấy độ quay
        //D3DXMatrixRotationY(&rotateMatrix, timeGetTime()/1000.f);
        //lấy vô hướng
        D3DXMatrixScaling(&rotaMatrix, 0.5f, 0.5f, 0.5f);
        //làm tăng vô hướng và độ quay của ma trận
        D3DXMatrixMultiply(&meshMatrix, &scaleMatrix, &roteMatrix);
        //dịch chuyển đối tượng trong môi trường
        pd3dDevice->SetTransform(D3DTS_WORLD, &rotateMatrix);
        //vòng lặp
        for( DWORD i=0;i< m_dwNumMaterials;i++)
        {
                //lấy materials cho mesh
                pd3dDevice->SetMaterial(&m_pMeshMaterials[i]);
                //vẽ mesh con
                pMeshSysMem->DrawSubset(i);
        }
}
```

Bây giờ bạn có thể thấy mesh trên màn hình mà bạn đã load từ file X. Hình 7.7 chỉ ra mô hình con cá heo được kết xuất. File X của mô hình cá heo có trong DirectX SDK và bạn có thể tim thấy trong DXSDK\Samples\Media . Bạn cũng có đầy đủ source code về ví dụ chỉ ra cách làm thế nào để load một file X từ đĩa ở thư mục Chapter7\example4 trong CD-ROM.



Hình 7.7 mô hình cá heo được kết xuất từ file X.

Tổng kết chương

Giờ bạn đã có thể load vào một 3D model, và làm cho game của bạn thật hơn trong phần tiếp theo.Bạn sẽ không bị giới hạn bởi những hình khối hay hình cầu đơn giản mà có thể dùng được những vật thể thật sự. Nếu bạn không có kỹ xảo thiết kế 3D, bạn vẫn có thể cải thiện game của mình bằng cách lấy những mô hình của rất nhiều model 3D có trên mạng.

Những vấn đề đã học

Trong chương này bạn đã học những vấn đề sau

- Làm thế nào để tạo một mesh
- Hiển thị một mesh đã tạo như thế nào
- Tối ưu dữ liệu trong mesh bằng cách nào
- Cắt mesh thành các phần nhỏ để nhiều material được dùng cùng một lúc như thế nào
- Các bước cần thiết để load mesh vào định dạng file X từ đĩa.
- Làm thế nào để tạo và lưu file X

Câu hỏi ôn tập

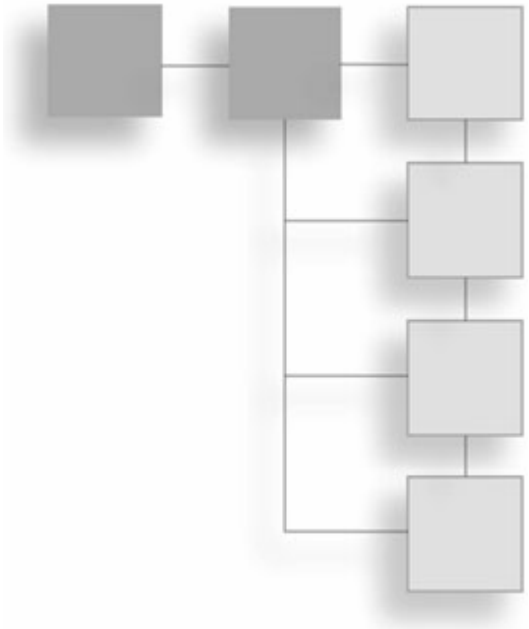
Bạn có thể tìm thấy câu trả lời cho câu hỏi ôn tập và bài tập trong Appendix A, “Answers to End-of-Chapter Exercises”.

1. Hai hàm nào được dùng khi tạo mesh?
2. Hàm OptimizeInplace khác với hàm Optimize ở điểm nào?
3. Bảng đặc tính chứa trong mesh dùng để làm gì?
4. Hàm nào trả về số vectơ trong mesh?
5. Ba kiểu cờ hiệu định dạng gì giúp bạn sử dụng hàm D3DXSaveMeshToX?

Bài tập tự làm

1. Viết một ví dụ nhỏ về load và hiển thị nhiều file X một lúc.
2. Viết một hàm mà trả về bản tối ưu của một mesh.

CHƯƠNG 8



VẬT THỂ, ĐIỂM CHÈN VÀ CÁC HIỆU ỨNG

Particle được sử dụng ở mọi nơi trong game để tạo ra những hiệu ứng khó quên. Từ những quả rocket tìm kiếm mục tiêu cho đến những vụ nổ mà nó tạo ra, particle làm cho thế giới ảo trong game trở lên hấp dẫn hơn.

Những gì bạn sẽ được học trong chương này:

- Particle là gì và được dùng thế nào
- Particle bao gồm những thuộc tính gì
- Làm thế nào để định nghĩa và render các particle
- Particle emitter là gì và ứng dụng của nó
- Cách render các particle bằng Direct3D's point sprite.

Particles

Particle được dùng trong game để biểu diễn những mảnh vỡ nhỏ, tia lửa của pháo hoa, hay bất kỳ một thực thể nhỏ chuyển động nào.

Mỗi particle là một thực thể độc lập với cách thức chuyển động và biểu hiện được định nghĩa trước, ngay khi nó được tạo ra. Trong suốt quá trình tồn tại, nó tự cập nhật các thuộc tính về hướng di chuyển và tốc độ di chuyển. Particle dùng cho các hiệu ứng khác nhau thường được tạo ra từ một thứ gọi là emitter. Công việc của emitter là tạo ra và xác lập các thuộc tính cho particle trước khi kích hoạt chúng. Emitter cũng điều khiển cả số lượng và thời điểm tạo ra các particle.

Particle được tạo ra bởi các đa giác có texture, gọi là billboard, chúng luôn hướng (mặt phẳng) về phía camera. Billboard có rất nhiều ứng dụng như tạo các đám mây, rừng cây, và nhiều hơn cả là tạo các particle. Bởi vì mỗI billboard thường chỉ chứa 2 tam giác (với 1 loạI texture), nên bạn có thể render một chuỗI nhiều billboard để tạo ra các hiệu ứng đặc biệt đẹp mắt.

Trước khi tung các particle vào trong scene, bạn cần biết chi tiết cách mà nó làm việc.

Các thuộc tính của Particle

Mỗi particle có những thuộc tính riêng quy định cách biểu hiện của nó. Danh sách dưới đây là một vài thuộc tính cơ bản mà hầu hết các particle đều có:

- **Vị trí.** Vị trí hiện tại của particle.
- **Chuyển động.** Điều khiển hướng và tốc độ di chuyển.
- **Màu sắc.** Màu của particle.

- **Cờ hoạt động.** Thuộc tính nhằm xác định trạng thái hoạt động của particle. Bởi vì một số particle có thể chuyển động ra ngoài màn hình do đó cờ này dùng để tiết kiệm tài nguyên hệ thống bằng cách giải phóng các particle này.
- **Texture**. Texture sử dụng cho particle.

Mỗi particle được phóng ra từ emitter đều chứa các thuộc tính trên. Trong phần tiếp theo, chúng ta sẽ học cách sử dụng các thuộc tính đó để tạo ra cấu trúc của một particle.

Cấu trúc Particle

Cấu trúc particle chứa toàn bộ các thuộc tính của một particle. Bằng cách tạo ra một mảng biến có cấu trúc này, bạn có thể cập nhật và render nhiều particle một cách nhanh chóng.

```
typedef struct
{
D3DXVECTOR3 m_vCurPos; // vecto vị trí
D3DXVECTOR3 m_vCurVel; // vecto chuyển động
D3DCOLOR m_vColor; // màu của particle
BOOL m_bAlive; // cờ hoạt động
} Particle;
```

Vị trí hiện tại của một particle được lưu trữ trong biến có cấu trúc D3DXVECTOR3. Cấu trúc này cho phép mô tả particle trong không gian ba chiều.

Vecto chuyển động chứa thông tin về hướng và vận tốc của particle. Màu của particle thì được chứa trong cấu trúc dạng D3DCOLOR. Bạn có thể sử dụng bất kì lệnh tạo màu D3DCOLOR nào để định nghĩa màu cho particle.

Biến cuối cùng trong cấu trúc particle là m_bAlive. Biến này xác định rằng một particle có đang được hoạt động hay không. Trước khi một particle được phóng ra từ emitter, thì biến này có giá trị là false.

Các particle được tạo ra như thế nào?

Các particle được tạo ra bằng cách gán cho các biến trong cấu trúc particle một giá trị khởi tạo. Trong suốt quá trình tồn tại của mỗi particle, các biến này sẽ được cập nhật liên tục dựa trên hiệu ứng của emitter. Bởi vì bạn sẽ thường xuyên cần đến nhiều hơn một particle, cho nên tốt nhất là bạn nên định nghĩa chúng trong một mảng. Đoạn code sau cho thấy cách khai báo và khởi tạo cho một nhóm các particle.

```
// Định nghĩa số particle cần tạo
#define MAXNUM_PARTICLES 150
// Định nghĩa cấu trúc lưu trữ thông tin về particle
typedef struct
{
// vecto vị trí
D3DXVECTOR3 m_vCurPos;
// vecto chuyển động
D3DXVECTOR3 m_vCurVel;
// màu
D3DCOLOR m_vColor;
} Particle;
// tạo một mảng các particle
Particle ParticleArray[MAXNUM_PARTICLES];
// vòng lặp để khởi tạo giá trị cho particle
for( int i = 0; i < MAXNUM_PARTICLES; i++ )
{
// đặt vị trí hiện tại cho các particle ở gốc tọa độ
ParticleArray[i].m_vCurPos = D3DXVECTOR3(0.0f,0.0f,0.0f);
// tạo các giá trị ngẫu nhiên cho các thành phần của vecto chuyển động
float vecX = ((float)rand() / RAND_MAX);
```

```
float vecY = ((float)rand() / RAND_MAX);
float vecZ = ((float)rand() / RAND_MAX);
// sử dụng các giá trị ngẫu nhiên để cài đặt cho vecto chuyển động
ParticleArray[i].m_vCurVel = D3DXVECTOR3 (vecX, vecY, vecZ);
// các particle đều có màu xanh lá cây
ParticleArray[i].m_vColor = D3DCOLOR_RGBA (0, 255, 0, 255);
}
```

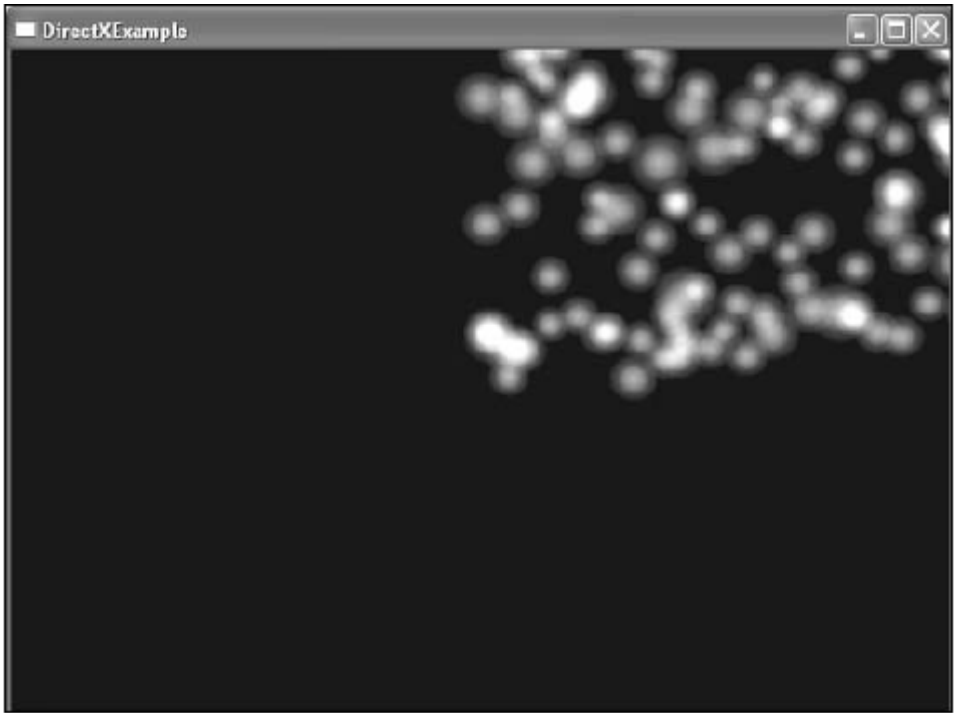
Đoạn code ở trên đầu tiên định nghĩa một cấu trúc để lưu trữ các thuộc tính của particle. Tiếp đó, tạo ra một mảng có cấu trúc trên và đưa vào đó các thông tin cần thiết cho mỗi particle. Sau khi tạo mảng xong, ta thực hiện một vòng lặp qua tất cả các particle và gán giá trị khởi tạo cho vecto vị trí ở gốc tọa độ, gán vecto vận tốc ngẫu nhiên, gán màu cho particle.

Các particle chuyển động như thế nào?

Các particle chuyển động căn cứ vào vecto chuyển động. Vecto này mô tả cả hướng lẫn vận tốc chuyển động của particle. Vecto chuyển động được tạo ra thông qua các lệnh tạo D3DXVECTOR3 với các giá trị X, Y, Z đưa vào.
Đoạn code dưới đây cho thấy cách định nghĩa vecto chuyển động và sự thay đổi của vecto vị trí thông qua việc cộng vecto vị trí và vecto chuyển động với nhau. Đoạn code này sử dụng các biến đã định nghĩa ở phần trên.

```
// tạo vecto chuyển động
Particle.m_vCurVel = D3DXVECTOR3 (0.0f,1.0f,0.0f);
// thay đổi vecto vị trí bằng cách cộng vecto vị trí với vecto chuyển động
Particle.m_vCurPos += Particle.m_vCurVel;
```

Ở dòng code 1 ta đã định nghĩa vecto chuyển động với thành phần Y là 1.0f, do đó khi ta cộng vecto vận tôc với vecto vị trí thì particle sẽ di chuyển lên trên của màn hình một đơn vị theo trục Y.
Ở dòng code 2 cho thấy sự thay đổi vecto vị trí khi ta cộng nó với vecto vận tốc. Thực hiện dòng code này trong mỗi một frame sẽ tạo chuyển động cho particle căn cứ trên vecto vận tốc đã định nghĩa.
Hình 8.1 cho thấy là một nhóm các particle sau khi được phóng ra từ emitter.



Tạo một vecto ngẫu nhiên
Đôi khi, bạn cần phải tạo ra các vecto với hướng chuyển động và vận tốc ngẫu nhiên. Ví dụ như, bạn muốn tạo các particle mà mỗi particle đi theo một hướng khác nhau sau khi được phóng ra từ emitter chẳng hạn. Có một cách để thực hiện điều đó là sử dụng hàm (rand). Hàm này trả về một giá trị ngẫu nhiên nằm trong khoảng từ 0 đến RAND_MAX. Bằng cách ép kiểu nó về dạng float và chia cho RAND_MAX, ta sẽ có một giá trị ngẫu nhiên nằm trong khoảng từ 0.0f đến 1.0f.
Đoạn code sau cho thấy cách tạo một vecto ngẫu nhiên bằng phương pháp này:

```
float vecX = ((float)rand() / RAND_MAX);
float vecY = ((float)rand() / RAND_MAX);
float vecZ = ((float)rand() / RAND_MAX);
D3DXVECTOR3(vecX,vecY,vecZ);
```

Phương pháp này chỉ tạo ra các giá trị dương cho các biến vecX, vecY, và vecZ.

Hệ thống particle

Hệ thống particle là một cách nhóm các emitter lại thành một hình thức tiện dụng. Khi bạn nhóm các emitter lại, việc render nhiều particle trong cùng một lúc sẽ dễ dàng hơn. Trong suột quá trình render các thành phần trong game, hệ thống particle sẽ đảm nhận việc vẽ lại toàn bộ các particle, trong khi bạn xử lý vẽ các phần khác trong game.

Thiết kế một hệ thống particle

Thiết kế một hệ thống particle là một quá trình rất đơn giản. Hệ thống particle điều khiển một hoặc nhiều emitter, và các emitter thì lại điều khiển các particle.
Một hệ thống particle bao gồm 3 thành phần:

- Các đối tượng emitter
- Các particle
- Điều hành hệ thống particle

Điều hành hệ thống particle sẽ điều khiển việc tạo ra, chuyển động và cách sử dụng các emitter. Các emitter sẽ kiểm soát một cách thực sự các particle. Emitter có thể cho kích hoạt hoặc dừng các dòng particle, điều khiển hướng và vận tốc của dòng này, và thậm chí là cả mẫu tạo ra các particle. Một điều cuỗi cùng nữa là – particle – là những hình vuông có texture và nó các thuộc tính điều khiển cách thức biểu hiện của nó như sự chuyển động, vị trí và màu sắc.

Chú ý
Một hiệu ứng particle mô tả kiểu hay cách biểu hiện của một nhóm particle. Các vị dụ về hiệu ứng particle là pháo hoa, dòng nước…
Các emitter được tạo ra như là một nơi sinh ra hay điểm gốc cho các particle biểu diễn một hiệu ứng. Các emitter sẽ khởi tạo giá trị cho các thuộc tính của mỗi particle mà nó phóng ra, ví dụ như vị trí, hướng và vận tốc chuyển động ban đầu. Sau khi particle rời khỏi một emitter, các thuộc tính nội tại này sẽ điều khiển cách biểu hiện particle.
Bởi vì tất cả các particle (được sinh ra từ một emitter) thường chia xẻ chung một kiểu texture, nên rất dễ dàng để render chúng. Bạn chỉ cần cài đặt một trang thái texture duy nhất và sau đó render toàn bộ các particle của emitter này.

Các thuộc tính của emitter

Emitter thường chứa một vài thuộc tính điều khiển cách biểu hiện của nó cũng như cách biểu hiện của các particle mà nó phóng ra. Ta đã từng liệt kê một vài thuộc tính cơ bản của một emitter, đó chỉ là một phần của danh sách dưới đây:

- **Ví trí.** Vị trí của emitter. Emitter có thể được đặt ở bất kì đâu trong không gian 3D.

- **Chuyển động.** Không phải tất cả các emitter đều nằm cố định ở một chỗ. Có một vài emitter, ví dụ như các emitter biểu diễn hiệu ứng khói súng thường xuyên phải chuyển động.
- **Texture.** Emitter thường chứa con trỏ tới một texture sử dụng cho các particle mà nó tạo ra.
- **Mảng các particle.** Bạn cần một vùng nhớ trong emitter để chứa các particle. Bạn có thể lưu trữ chúng theo các cách khác nhau.
- **Số các particle.** Bạn nên lưu trữ số lượng các particle mà emitter sinh ra. Điều này thuận tiện cho việc tăng hay giảm số lượng các particle được sử dụng.
- **Các thuộc tính của particle.** Những thuộc tính này là các giá trị dùng để cài đặt thông số mặc định cho các particle.
- **Lực hấp dẫn.** Một vài emitter có thể chịu ảnh hưởng của lực hấp dẫn.

Cấu trúc emitter

Cấu trúc particle nhóm tất cả các thuộc tính của particle. Bằng cách tạo ra một mảng các biến có cấu trúc này, bạn có thể cập nhật và render nhiều particle một cách nhanh chóng.

```
typedef struct
{
// vecto vị trí
D3DXVECTOR3 m_vCurPos;
// vecto chuyển động
D3DXVECTOR3 m_vCurVel;
// texture dùng cho các particle sinh ra từ emitter này
LPDIRECT3DTEXTURE9 m_texture;
// mảng các phần tử có cấu trúc particle
Particle m_aParticles [MAXNUM_PARTICLES];
// số particle mà emitter sử dụng
Int m_NumParticles;
// cờ hoạt động
BOOL m_bAlive;
} Emitter;
```

Cấu trúc emitter chứa các thuộc tính nội tại của một emitter. Thành phần thứ nhất m_vCurPos, là vị trí hiện tại của emitter. Bởi vì các emitter trên thực tế có thể chuyển động tới bất kì đâu, nên giá trị này có thể bị thay đổi.
Thành phần thứ hai, m_vCurVel, là hướng và vận tốc chuyển động của emitter. Nó quy định chuyển động của emitter.
Thành phần thứ ba là texture được sử dụng cho tất cả các particle của emitter.
Emitter chưa các particle trong một mảng có cấu trúc particle. Biến m_NumParticle là số particle mà emitter đang điều khiển.
Thành phần cuối cùng có kiểu boolean xác định trạng thái hoạt động của emitter.
Bạn có thể tìm thấy các ví dụ minh họa các particle đơn giản và tổng quát trong thư mục chapter8\example1 trên đĩa CD-ROM.

Quản lý hệ thống particle

Bây giờ bạn đã biết những gì cần thiết để tạo một hệ thống particle, chúng ta sẽ đi chi tiết hơn về việc quản lý hệ thống particle.
Đầu tiên bạn cần tạo một lớp particlemanager. Lớp này kiểm soát việc tạo và đặt các emitter.

The Particle Manager Class

Đoạn code dưới đây là header của lớp này.

```
#pragma once
#include <vector>
#include <string>
#include “Emitter.h”
```

```
#include “Particle.h”
// khai báo trước
class Particle;
class Emitter;
class particleManager
{
// các thành phần public
public:
particleManager(void);
~particleManager(void);
bool init(void);
// chấm dứt hệ thống particle
void shutdown(void);
// Tạo một emitter mới
void particleManager::createEmitter(LPDIRECT3DDEVICE9 pDevice,
int numParticles,
std::string textureName,
D3DXVECTOR3 position,
D3DCOLOR color);
// xóa một emitter căn cứ vào chỉ mục của nó
void removeEmitter(int emitterNum);
// xóa một emitter căn cứ vào con trỏ tới nó
void removeEmitter(Emitter *which);
// cập nhật vị trí của emitter và các particle của nó
void update(void);
// render các particle của emitter
void render(LPDIRECT3DDEVICE9 pDevice);
// thành phần private
private:
// vecto của đối tượng emitter
std::vector <Emitter*> emitters;
};
```

Bởi vì số emitter biến đổi liên tục, cho nên ta lưu trữ nó ở dạng vecto. Một vecto có thể tự thay đổi kích thước của nó theo số emitter mà bạn tạo ra; Nó không giới hạn về số lượng như với mảng tĩnh.

Dưới đây là các hàm quan trọng trong lớp này:

- **createEmitter**. Hàm này tạo ra một emitter mới. Các đối số của nó cho phép bạn chỉ định vị trí, sự chuyển động, màu, texture cho emitter này.
- **removeEmitter**. Hàm này cho phép bạn xóa một emitter.

- **update**. Hàm này sẽ gọi hàm update của emitter. Khi hàm update này được gọi thì vị trí của các particle và có thể cả của emitter được thay đổi và tạo ra sự chuyển động.
- **render**. Hàm này sẽ gọi hàm render của emitter, nó đảm nhận vẽ tất cả các particle chứa trong emitter này.

Đoạn code sau là cho hàm createEmitter:

```
/*****************************************************************************
* createEmitter
*****************************************************************************/
void particleManager::createEmitter(LPDIRECT3DDEVICE9 pDevice,
int numParticles,
std::string textureName,
D3DXVECTOR3 position,
D3DCOLOR color)
{
```

```
// tạo một emitter mới
Emitter *tempEmitter = new Emitter(pDevice);
// nạp texture
tempEmitter->addTexture(textureName);
// đặt số lượng particle
tempEmitter->setNumParticles(numParticles);
tempEmitter->initParticles(position, color);
// thêm emitter này vào vecto
emitters.push_back(tempEmitter);
}
```

Hàm createEmitter đầu tiên sẽ tạo ra một con trỏ tới đối tượng emitter gọi là tempEmitter. Tiếp theo, sẽ truyền tên của texture từ textureName vào cho đối tượng emitter vừa tạo, và sau đó là số lượng particle .
Sau đó, hàm initParticle được gọi với vị trí của emitter và màu mặc định của các particle. Và lớp emitter sẽ thực hiện công việc tạo ra và cài đặt cho các particle của nó.
Cuối cùng, emitter mới sẽ được thêm vào vị trí sau cùng của vecto emitter.
Giờ đây, khi ta đã tạo được emitter và cài đặt cho các particle của nó, câu hỏi tiếp theo là: làm thế nào để cập nhật thông tin cho các particle?. Câu trả lời là, lớp particleManager sẽ gọi hàm update của emitter. Hàm update như ở dưới đây, sẽ thực hiện duyệt qua tất cả các emitter trong vecto emitter, kiểm tra các emitter ở trạng thái hoạt động và gọi hàm update của các emitter này.

```
/*****************************************************************************
* update
*****************************************************************************/
void particleManager::update(void)
{
// duyệt qua các emitter
for (unsigned int i=0; i<emitters.size(); i++)
{
// kiểm tra cờ hoạt động
if (emitters[i]->getAlive())
// nếu ở trang thái hoạt động thì gọi hàm update
emitters[i]->update();
}
}
```

Thay vì lưu trữ số lượng emitter đang lưu trữ trong vecto, chúng ta sử dụng hàm size của vecto. Khi các emitter được thêm vào hoặc xóa đi, kích thước của vecto sẽ được thay đổi. Sử dụng hàm size, ta sẽ không cần bận tâm đến điều này.
Hàm update duyệt qua tất cả các emitter chứa trong vecto và gọi hàm update của chúng. Sau khi các particle được cập nhật, bạn cần phải đưa nó ra màn hình. Việc này được thực hiện qua hàm render như dưới đây:

```
/*****************************************************************************
* render
*****************************************************************************/
void particleManager::render(LPDIRECT3DDEVICE9 pDevice)
{
// duyệt các emitter
for (unsigned int i=0; i<emitters.size(); i++)
{
// kiểm tra cờ hoạt động
if (emitters[i]->getAlive())
```

```
// thực hiện render
emitters[i]->render();
}
}
```

Again, I'm looping through the vector of emitters, checking for active emitters. When an active emitter is found, its render function is called. This results in all the particles within an emitter being rendered to the screen.

Tạo một lớp emitter

Lớp emitter gói gọn toàn bộ các chức năng cần thiết cho một emitter. Bên cạnh việc chứa các thuộc tính của emitter, lớp emitter kiểm soát việc nạp và lưu trữ texture sử dụng cho các particle. Mỗi emitter chứa một texture sử dụng cho các particle mà nó sinh ra.
File header dưới đây cho thấy cách tạo lớp emitter:

```
#pragma once
#include <string>
#include <vector>
#include <d3d9.h>
#include <d3dx9tex.h>
#include "Particle.h"
class Particle;
class Emitter
{
// khai báo cấu trúc vecto sử dụng cho particle
struct CUSTOMVERTEX
{
D3DXVECTOR3 psPosition;
D3DCOLOR color;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
public:
Emitter(void);
Emitter(LPDIRECT3DDEVICE9 pDevice);
~Emitter(void);
// thêm 1 texture cho emitter
void addTexture(std::string textureName);
// cài đặt số lượng particle và kích thước của vecto
void setNumParticles(int nParticles);
// cài đặt cho particle và vị trí của emitter
void initParticles(D3DXVECTOR3 position, D3DCOLOR color);
// cập nhật cho toàn bộ particle trong emitter
void update(void);
// render các particle trong emitter
void render();
// hàm inline
inline bool getAlive(void) { return m_bAlive; }
// hàm inline
inline DWORD FLOAT_TO_DWORD( FLOAT f ) { return *((DWORD*)&f); }
private:
// lưu trữ một Direct3D device để sử dụng cho các bước sau
LPDIRECT3DDEVICE9 emitterDevice;
// vị trí hiện tại của particle
D3DXVECTOR3 m_vCurPos;
```

```
// hướng và vận tốc của particle
D3DXVECTOR3 m_vCurVel;
// vertex buffer chứa point sprites
LPDIRECT3DVERTEXBUFFER9 pVertexBuffer;
// texture được sử dụng cho particle
LPDIRECT3DTEXTURE9 pTexture;
// con trỏ dạng particle dùng để tạo ra mảng các particle
Particle *m_particles;
// số particle trong emitter
int numParticles;
// cờ hoạt động của emitter
bool m_bAlive;
// hàm tạo một vertex buffer chứa các particle
LPDIRECT3DVERTEXBUFFER9 createVertexBuffer(unsigned int size,
DWORD usage,
DWORD fvf);
};
```

Lớp emitter định nghĩa một vài hàm cần thiết:

- **addTexture**. Hàm này nạp texture sử dụng cho các particle và lưu trong.
- **setNumParticles**. Mảng particle được thay đổi kích thước trong hàm này cho phù hợp với số lượng particle mà emitter cần dùng.
- **initParticles**. Hàm này tạo ra vertex buffer và các thuộc tính nội tại của các particle.
- **createVertexBuffer**. Hàm địa phương này phát sinh các vertex buffer chứa toàn bộ các particle.
- **Update**. Hàm này điều khiển việc chuyển động của các particle.
- **Render**. Hàm này điều khiển việc render các particle.

Ba hàm quan trọng nhất là initParticles, update, and render. Ta sẽ tìm hiểu kĩ hơn về chúng trong phần tiếp theo:

```
/*****************************************************************************
* initParticles
*****************************************************************************/
void Emitter::initParticles(D3DXVECTOR3 position, D3DCOLOR color)
{
// tạo ra vertex buffer cho emitter và lưu trữ trong biến pVertexBuffer
pVertexBuffer = createVertexBuffer(numParticles * sizeof(CUSTOMVERTEX),
D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY | D3DUSAGE_POINTS,
D3DFVF_CUSTOMVERTEX);
// duyệt qua tất cả các particle của emitter và cài đặt các thuộc tính cho chúng
for (int i=0; i<numParticles; i++)
{
// đặt cờ hoạt động
m_particles[i].m_bAlive = true;
// đặt màu cho particle do  particleManager truyền vào
m_particles[i].m_vColor = color;
// đặt vị trí cho particle do  particleManager truyền vào
m_particles[i].m_vCurPos = position;
// tạo các thành phần vecto chuyển động một cách ngẫu nhiên
float vecX = ((float)rand() / RAND_MAX);
float vecY = ((float)rand() / RAND_MAX);
float vecZ = ((float)rand() / RAND_MAX);
m_particles[i].m_vCurVel = D3DXVECTOR3(vecX,vecY,vecZ);
}
}
```

Dòng đầu tiên trong initParticles là lời gọi tới hàm tạo vertex buffer của emitter. Bởi vì vertex buffer cần được cập nhật thường xuyên, nên nó được tạo ra với các cờ là D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY.
Tiếp theo, một vòng lặp duyệt qua tất cả các particle chứa trong emitter và thiết lập cờ hoạt động cho cúng, thiết lập màu, và vị trí ban đầu. Thông thường, các particle của một emitter thường xuất hiện ở cùng một vị trí.
Hướng và vận tốc của particle được thiết lập một cách ngẫu nhiên. Điều này khiến cho mỗi particle chuyển động về các hướng khác nhau.
Hàm update dưới đây, cập nhật vị trí cho từng particle mỗi frame:

```
/*****************************************************************************
* update
*****************************************************************************/
void Emitter::update(void)
{
// duyệt qua để cập nhật vị trí cho các particle
for (int i=0; i<numParticles; i++)
{
// cộng vecto vị trí với vecto vận tốc
m_particles[i].m_vCurPos += m_particles[i].m_vCurVel;
}
}
```

Vị trí của mỗi particle được thay đổi bằng cách cộng vecto vận tốc (bao gồm cả hướng và độ lớn) với vecto vị trí hiện tại. Bởi vì giá trị này được cập nhật trên mỗi frame, nên ta thu được các particle chuyển động trên màn hình.
Hàm cuối cùng là render:

```
/*****************************************************************************
* render
*****************************************************************************/
void Emitter::render()
{
CUSTOMVERTEX *pPointVertices;
// Khóa vertex buffer và cập nhật các particle trong đó
pVertexBuffer->Lock( 0,
numParticles * sizeof(CUSTOMVERTEX),
(void**)&pPointVertices,
D3DLOCK_DISCARD );
// duyệt qua các particle
for( int i = 0; i < numParticles; ++i )
{
pPointVertices->psPosition = m_particles[i].m_vCurPos;
pPointVertices->color = m_particles[i].m_vColor;
pPointVertices++;
}
// Mở khóa vertex buffer
pVertexBuffer->Unlock();
// thiết lập texture
emitterDevice->SetTexture( 0, pTexture );
// thiết lập luồng vertex
emitterDevice->SetStreamSource( 0, pVertexBuffer, 0, sizeof(CUSTOMVERTEX) );
// thiết lập định dạng vertex
emitterDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
```

```
// gọi hàm DrawPrimitive để render các particle
emitterDevice->DrawPrimitive( D3DPT_POINTLIST, 0, numParticles );
}
```

Hàm render đầu tiên sẽ khóa vertex buffer và duyệt qua mảng các particle của emitter, sao chép dữ liệu từ đó vào buffer. Sau đó, nó mở khóa vertex buffer và thiết lập texture sử dụng cho các particle thông qua hàm SetTexture. Tiếp theo, nó thiết lập nguồn luồng và định dạng vertex trước khi gọi tới hàm DrawPrimitive. Bạn có thể thấy rằng ta đã sử dụng chế độ vẽ D3DPT_POINTLIST, nó có nghĩa là ta render các particle ở dạng các điểm không nối với nhau.

Tạo lớp particle

Lớp cuối cùng cần thiết cho hệ thống particle là lớp Particle. Bởi vì lớp emitter kiểm soát hầu hết các hoạt động của particle, nên lớp particle thực ra chỉ dùng để lưu trữ dữ liệu. File header của lớp này như sau:

```
#pragma once
#include <d3d9.h>
#include <d3dx9tex.h>
class Particle
{
public:
Particle(void);
~Particle(void);
// vecto vị trí của particle
D3DXVECTOR3 m_vCurPos;
// vecto vận tốc của particle
D3DXVECTOR3 m_vCurVel;
// màu của particle
D3DCOLOR m_vColor;
// cờ hoạt động
bool m_bAlive;
};
```

Ta đã đặt toàn bộ các thuộc tính của particle ở dạng public cho nên chúng có thể được truy cập tự do từ emitter. Bởi vì số particle có thể lên đến hàng nghìn, nên việc đặt các thuộc tính ở dạng public làm cho giảm bớt việc quá tải của hàm getter và setter.

Point Sprites: giúp cho particle trở lên dễ dàng hơn

Khái niệm particle ta đã được học ở trên là căn cứ vào billboard, đó là một hình phẳng có texture luôn hướng mặt về phía camera. Mỗi particle được tạo nên theo cách trên đòi hỏi hai tam giác. Để giảm thiểu khối lượng vẽ cho mỗi particle, DirectX đưa ra khái niệm point sprites. Một point sprites có thể coi như như một điểm(point) nói chung, nó có các tọa độ X, Y, Z. Nhưng nó khác những điểm thông thường ở chỗ nó có texture và có kích thước thay đổi. Point sprites có ưu điểm hơn hẳn so với particle (sử dụng chế độ billboard). Trong khi billboard đòi hỏi một quá trình biến đổi tọa độ để có thể hướng mặt về camera thì point sprites mặc định đã luôn hướng về camera.

Sử dụng Point Sprites trong Direct3D

Sự khác biệt lớn nhất giữa việc sử dụng billboard cho các particle với dùng point sprite là chế độ render. Billboard đòi hỏi render hai tam giác nên nó dùng chế độ vẽ triangle strip với bốn vecto. Point sprites được render ở chế độ các điểm, do đó nó giảm thiểu lượng dữ liệu cần render.

Đoạn code sau cho thấy cách gọi hàm DrawPrimitive với point sprite.

```
emitterDevice->DrawPrimitive( D3DPT_POINTLIST, 0, 100 );
```

Lời gọi tới DrawPrimitive sử dụng chê độ vẽ D3DPT_POINTLIST để render 100 particle đang sử dụng.

Cách sử dụng Point Sprites

Point sprite chỉ khác phần bạn học ở trên 1 chút thôi. Để bạn có khái niệm về cách dùng point sprite, chúng ta sẽ đi chi tiết tưng bước một ở dưới đây:
1. Nạp texture dùng cho point sprite thông qua hàm D3DXCreateTextureFromFile.
2. Tạo một vertex buffer động thông qua các cờ D3DUSAGE_DYNAMIC, D3DUSAGE_ WRITEONLY, và D3DUSAGE_POINTS. Chú ý là cờ D3DUSAGE_POINTS sử dụng ở trên sẽ thông báo với Direct3D rằng vertex buffer đang sử dụng chế độ vẽ điểm.
3. Định nghĩa một cấu trúc CUSTOMVERTEX được dùng cho định dang vertex.
Đoạn code sau là một ví dụ về cấu trúc và định dạng đó:

```
struct CUSTOMVERTEX
{
D3DXVECTOR3 psPosition;
D3DCOLOR color;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
```

4. Đến lúc này, ta đã sẵn sàng để render point sprites. Đầu tiên ta cần khóa vertex buffer vừa tạo ở trên và sao chép dữ liệu từ các particle vào đó. Sau khi xong, ta mở khóa cho vertex buffer.
5. Thay đổi trạng thái reder cho thích hợp với point sprite.
6. Gọi hàm DrawPrimitive với tham số D3DPT_POINTLIST.
Những trang thái render gắn liền với point sprite:

- **D3DRS_ALPHABLENDENABLE**. Bật chế độ Alpha blending trong giai đoạn render. Nó tạo cho point sprite có hình dạng tùy ý tùy thuộc vào texture dùng cho nó.
- **D3DRS_ZWRITEENABLE**. Cho phép ứng dụng ghi dữ liệu lên bộ đệm chiều sâu.
- **D3DRS_POINTSPRITEENABLE**. Cho phép sử dụng chế độ texture cho point
- **D3DRS_POINTSCALEENABLE**. Nếu trạng thái này được thiết lập là true, thì các điểm sẽ được thu phóng tùy thuộc vào khoảng cách của nó tới camera.
- **D3DRS_POINTSIZE**. Kích cỡ của point sprite.
- **D3DRS_POINTSIZE_MIN**. Kích thước nhỏ nhất của point sprite.

Như vậy ta đã biết được những công việc cần thực hiên với point sprite, dưới đây sẽ là hàm render thực hiện tất cả các công việc vừa đề cập ở trên:

```
/*****************************************************************************
* render
* sử point sprite để render các particle
*****************************************************************************/
void Emitter::render()
{
emitterDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
emitterDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
emitterDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_ONE );
// thiết lập sử dụng chế độ point sprite
emitterDevice->SetRenderState( D3DRS_POINTSPRITEENABLE, TRUE );
// thiết lập chế độ thu phóng
emitterDevice->SetRenderState( D3DRS_POINTSCALEENABLE, TRUE );
// kích thước vẽ điểm trong trường hợp tham số này không có trong cấu trúc vertex
emitterDevice->SetRenderState( D3DRS_POINTSIZE, FLOAT_TO_DWORD(1.0f) );
// kích thước nhỏ nhất của điểm
emitterDevice->SetRenderState( D3DRS_POINTSIZE_MIN, FLOAT_TO_DWORD(1.0f) );
// 3 trạng thái điều khiển sự thu phóng point sprite
emitterDevice->SetRenderState( D3DRS_POINTSCALE_A, FLOAT_TO_DWORD(0.0f) );
emitterDevice->SetRenderState( D3DRS_POINTSCALE_B, FLOAT_TO_DWORD(0.0f) );
```

```
emitterDevice->SetRenderState( D3DRS_POINTSCALE_C, FLOAT_TO_DWORD(1.0f) );
// Khóa vertex buffer và cài đặt cho point sprite
// cho phù hợp với particle mà ta cần dùng
CUSTOMVERTEX *pPointVertices;
// Khóa vertex buffer để cho phép point sprite có thể di chuyển được
pVertexBuffer->Lock( 0,
numParticles * sizeof(CUSTOMVERTEX),
(void**)&pPointVertices,
D3DLOCK_DISCARD );
// duyệt qua các particle
// copy dữ liệu vào vertex buffer
for( int i = 0; i < numParticles; ++i )
{
pPointVertices->psPosition = m_particles[i].m_vCurPos;
pPointVertices->color = m_particles[i].m_vColor;
pPointVertices++;
}
// mở khóa vertex buffer
pVertexBuffer->Unlock();
// vẽ point sprites
// thiết lập texture dùng cho point sprites
emitterDevice->SetTexture( 0, pTexture );
// đặt luồng vertex
emitterDevice->SetStreamSource( 0,
pVertexBuffer,
0,
sizeof(CUSTOMVERTEX) );
// đặt định dạng vertex
emitterDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
// vẽ point sprites với chế độ D3DPT_POINTLIST
emitterDevice->DrawPrimitive( D3DPT_POINTLIST, 0, numParticles );
// đưa trạng thái render về như ban đầu
emitterDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
emitterDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
emitterDevice->SetRenderState( D3DRS_POINTSPRITEENABLE, FALSE );
emitterDevice->SetRenderState( D3DRS_POINTSCALEENABLE, FALSE );
}
```

Việc đầu tiên mà hàm render thực hiện là thiết lập trạng thái render cần thiết để vẽ point sprite. Tiếp theo, bật chế độ alpha blending và point sprites. Sau đó, nó thiết lập kích thước điểm nhỏ nhất và chế độ thu phóng.
Sau khi đã thiết lập trạng thái render chuẩn, hàm này sẽ cập nhật vertex buffer. Nó khóa buffer và duyệt qua các particle để lấy dữ liệu mới cho vertex bufer.
Sau khi mở khóa cho vertex buffer, point sprite được render lên màn hình. Cuối cùng, hàm sẽ đưa trạng thái render trở về trạng thái mặc định.
Bạn có thể tìm thầy toàn bộ code thể hiện chi tiết cách sử dụng point sprite ở trong thư mục chapter8\example2 trên đĩa CD-ROM.

Chú ý
Các trạng thái render như D3DRS_POINTSIZE và D3DRS_POINTSCALE_A đòi hỏi đưa vào giá trị kiểu DWORD. Bạn có thể đảm bảo điều này bằng cách định nghĩa một hàm inline như sau:

```
inline DWORD FLOAT_TO_DWORD( FLOAT f ) { return *((DWORD*)&f); }
```

Tổng kết chương

Đến lúc này, bạn đã có những kiến thức cơ bản để tạo ra một hệ thống particle. Bằng cách thay đổi các thuộc tính của particle, bạn có thể tạo ra rất nhiều các hiệu ứng particle ấn tương.

Những gì đã được học

Trong chương này bạn đã được học:

- Particle dùng để làm gi
- Hệ thống particle được dùng như thế nào
- Cách tạo và sử dụng emitter
- Sự hữu dụng của point sprites khi làm việc với particle
- Cách render point sprites

Câu hỏi cuối chương

Bạn có thể tìm thấy đáp án cho phần này và phần bài tập tự làm trong phụ lục "Đáp án phần bài tập cuối chương".

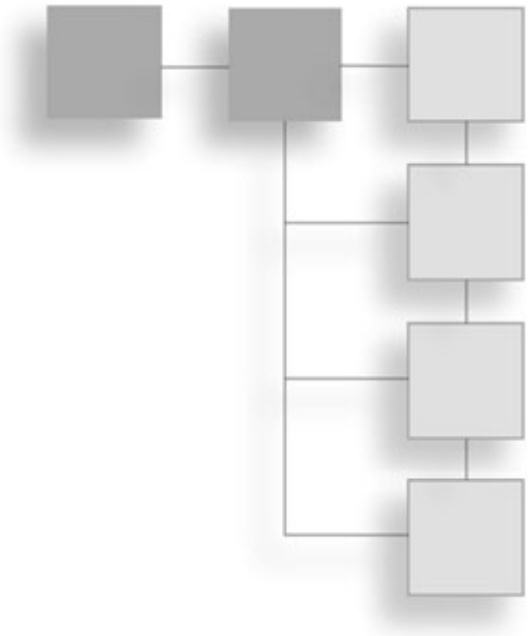
1. Những thuộc tính nào cần thiết để tạo một particle?
2. Ta cần kết hợp 2 thuộc tính nào để làm particle chuyển động?
3. Đối tượng nào được sử dụng để cài đặt các thuộc tính cho particle?
4. Chế độ vẽ nào sử dụng cho DrawPrimitive để vẽ point sprite?
5. Lợi thế của point sprite so với billboard?

Bài tập

1. Lập một hàm update điều khiển cho particle kết thúc sau 300 frame.

2. Tạo một emitter phóng các particle một cách liên tục.

CHƯƠNG 9



SỬ DỤNG DIRECTINPUT

Khả năng tương tác với thế giới ảo là một vấn đề then chốt trong bất kỳ một game nào, việc đó có thể thông qua bàn phím, chuột, hoặc bất kỳ một thiết bị nào. Trong chương này, tôi sẽ giải thích lợi ích của DirectInput và cách sử dụng chúng.

Đây là những vấn đề mà bạn sẽ học trong chương này:

- DirectInput có thể làm cuộc sống của bạn dễ dàng hơn như thế nào.
- Các dạng thiết bị mà DirectInput có thể hỗ trợ
- Phát hiện thiết bị input đang cài đặt hiện tại như thế nào
- Sử dụng bàn phím, chuột và cần điều khiển như thế nào.
- Sử dụng điều khiển tùy biến (analog) hoặc điều khiển số(digital) như thế nào
- Làm thế nào để hỗ trợ nhiều thiết bị input.
- Làm thế nào để sử dụng force feedback.

I Need Input

Tất cả mọi game đều cần khả năng tương tác với người sử dụng chúng. Game của bạn luôn cần cách khai thác điều khiển từ người chơi. Một thiết bị input có thể được sử dụng để điều khiển xe hơi theo nhiều hướng của đường ray, di chuyển đặc tính xung quanh môi trường của nó hoặc bất cứ thứ gì mà bạn có thể tưởng tượng.
Trở về những ngày của DOS, người lập trình viên có sự lựa chọn thật ít ỏi, còn sự thăm dò phần cứng thì bị chặn nếu muốn lấy sự kiện nhấn phím từ bàn phím. Hàm chuẩn trong C của thời kỳ đó như `getchar` rất chậm và không đủ hữu ích cho game. Người ta cần một cách khác tốt hơn. Basic Input Output System (BIOS) được đưa ra là mức phần mềm thấp nhất trong máy tính.
Được lưu giữ trong bộ nhớ ROM trên motherboard, BIOS thông báo cho hệ thống rằng phải khởi động và chuẩn bị phần cứng cho hệ điều hành như thế nào. Trong DOS, người lập trình viên có được truy cập trực tiếp tới BIOS thông qua ngôn ngữ Assemply. Vì BIOS biết tất cả những gì mà phần cứng đã làm, nên những người thiết kế có thể hỏi chúng về những thông tin chính xác. Một

trong những phần quan trọng mà của hệ điều hành mà BIOS luôn quan sát là bàn phím. Mỗi lần nhấn một phím sẽ gây ra một lần ngắt phần cứng và thông báo hệ điều hành được thông báo rằng có một phím đã nhấn. Vì việc này hầu như sảy ra ngay lập tức nên một phương thức nhận sự kiện nhấn phím nhanh và hiệu quả từ bàn phím đã có hiệu lực.
Window NT đã loại trừ khả năng đọc bàn phím trực tiếp từ phần cứng. Window đã trở thành một ranh giới tuyệt đối giữa phần mềm ứng dụng và phần cứng. Bất kỳ một thông tin cần thiết về hệ thống đều phải được lấy từ hệ điều hành vì các ứng dụng không còn được cho phép truy cập trực tiếp tới phần cứng nữa. Window có cách riêng lấy dữ liệu vào từ người sử dụng thông qua hàng đợi thông điệp. Bạn đã nhìn thấy hàng đợi thông điệp trong sách ở phần trước:

```
MSG msg;
ZeroMemory (&msg,  sizefo(msg));
While(msg.message!=WM_QUIT)
{
      //kiểm ta thông điệp
      if( PeekMessage(&msg, NULL, OU, OU, PM_REMOVE))
      {
            TranslateMessage(&msg);
            DispatchMesage(&msg);
      }
}
```

Hàng đợi thông điệp thu thập các sự kiện như dịch chuyển chuột và bàn phím đưa vào từ hệ thống. Mặc dù phương thức này đủ đáp ứng cho ứng dụng Windows, nhưng nó không đủ nhanh cho game. Hầu hết các nhà thiết kế đều quay trở về các hàm khác của Window như `GetAsyncKeyState` – là hàm lấy thông tin mà họ cần.
`GetAsyncKeyState` cho phép kiểm tra các phím trên bàn phím, thậm chí nó còn cho phép kiểm tra các tổ hợp phím và trạng thái của sự kiện nhấn chuột. Phương thức tập trung input của người sử dụng đã trở thành phổ biến trong thiết kế game, nhưng có một vấn đề lớn: không cho phép input được thu thập từ các thiết bị khác như gamepads hoặc cần điều khiển. Người làm game bị cản trở do sự hỗ trợ chỉ dành riêng cho một số thiết bị vì mỗi thiết bị thường có một phương pháp riêng biệt để chọn lọc và chuyển đổi dữ liệu input cho hệ thống.
Cách lấy input một cách nhanh chóng từ người sử dụng theo một tiêu chuẩn là rất cần thiết, bất chấp phương thức hoặc thiết bị được sử dụng đó là gì. DirectInput đã cung cấp một lớp phổ biến cần thiết để giải quyết vấn đề này. DirectInput cho phép game của bạn hỗ trợ vô số các thiết bị vào mà không bắt buộc bạn phải biết chính xác các detail của mỗi thiết bị. Một số ví dụ nhỏ về thiết bị được hỗ trợ bởi DirectInput:

- Bàn phím
- Chuột
- Gamepads
- Cần điều khiển
- Tay lái

Sử dụng DirectInput

DirectInput cũng giống như các thành phần khác của DirectX, nó khởi tạo theo cùng một kiểu như các thành phần khác của DirectX. Nó cần tạo cả DirectInput object và Input Device.
DirectInput Object cung cấp giao diện cần thiết để truy cập vào DirectInput Device. Qua giao diện này, bạn có thể tạo các Device, liệt kê Device ở trong hệ thống hoặc kiểm tra trạng thái của một Device riêng biệt.

Sau khi bạn đã tạo DirectInput Object, bạn phải tạo Device cho chúng. DirectInput Device mà bạn tạo sẽ cho phép bạn lấy truy cập một cách nhanh chóng tới một Input Device, đó có thể là bàn phím, chuột, cần điều khiểm, hoặc là các thiết bị khác đành cho game.
Sau khi tạo Device, bạn cần phải truy cập tới Input của nó. Việc này được thực hiện thông qua quá trình gọi "acquiring a device". Khi bạn có được một Device, bạn có thể khởi động thiết bị, lấy danh sách các khả năng của chúng, hoặc là đọc dữ liệu vào của chúng.
Dường như là chúng ta gặp phải vấn đề khi lấy sự kiện nhấp đôi phím từ bàn phím hoặc cần điều khiển, nhưng khi có được truy cập trực tiếp tới Input Device sẽ giúp hoạt động của chúng ta đơn giản đi rất nhiều.
Bây giờ bạn đã có truy cập tới thiết bị Device, bạn có thể đọc dữ liệu vào từ chúng cho mỗi trạng thái. Ví dụ như nếu bạn đang sử dụng gamepad như một Input Device, thì bạn có thể kiểm tra để biết được người chơi đã nhấn những nút gì. Như vậy bạn có thể làm việc trên những thông tin này.
Ở đây, bạn nên hiểu biết một cách rõ ràng về việc xây dựng một DirectInput, khởi chạy và lấy dữ liệu từ Input Device. Bây giờ tôi sẽ từng bước dẫn dắt bạn qua những bước cần thiết để làm điều này.

Tạo DirectInput Object

Như tôi đã nói trước, bước đầu tiên để sử dụng DirectInput là phải tạo DirectInput object. Hàm *DirectInput8Create* sẽ tạo DirectInput object. Hàm này được định nghĩa như sau:

```
HRESULT WINAPI DirectInput8Create(
      HINSTANCE hInst,
      DWORD  dwVersion,
      REFIID riidltf,
      LPVOID *ppvOut,
      LPUNKNOWN punkOuter
);
```

Đây là 5 tham số được truyền cho hàm:

- hInst – trường hợp ứng dụng tạo DirectInput object.
- dwVersion – số phiên bản của DirectInput mà ứng dụng này cần. Giá trị chuẩn của tham số này là *DIRECTINPUT_VERSION*.
- riidltf – định dạng của giao diện cần thiết. Giá trị mặc định như sau *IID_Idirectinput8* được áp dụng cho tham số này.
- ppvOut – con trỏ trỏ tới biến chứa DirectInput object đực tạo.
- punkOuter – tham số này thường lấy giá trị NULL.

Sau đây là một đoạn trich nhỏ về tạo DirectInput Object:

```
HRESULT hr;                    // biến dùng để lưu giá trị trả về
LPDIRECTINPUT8 DI_Object;      //DirectInput object

//tạo DirectInput Object
hr= DirectInput8Create( hInst,
                        DIRECTINPUT_VERSION,
                        IID_IdirectInput8,
                        (void**) & DI_Object,
                        NULL);
```

```
//kiểm tra giá trị trả  về
if FAILED (HR)
      return false;
```

chú ý:
Xin nhắc lại các bạn là hãy kiểm tra giá trị trả về khi bạn tạo đối tượng DirectX. Việc này thông báo cho bạn khi một đối tượng tạo không thành công thì bạn theo dõi được lỗi trong chương trình.

Ở trong đoạn trước bạn đã tạo hai biến *hr* và *DI_Object*. *hr* có kiểu chuẩn *HRESULT*. Nó kiểm tra giá trị trả về của hàm đươc gọi. Biến thứ hai là *DI_Object* sẽ giữ DirectInput Object được tạo.
Chương trình tiếp tục bằng việc gọi hàm *DirectInput8Create.* Sau đó là kiểm tra nhanh giá trị trả về của *hr* được thực hiện để chắc chắn rằng hàm thực hiện thành công.

Tạo DirectInput Device

Bây giờ bạn đã có một DirectInput object hợp lệ, bạn sẽ dễ dàng tạo Device bằng việc sử dụng hàm *CreateDevice.*

```
HRESULT CreateDevice(
      REFGUID rguid,
      LPDIRECTINPUTDEVICE * lplpDirectInputDevice,
      LPUNKNOWN pUnkOuter
);
```

Hàm *CreateDevice* cần3 tham số:

- rguid – biến giữ tham chiếu tới kiểu *GUID* của Input Device theo yêu cầu. Giá trị này có thể là một trong hai kiểu sau : hàm *EnumDevices* trả về giá trị *GUID*, hoặc là một trong hai giá trị mặc định sau:
  - GUID_SysKeyboard
  - GUID_SysMouse
- lplpDirectInputDevice – biến giữ giá trị trả về của DirectInput Device khi tạo chúng.
- pUnkOuter – địa chỉ của điều khiển giao diện Object. Giá trị này thường là NULL

Đoạn chương trình sau cho bạn thấy rằng có thể tạo DirectInput Device cho hệ thống bàn phím đã cài đặt.

```
HRESULT hr;       //biến dùng để lưu gia trị trả về của hàm
LPDIRECTINPUTDEVICE DI_Device;      //DirectInput Device

//lấy một con trỏ tới giao diện IdirectInputDevice8
hr-DI_object->CreateDevice(GUID_SysKeyboard, &DI_Device, NULL);
//kiểm tra giá trì trả về của hàm CreateDevice
if FAILD (hr)
      return false;
```

Đoạn chương trình này trước tiên tạo biến *DI_Device*. Biến này có kiểu *LPDIRECTINPUTDEVICE* dữ DirectInput Device được tạo.

Việc gọi hàm *CreateDevice* là một phương thức sẵn có trong DirectInput Object, giá trị *GUID_SysKeyboard* được gán cho tham số đầu tiên. Việc gán này thông báo cho *CreateDevice* biết rằng bạn muốn tạo một thiết Device dựa trên hệ thống bàn phím. Tham số thứ hai là biến *DI_Device* mà đã được khai báo trước đó, và tham số thứ 3 là NULL.
Sau khi việc gọi hàm này kết thúc, biến *DI_Device* sẽ lưu DirectInput Device hợp lệ. Để chắc chắn có một Device hợp lệ ta nên kiểm tra giá trị trả về của hàm.

Thiết lập định dạng dữ liệu

Sau khi bạn đã tạo một DirectInput Device hợp lệ, bạn cần phải xây dựng định dạng dữ liệu mà DirectInput sẽ sử dụng để đọc dữ liệu vào từ Device. Hàm *SetDateFormat* đã được định nghĩa là một hàm có một tham số duy nhất kiểu *DIDATAFORMAT*.

```
HRESULT SetDataFormat(
      LPCDIDATAFORMAT lpdf
);
```

Cấu trúc *DIDATAFORMAT* mô tả thiết bị khác nhau trong DirectInput. Cấu trúc *DIDATAFORMAT* đuoc định nghĩa như sau:

```
typedef struct DIDATAFORMAT {
DWORD dwSize;
DWORD dwObjSize;
DWORD dwFlags;
DWORD dwDataSize;
DWORD dwNumObjs;
LPDOBJECTDATAFORMAT rgodf;
} DIDATAFORMAT *LPDIDATAFORMAT;
```

cấu trúc *DIDATAFORMAT* được mô tả trong bảng 9.1

bảng 9.1 cấu trúc DIDATAFORMAT

thành phần	ý nghĩa
dwSize	kích thước của cấu trúc được tính theo bytes
dwObjSize	kích thước của *DIOBJECTDATAFORMAT* tính theo bytes
dwFlags	một gía trị kiểu *DWORD* mà chỉ ra đặc tính của định dạng dữ liệu. Các giá trị hợp lệ: *DIDF_ABSAXIS* có nghĩa là các trục là một giá trị tuyệt đối, hoặc *DIDF_RELAXIS* có nghĩa là các trục của Device này tương đối.
dwDataSize	giá trị này lưu giữ kích thước của túi dữ liệu trả về từ Input Device được tính theo byte
dwNumObjs	số Object trong ma trận rgodf
rgodf	địa chỉ tới ma trận có cấu trúc *DIOBJECTDATAFORMAT*

Bạn cần phải tạo và sử dụng cấu trúc *DIDATAFORMAT* nếu như Input Device mà bạn muốn sử dụng không phải là một thiết bị chuẩn. Sau đây là cấu trúc *DIDATAFORMAT* được định nghĩa trước cho các Input Device thông dụng:

- c_dfDIKeyboard – đây là một cấu trúc định dạng dữ liệu dùng để mô tả hệ thống đối tượng bàn phím.
- c_dfDIMouse – bạn sử dụng cấu trúc định dạng dữ liệu này khi Input Device được sử dụng là chuột cùng với 4 nút bấm.
- c_dfDIMouse2 – bạn sử dụng cấu trúc dữ liệu này khi Input Device được sử dụng là chuột hoặc là thiết bị tương tự cùng với 8 nút bấm.
- c_dfDIJoystick – đây là cấu trúc định dạng dữ liệu dành cho cần điều khiển.
- c_dfDIJoystick2 – đây là cấu trúc định dạng dữ liệu dành cho cần điều khiển với những chức năng mở rộng.

Nếu như Input Device mà bạn muốn sử dụng không nằm trong những dạng đã được xác định trước, bạn cần phải tạo riêng một cấu trúc kiểu `DIDATAFORMAT`. Hầu hết các Input Device thông dụng không cần làm việc này.
Ví dụ chỉ ra dưới đây thực hiện việc gọi hàm `SetDataForm` sử dụng cấu trúc đã định nghĩa trước `DIDATAFORMAT` dành cho thiết bị bàn phím.

```
//biến giữ giá trị trả về
HRESULT hr;
// lấy định dạng dữ liệu cho thiết bị
// gọi hàm SetDataFormat
hr = DI_Device->SetDataFormat(&c_dfDIKeyboard);
//kiểm tra giá trị trả về của hàm
if FAILED (hr)
      return false;
```

Thiết lập Cooperative Level

Cooperative Level thông báo cho hệ thống biết Input Device mà bạn tạo làm việc với hệ thống như thế nào. Bạn có thể thiết lập Input Device để sử dụng một trong hai kiểu truy cập : *Truy cập độc quyền* hoặc *truy cập không độc quyền*.
Truy cập độc quyền có nghĩa là chỉ ứng dụng của bạn có thể sử dụng một thiết bị riêng biệt và không cần chia sẻ nó cho các ứng dụng khác của Windows. Việc này hữu ích nhất khi game của bạn là một ứng dụng chạy tràn màn hình. Khi game là một thiết bị sử dụng độc quyền ví dụ như chuột hoặc bàn phím thì bất kỳ một ứng dụng nào khác sử dụng thiết bị này đều bị lỗi.

Nếu game của bạn không nhất thiết phải cản trở việc chia sẻ thiết bị thì ta gọi đây là truy cập không độc quyền. Khi game tạo thiết bị với truy cập không độc quyền, các ứng dụng khác đang chạy có thể sử dụng cùng một thiết bị. Kiểu dùng này hữu ích nhất khi game của bạn chạy ở chế độ một cửa sổ của Windows, tức không tràn màn hình. Sử dụng chuột như một Input Device truy cập không độc quyền không giới hạn việc sử dụng nó trong các ứng dụng khác của Windows.
Đối với mỗi game mà bạn muốn sử dụng DirectInput Device, bạn phải thiết lập Cooperative Level để sử dụng nó. Bạn làm việc này thông qua hàm `SetCooperativeLevel` được định nghĩa dưới đây:

```
HRESULT SetCooperativeLevel(
HWND hwnd,
DWORD dwFlags
);
```

hàm `SetCooperativeLevel` có hai tham số:

- hwnd – một điều khiển tới cửa sổ mà yêu cầu truy cập tới Device.
- dwFlags – một chuỗi cờ hiệu mô tả kiểu truy cập mà bạn đang yêu cầu. Có những kiểu cờ hiệu sau:
  - DISCL_BACKGROUND – ứng dụng yêu cầu truy cập background tới thiết bị. Điều này có nghĩa là bạn có thể sử dụng Input Device thậm trí trong trường hợp cửa sổ của game hiện tại không được kích hoạt.
  - DISCL_EXCLUSIVE – game yêu cầu kiểm tra toàn bộ và kiểm tra hoàn tất các Input Device, giới hạn các ứng dụng khác sử dụng chúng.
  - DISCL_FORGROUND – game yêu cầu Input chỉ khi cửa sổ game đang bị kích hoạt trên màn hình. Nếu cửa sổ game đánh mất tiêu điểm, Input tới cửa sổ này tạm thời bị ngưng hoạt động.
  - DISCL_NONEXCLUSIVE – *truy cập độc quyền* không cần thiết cho ứng dụng này. Sự xác định cờ hiệu này cho phép các ứng dụng khác đang hoạt động tiếp tục sử dụng thiết bị này.
  - DISCL_NOWINKEY – cờ hiệu này thông báo cho DirectInput vô hiệu hóa các phím Windows trên bàn phím. Khi những phím này được nhấn, thì Start Button trên màn hình đã kích hoạt phải chuyển tiểu điểm đến cửa sổ đang kích hoạt. Khi cờ hiệu này được chọn, phím Windows mất hiệu lực, cho phép game của bạn trở thành tiêu điểm.

Chú ý:
Mỗi ứng dụng phải chỉ rõ là cần truy cập background hay là truy cập foreground tới Device bằng thiết lập một trong hai cờ hiệu: `DISCL_BACKGROUND` hoặc `DISCL_FOREGROUND`. Ứng dụng cũng cần thiết lập một trong hai cờ hiệu: `DISCL_EXCLUSIVE` hoặc là `DISCL_NONEXCLUSIVE`. Cờ hiệu `DISCL_NOWINKEY` là tùy chọn.

Đoạn chương trình sau thiết lập Device để sử dụng *truy cập không độc quyền* và có hiệu lực khi cửa sổ ứng dụng là tiêu điểm.

```
//thiết lập cooperative level
hr = DI_Device->SetCooperativeLevel(wndHandle, DISCL_FOREGROUND |
DISCL_NONEXCLUSIVE );
// Kiểm tra giá trị trả về của hàm SetCooperativeLevel
if FAILED (hr)
      return false;
```

Hàm ***`SetCooperativeLevel`*** là một phương thức mà có thể gọi được thông qua giao diện DirectInput Device. Biến ***`DI_Device`*** trong đoạn chương trình trên biểu diễn DirectInput Device hiện thời được tạo bởi việc gọi hàm ***`CreateDevice`***.
Những tham số mà được truyền trong ví dụ hàm ***`SetCooperativeLevel`*** gồm có ***`wndHandle`*** tương ứng với điều khiển tới cửa sổ đang yêu cầu truy cập tới Input Device, và cờ hiệu ***`DISCL_FORGROUND`*** và ***`DISCL_NONEXCLUSIVE`*** thông báo cho DirectInput kiểu truy cập mà bạn đang cần cho thiết bị.

Lấy truy cập

Bước cần thiết cuối cùng trước khi bạn có thể đọc dữ liệu vào từ thiết bị riêng biệt là gọi *"lấy truy cập"*. Khi bạn lấy truy cập tới thiết bị, bạn sẽ thông báo cho hệ thống biết rằng bạn đã sẵn sàng sử dụng và đọc dữ liệu từ Device này. Hàm này là một phương thức khác của DirectInput

Device mà thực hiện công việc này. Hàm này sẽ được định nghĩa dưới đây, nó không có tham số và nó chỉ trả về khi nó thực hiện thành công.

```
RESULT Acquire (VOID);
//đoạn ví dụ nhỏ sau chỉ cách gọi hàm Acquire
//lấy truy cập tới input device
hr = DI_Device->Acquire();
if  FAILED (hr)
      return false;
```

Giá trị trả về của hàm này được kiểm tra để chắc chắn là hàm đã thực hiện thành công. Vì đây là bước cần thiết cuối cùng trước khi đọc dữ liệu vào từ thiết bị, nên tốt nhất là phải kiểm tra giá trị trả về để chắc chắn rằng thiết bị đã sẵn sàng.

Đọc dữ liệu vào

Bây giờ bạn đã hoàn thành những bước cần thiết để khởi động một Input Device thông qua DirectInput, đây sẽ là thời điểm thực sự để sử dụng nó. Tất cả các Device đều sử dụng hàm `GetDeviceState` khi đọc Input. Input Device có phải là bàn phím, chuột hoặc gampad hay không, hàm `GetDeviceState` được sử dụng như sau:

```
HRESULT GetDeviceState(
      DWORD cbData,
      LPVOID lpvData
);
```

Tham số đầu tiên là giá trị có kiểu `DWORD` dùng để lưu giữ kích thước của bộ nhớ đệm mà bộ nhớ đệm này được dùng cho tham số thứ 2. Tham số thứ hai là một con trỏ trỏ tới vùng nhớ đệm sẽ lưu giữ dữ liệu được đọc từ thiết bị. Như đã nhắc trước, định dạng của dữ liệu từ Input Device được định nghĩa trước khi sử dụng hàm `SetDataFormat`.
Một số bước tiếp theo sẽ chỉ cho bạn thấy cách liệt kê các Input Device hiện có trong ứng dụng của bạn qua DirectInput như thế nào.

Liệt kê Input Device

Hầu hết các game hiện nay trên máy tính đều cho phép sử dụng các Input Device khác ngoài bàn phím và chuột như gamepad hoặc cần điều khiển. Một số máy tính theo mặc định không có những Device không chuẩn (nonstandar) này, vì thế DirectInput không thể chấp đảm đương sự có mặt của chúng. Ngoài ra Windows cho phép nhiều gamepad hoặc là cần điều khiển cài đặt đồng thời, nên DirectInput cần biết cách xác định có bao nhiêu và có những Device nào. Phương thức mà DirectInput sử dụng để lấy những thông tin cần thiết trên các Input Device được gọi là liệt kê.
Chỉ Direct3D có thể liệt kê qua video adapter được cài đặt trong hệ thống và lấy những khả năng của chúng, DirectInput có thể làm những việc này đối với các Input Device.
Sử dụng các hàm hiện có trong DirectInput Object, DirectInput có thể giới hạn số lượng Input Device trong hệ thống. Như thường lệ mỗi một thiết bị có kiểu và chức năng của mình. Ví dụ nếu game của bạn cần sử dụng gamepad với cần điều khiển analog, thi bạn có thể liệt kê và thấy được những Device đã cài đặt nếu như bất kỳ cái nào nằm trong tiêu chuẩn bạn đề ra.

Quá trình liệt kê những Device đã cài đặt trên hệ thống cần tập hợp một danh sách các Device mà Input hợp lệ của bạn cần.
DirectInput sử dụng hàm *EnumDevices* để tập hợp đanh sách các Input Device đã cài đặt. Vì chúng là các dạng thiết bị khác nhau trong máy và hầu như bạn không cần quan tâm đến việc lấy đanh sách của tất cả, nên *EnumDevices* cho phép bạn chỉ ra dạng thiết bị mà bạn đang tìm kiếm. Ví dụ nếu bạn không quan tâm chuột và bàn phím mà bạn chỉ tìm thiết bị khác như cần điều khiển, *EnumDevice* sẽ cung cấp cho bạn cách loại trừ các thiết bị không cần đến trong danh sách.

Trước tiên tôi sẽ giải thích hàm *EnumDevices* được sủ dụng như thế nào. Hàm *EnumDevice* được định nghĩa như sau:

```
HRESULT EnumDevices(
      DWORD dwDevType,
      LPDIENUMDEVICESCALLBACK lpCallback,
      LPVOID pvRef,
      DWORD dwFlags
);
```

hàm này có 4 tham số:

- dwDevType – tham số này thiết lập bộ lọc cho việc tìm kiếm thiết bị. Như tôi đã nói trước, bạn có thể thông báo cho *EnumDevices* là chỉ tìm kiếm dạng thiết bị riêng nào đó. Tham số này có thể sử dụng các giá trị sau:
  - DI8DEVCLASS_ALL – giá trị này làm cho hàm *EnumDevices* trả về danh sách tất cả các Input Device được cài đặt trên hệ thống.
  - DI8DEVCLASS_DEVICE – giá trị này giúp việc tìm kiếm thiết bị không rơi vào những lớp thiết bị khác, ví dụ như bàn phím, chuột, hoặc game controller.
  - DI8DEVCLASS_GAMECTRL – giá trị này giúp cho hàm *EnumDevices* tìm tất cả các game controller Device như gamepad hoặc là cần điều khiển.
  - DI8DEVCLASS_KEYBOARD – *EnumDevices* tìm kiếm trong hệ thống tất cả các thiết bị bàn phím
  - DI8DEVCLASS_POINTER – giá trị này thông báo cho *EnumDevices* tìm kiếm thiết bị con trỏ như chuột.
- lpCallback – *EnumDevice* sử dụng cơ cấu *callback* khi tìm kiếm trong hệ thống các Input Device. Tham số này là dịa chỉ của hàm mà bạn định nghĩa để làm việc như *callback*.
- pvRef – tham số này truyền dữ liệu tới hàm *callback* được xác định trong tham số *lpCallback*. Bạn có thể sử dụng giá trị 32 bit ở đây. Nếu bạn không cần gửi thông tin đến hàm *callback*, bạn truyền cho nó giá trị NULL.
- dwFlags – tham số cuối cùng là giá trị kiểu *DWORD* bao gồm tập hợp các cờ hiệu cho phép *EnumDevices* biết phạm vi liệt kê. Ví dụ nếu bạn muốn *EnumDevices* tìm trong hệ thống chỉ những thiết bị đã cài đặt hoặc là những thiết bị có *force feedback*, bạn cần chỉ ra một trong các giá trị sau:
  - DIEDFL_ALLDEVICES – đây là giá trị mặc định. Tất cả các thiết bị trong hệ thống đều được liệt kê.
  - DIEDFL_ATTACHEONLY – chỉ những thiết bị mà hiện tại gắn với hệ thống được chỉ ra
  - DIEDFL_FORCEFEEDBACK – chỉ những thiết bị mà cung cấp force feedback được chỉ ra.

- DIEDFL_INCLUDEALIASES – Windows cho phép tạo biệt danh cho các Device. Những biệt danh này xuất hiện trong hệ thống như những Input Device, nhưng chúng mô tả Device khác trong hệ thống.
- DIEDFL_INCLUDEHIDEN – giá trị này giúp ***EnumDevices*** chỉ ra các thiết bị ẩn.
- DIEDFL_INCLUDEPHANTOMS – một vài thiết bị phần cứng có nhiều Input Device, ví dụ như bàn phím cũng chứa chuột gắn liền. Giá trị này giúp DirectInput trả về những thiết bị đồng bộ.

Đoạn chương trình sau sử dụng hàm ***EnumDevices*** để gọi danh sách các game controller mà hiện tại đang gắn với hệ thống.

```
HRESULT hr;       //biến sử dụng để lưu giá trị trả về
// gọi hàm  EnumDevices
hr – DI_Object->EnumDevices( DI8DEVCLASS_GAMECTRL,
                                EnumJoysticksCallback,
                                NULL,
                                DIEDFL_ATTACHEONLY
);
// kiểm tra giá trị trả về
if FAILED (hr)
      return false;
```

Đoạn trên gọi hàm ***EnumDevices*** đã sử dụng giá trị ***DI8DEVCALSS_GAMECTRL*** để tìm kiếm game controller. Giá trị ***DIEDFL_ATTACHEONLY*** chỉ tìm kiếm những thiết bị mà đã được gắn với hệ thống. Tham số thứ hai có giá trị là ***EnumJoysticksCallback*** biểu diễn tên của hàm *callback* để tiếp nhận thiết bị tìm thấy.Tham số thứ 3 là NULL vì không có thông tin bổ xung cần thiết để gửi tới hàm *callback*.
Hàm *callback* giúp cho ***EnumDevices*** được gọi trong mọi thời điểm một thiết bị, thiết bị này được tim thấy thỏa mãn chuẩn tìm kiếm. Ví dụ nếu bạn đang tìm kiếm trong hệ thống gamepad và hiện tại có 4 plugged in, hàm *callback* sẽ được gọi 4 lần.
Mục đích của hàm *callback* là cho ứng dụng của bạn cơ hội tạo một DirectInput Device để mỗi thành phần của phần cứng, sau đó bạn có thể quét các khả năng của thiết bị.
Hàm callback phải được xác định trong mã nguồn sử dụng định dạng đặc biệt *DIEnumDevicesCallback*

```
BOOL CALLBACK DIEnumDevicesCallback(
LPCDIVICEINSTANCE lpddi,
LPVOID pvRef
);
```

hàm ***DIEnumDevicesCallback*** cần hai tham số, một con trỏ tới cấu trúc ***LPCDIVICEINSTANCE***, và một giá trị được truyền cho tham số ***pvRef*** của ***EnumDevices***.

Cấu trúc ***LPCDIVICEINSTANCE*** được định nghĩa sau đây sẽ lưu các chi tiết liên quan đến một Input Device, ví dụ như ***GUID*** của chúng và tên sản phẩm của chúng. Thông tin trong cấu trúc rất hữu ích khi hiển thị sự chọn lựa thiết bị tới người sử dụng vì nó cho phép nhận ra một thiết bị dựa vào tên của chúng.

```
Typedef struct DIDEVICEINSTANCE {
DWORD dwSize;
GUID guidInstance;
GUID guidProduct;
DWORD dwDevType;
```

```
TCHAR tszInstanceName[MAX_PATH];
TCHAR tszProductName[MAX_PATH];
GUID guidFFDrive;
WORD wUsagePage;
WORD wUsage;
} DIDEVICEINSTANCE, * LPDIDEVICEINSTANCE;
```

Bảng 9.2 mô tả cấu trúc ***DIDEVICEINSTANCE*** một cách chi tiết

Bảng 9.2 cấu trúc DIDEVICEINSTANCE

Tên thành phần	mô tả
dwSize	kích thước của cấu trúc này tính theo byte
guidInstance	kiểu GUID dành cho thiết bị riêng biệt. Giá trị này có thể được lưu lại và sử dụng sau với hàm ***CreateDevice*** để lấy truy cập tới thiết bị.
guidProduct	định dạng đơn nhất của Input Device. Giá trị này là chỉ số ID sản phẩm cơ bản của thiết bị.
dwDevType	giá trị này là dạng thiết bị chỉ định. Giá trị này có thể là bất kỳ giá trị nào theo lý thuyết trong tư liệu DirectX dành cho cấu trúc này.
tszInstanceName	tên thân thuộc của thiết bị như là Joystick 1 hoặc là AxisPad.
tszProductName	đây là tên sản phẩm đầy đủ của thiết bị này.
guidFFDriver	nếu thiết bị này hỗ trợ force feedback, giá trị này biểu diễn ***GUID*** của driver được sử dụng.
wUsagePage	giá trị này lưu giữ Human Interface Device (HID) usage page code
wUsage	đây là cách sử dụng code cho một HID

Hàm ***DIEnumDevicesCallback*** cần một giá trị kiểu ***BOOLEAN*** để trả về. DirectInput đã xác định 2 giá trị được sử dụng thay cho các giá trị chuẩn ***TRUE*** và ***FALSE*** là:

- DIENUM_CONTINUE – giá trị này thông báo cho liệt kê tiếp tục
- DIENUM_STOP – giá trị này làm cho liệt kê thiết bị dừng lại.

Những giá trị này điều khiển quá trình liệt kê thiết bị. Nếu bạn đang tìm kiếm trong hệ thống chỉ những thiết bị điều khiển, chúng sẽ vô dụng khi liệt kê tất cả các thiết bị điều khiển đã cài đặt. Sự trả về ***DIENUM_STOP*** sau khi tìm kiếm thiết bị thích hợp đầu tiên là những gì chúng ta cần. Thông thường thì bạn sẽ muốn tập hợp lại một danh sách tất cả các thiết bị thích hợp để người sử dụng có thể chọn thiết bị nào mà muốn dùng. Sử dụng cơ cấu callback, bạn có thể tạo DirectInput Device cho mỗi thành phần của phần cứng và đưa chúng vào một danh sách. Người sử dụng có thể chọn thiết bị mà anh ta muốn sử dụng.

Ví dụ sau chỉ ra hàm collback sẽ trả về thiết bị điều khiển được tim thấy mà ***EnumDevices*** gặp đầu tiên:

```
BOOL CALLBACK DeviceEnumCallback (const DIDEVICEINSTANCE* pdidInstance, VOID*
pContext)
{
//biến giữ giá trị trả về
HRESULT hr;
//tạo thiết bị
hr-DI_Object->CreateDevice(pdidInstance->guidInstance, &g_pJoystick, NULL);
//gọi tới hàm CreateDevice bị lỗi thì tiếp tục tìm kiếm khác.
If (FAILED (hr)) return DIENUM_CONTINUE;
      //thiết bị được tìm thấy và hợp lệ thì ngưng quá trình liệt kê
return DIENUM_STOP;
```

```
}
```

Ở đoạn chương trình trên sự thử nghiệm đầu tiên sử dụng hàm `CreateDevice` để truy cập tới thiết bị được truyền cho hàm *callback*. Nếu việc gọi hàm `CreateDevice` bị lỗi, hàm *callback* trả về `DIENUM_CONTINUE`, thông báo cho quá trình liệt kê các thiết bị tiếp tục, nếu việc gọi hàm `CreateDevice` thành công, *callback* trả về giá trị `DIENUM_STOP`.

Bạn có thể tìm thấy ví dụ minh họa liệt kê các thiết bị như thế nào trong hệ thống và hiển thị tên các thiết bị của chúng trong chapter9\example3 trên đĩa CD-ROM đi kèm. Hình 9.1 chỉ ra hộp thoại được tạo trong ví dụ trên:

Thu hoạch các khả năng của thiết bị

Sau khi bạn có một thiết bị hợp lệ trả về từ hàm `EnumDevices`, bạn cần phải kiểm tra các chức năng cơ bản của nó. Ví dụ bạn cần phải tìm dạng force feedback mà Device này có thể hỗ trợ. Liệt kê các khả năng của một Device cũng tương tự như liệt kê các Device. Để lấy những đặc điểm cụ thể của mỗi Device, bạn phải gọi hàm `EnumObjects`. Giống như gọi hàm `EnumDevices`, hàm này làm việc với phương thức callback

```
HRESULT EnumObjects(
LPDIENUMDEVICEOBJECTSCALLBACK lpCallback,
LPVOID pvRef,
DWORD dwFlags
);
```

hàm `EnumObjects` cần 3 tham số:

lpCallback – đây là tên của hàm callback
pvRef – đây là dữ liệu mở rộng sẽ được gửi đến hàm callback khi nó được gọi.
dwFlags – là những cờ hiệu có giá trị kiểu `DWORD`, chúng chỉ rõ các dạng của đối tượng trên Input Device mà bạn quan tâm trong bảng liệt kê.

Bảng 9.3 mô tả các tham số `dwFlags` cụ thể hơn

Tên Flag	mô tả
DIDFT	sử dụng một trục tuyệt đối
DIDFT_ALIAS	tìm kiếm điều khiển đã xác nhận bởi HID bằng cách sử dụng biệt danh
DIDFT_ALL	tìm kiếm tất cả các dạng Object trong thiết bị
DIDFT_AXIS	tìm kiếm một trục: tương đối hoặc tuyệt đối
DIDFT_BUTTON	kiểm tra nút nhấn hoặc nút bật tắt
DIDFT_COLLECTION	danh sách các tập trung liên kết HID
DIDFT_ENUMCOLLECTION	liên hệ tới kết nối tập trung
DIDFT_FFACTUATOR	chứa đựng một phát động force feedback
DIDFT_FFEFFECTTRRIGGER	chứa đựng nút bấm force feedback

DIDFT_NOCOLLECTION	tìm kiếm những đối tượng không liên quan đến một liên kết tập trung
DIDFT_NODATA	không khởi tạo dữ liệu
DIDFT_PDV	tìm kiếm một điều khiển POV
DIDFT_PSHBUTTON	tìm kiếm một nút nhấn
DIDFT_RELAXIS	sử dụng một trục tương đối
DIDFT_TGLBUTTON	tìm kiếm một nút bật tắt
DIDFT_VENDORDEFINED	trả về một đối tượng của một kiểu đã xác định trước

Mục đích của hàm callback *EnumObjects* là thu thập thông tin về thành phần của Input Device. Thông tin này tập hợp lại cho mỗi thiết bị được truyền tới callback như một cấu trúc *DIDEVICEOBJECTINSTANCE*

```
BOOL CALLBACK DIEnumDeviceObjectsCallback(
LPDIDEVICEOBJECTINSTANCE lpddoi,
LPVOID pvRef
);
```

hàm *DIEnumDeviceObjectsCallback* lấy 2 tham số. Tham số thứ nhất là cấu trúc kiểu *DIDEVICEOBJECTINSTANCE* mà giữ thông tin trả về liên quan đến thiết bị. Tham số thứ hai là bất kỳ giá trị nào được truyền cho tham số *pvRef* của hàm *EnumObjects* .
Cấu trúc *DIDEVICEOBJECTINSTANCE* chứa đựng sự giàu có thông tin có giá trị về thiết bị. Nó hữu ích cho việc thiết lập giới hạn của force feedback, cũng như giúp xác định các dạng riêng biệt và chỉ số của điều khiển trên thiết bị.
Bạn có thể tìm thấy giải thích đầy đủ về cấu trúc *DIDEVICEOBJECTINSTANCE* trong tư liệu của DirectInput

Khai thác Input từ bàn phím

Thu hoạch input từ bàn phím là một việc có phần nào đơn giản vì nó là một thiết bị xác lập mặc định. Bàn phím cần có một bộ nhớ đệm chứa 256 phần tử ma trận kí tự.

```
Char buffer[256];
```

Ma trận kí tự này lưu giữ trạng thái của mỗi phím trên bàn phím. Trạng thái của một hoặc nhiều phím có thể được lưu trong ma trận này mỗi khi biết bị bàn phím được đọc. Điều mà hầu hết các game đều cần là Input Device có thể đọc mỗi trạng thái từ trong vòng lặp chính của game.

Trước khi bạn có thể đọc từ bàn phím, bạn cần xác định một bước quan trọng là phím nào trên bàn phím đã nhấn. Macro *KEYDOWN* cung cấp dưới đây trả về *TRUE* hoặc *FALSE* dựa trên cơ sở phím mà bạn đang kiểm tra có được nhấn hay không..

```
#define KEYDOWN (name, key) { name[key]& 0x80}
sau đây là ví dụ đọc từ bàn phím.

//xác định macro cần kiểm tra trạng thái cảu phím trên bàn phím.
#define KEYDOWN (name, key) { name[key]& 0x80}

Đây là bộ nhớ đệm cần thiết của bàn p hím
```

```
Char buffer[256];

//đây là vòng lặp chính đọc từ Input Device của mỗi khung
while(1)
{
//kiểm tra bàn phím và xem  xét phím nào hiện tại đang được nhấn
g_lpDIDevice->GetDeviceState(sizeof (buffer), (LPVOID )&buffer);

//làm việc gì đó với Input

//KEYDOWN macro ở đay kiểm tra phím mũi tên sang trái có được nhấn hay không
if (KEYDOWN(buffer, DIK_LEFT))
{
//làm gì đó vói mũi tên sang trái
}
// KEYDOWN được sử dụng một lần nữa để kiểm tra phím mũi tên lên trên
//có đuoc nhấn hay không
if(KEYDOWN(buffer, DIK_UP))
{
//làm một việc gì đó với phím mũi tên lên trên
}
}
```

Như bạn có thể thấy là vòng lặp chính của game gọi hàm *GetDeviceState* cho mỗi trạng thái và đưa trạng thái hiện tại của bàn phím vào Input Buffer. *KEYDOWN* macro kiểm tra trạng thái của một phím nào đó.
Hình 9.2 chỉ ra một minh họa nhỏ về sử dụng bàn phím đưa vào để hiển thị mũi tên định hướng nào đã đượcc nhấn.
Bạn có thể tìm thấy mã nguồn của ví dụ này trong thư mục chapter9\example1 trên đĩa CD-ROM

Thu dữ liệu vào từ Chuột

Đọc dữ liệu vào từ chuột cũng tương tự như từ bàn phím. Sự khác nhau cơ bản ở đây là *GUID* được gán cho hàm *CreateDevice* và cấu trúc *DIDATAFORMAT* lưu dữ liệu vào của thiết bị này.

hình 9.2 ví dụ minh họa bàn phím.

Ở ví dụ trước, việc gọi hàm *CreateDevice* sử dụng *GUID_SysKeyboard* cho tham số thứ nhất. Khi bạn sử dụng chuột, bạn phải thiết lập *GUID* kiểu *GUID_SysMouse* cho *CreateDevice.*

Chú ý:

Khi thiết lập cooperative level ở chế độ độc quyền cho chuột thì Input ngăn chặn con trỏ Windows hiển thị. Trong chế độ độc quyền, bạn có trách nhiệm phải vẽ con trỏ chuột.

Đoạn chương trình sau chỉ ra cách sử dụng hàm *CreateDevice* như thế nào

```
//gọi hàm CreateDevice sử dụng tham số GUID_SysMouse
hr=g_lpDI->CreateDevice(GUID_SysMouse, &g_lpDiDevice, NULL);

//kiểm tra giá trị trả về cảu hàm CreateDevice
nếu FAILED (hr)
      return FALSE;
```

Việc gọi tới hàm *SetDataFormat* đã sử dụng định dạng dữ liệu định trước *c_dfDIKeyboard*. Bạn phải thay đổi giá trị này thành *c_dfDIMouse* khi bạn sử dụng chuột là Input Device.

```
//thiết lập định dạng dữ liệu cho Chuột
hr= g_lpDIDevice->SetDataFomat(&c_dfDIMouse);

//kiểm tra giá trị trả về cho hàm SetDataFormat
if FAILED (hr)
      return FALSE;
```

Sự thay đổi cuối cùng cần phải làm trước khi bạn đọc từ chuột là bộ nhớ mà được định nghĩa bởi *DIDATAFORMAT*. Bàn phím cần một bộ nhớ ký tự chứa 256 phần tử, ngược lại chuột chỉ cần một buffer có kiểu *DIMOUSESTATE*.
Cấu trúc *DIMOUSESTATE* bao gồm 3 biến để lưu vị trí của chuột là X,Y và Z. Đồng thời nó cần thêm một ma trận kiểu *BYTE* có 4 phần tử để lưu trạng thái của nút bấm chuột. Cấu trúc *DIMOUSESTATE* được định nghĩa như sau:

```
Typedef struct DIMOUSESTATE{
LONG lX;          //lưu khoảng cách mà chuột đã di chuyển trên trục X
LONG lY;          //lưu khoảng cách mà chuột đã di chuyển trên trục Y;
LONG lZ;          //lưu khoảng cách mà chuột đã di chuyển trên trục Z;
BYTE rgbButtons[4];     //trạng thái hiện tại của các nút nhấn chuột.
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Phần trước, một macro đã giúp xác định phím riêng biệt nào trên bàn phím được nhấn. Bạn có thể sử dụng macro tương tự để kiểm tra trạng thái của nút bấm chuột.

```
#define BUTTONDOWN(name, key){name.rgbButtons[key] & 0x80}
```

Macro này trả về *TRUE* hoặc *FALSE* cho mỗi nút bấm trên chuột.

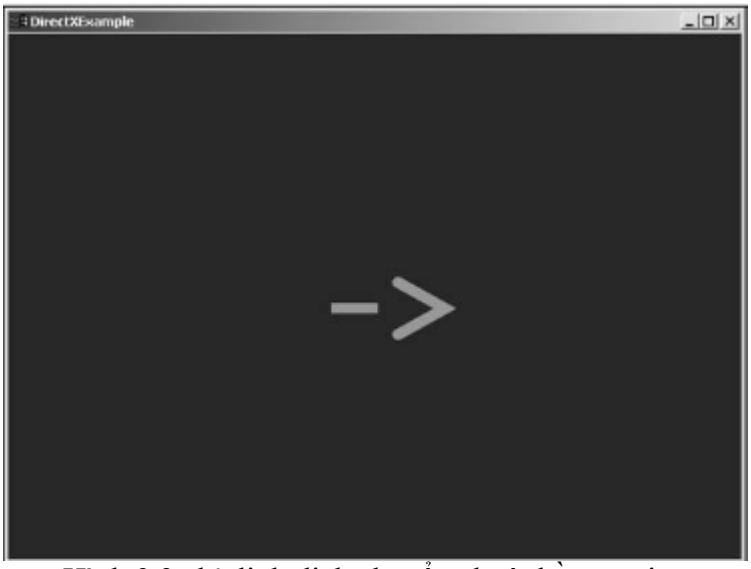
Chú ý:

Giá trị X, Y và Z trong cấu trúc *DIMOUSESTATE* không lưu vị trí hiện tại cảu chuột; đúng hơn là chúng lưu vị trí tương đối của chuột so với vị trí trước. Ví dụ, nếu bạn chuyển chuột xuống 10 đơn vị, giá trị Y sẽ bằng 10. Khi bạn dọc từ chuột, bạn phải giữ lại các giá trị đọc từ chuột ở trạng thái trước. Như vậy bạn có thể giải thích chính xác dịch chuyển của chuột.

Đoạn chương trình sau minh họa giá trị cần để đọc thiết bị chuột. Điều khiển giá trị này kiểm tra cả sự dịch chuyển của chuột và trạng thái của mỗi nút bấm trên chuột.

```
//xác định macro cần để kiểm tra trạng thái của các phím trên bàn phím.
#define KEYDOWN (name, key) { name[key]& 0x80}

//cần lưu trạng thái của chuột.

//biến này lưu giữ trạng thái hiện tại của thiết bị chuột.
DIMOUSESTATE mouseState;

//biến này lưu giữ vị trí hiện tại X của sprite
LONG currentXpos;
//biến này lưu giữ vị trí hiện Y tại của sprite
LONG currentYpos;
//biến này lưu giữ vị trí hiện tại Z của sprite
LONG currentZpos;

//thiết lập vị trí theo mặc định cho sprite
curretnXpos=320;
curretnYpos=240;

//đây là vòng lặp chính của game, đọc từ thiết bị chuột mỗi trạng thái.
While(1)
{
//kiểm tra chuột và lấy trạng thái hiện tại của nút được nhấn.
g_lpDIDevice->GetDeviceState(sizeof (mouseState), (LPVOID) &mouseState);

//làm gì đó với Input

//BUTTONDOWN macro này kiểm tra nếu nút bấm chuột thứ nhất được nhấn
if (BUTTONDOWN (mouseState, 0))
{
//làm gì đó với nút bấm chuột này
}
//kiểm tra sự dịch chuyển của chuột
//xem xét hướng đi của chuột theo trục X được dịch chuyển bao xa
currentXpos +=mousesState.lX;
//xem xét hướng đi của chuột theo trục Y được dịch chuyển bao xa
currentYpos +=mousesState.lY;

//làm gì đó vói dịch chuyên chuột
}
```

Bạn có thể tìm thấy mã nguồn của ví dụ trong thư mục chapter9\example2 trên đĩa CD-ROM. Ví dụ này minh họa dịch chuyển của chuột sử dụng 2D sprite, nhấn trái và nhấn phải chuột được biểu diễn băng là mũi tên định hướng trên màn hình. Hình 9.3 chỉ ra ví dụ này.



Hình 9.3 chỉ định dịch chuyển chuột bằng sprite.

Chú ý:

Cấu trúc *DIMOUSESTATE* cung cấp biến để lưu giữ trạng thái của chuột với 8 nút hỗ trợ.

Sử dụng gamepad hoặc Joystick

Gamepad và các cần điều khiển đã trở thành thông dụng hiện nay. Ngoài ra hầu hết các joystick controller sử dụng để gắn vào trong game port trên card âm thanh, hầu hết các thiết bị bán trên thị trường hiện nay sử dụng kết nối USB. Kết nối USB cho thiết bị một ưu thế hơn các thiết bị khác. Thiết bị USB dễ dàng tìm thấy bởi hệ thống và điều khiển thông qua giao diện thông dụng HID. Vì thế, đọc từ gamepad và joystick đã trở nên dễ dàng hơn.
Sự khác nhau chính giữa sử dụng joystick và gamepad là sự cần thiết liệt kê tuyệt đối các Input Device. Vì nhiều joystick có thể gắn vào hệ thống, nên DirectInput không có *GUID* xác định trước cho những thiết bị này. Trước khi bạn có thể gọi *CreateDevice* để chuẩn bị sử dụng một joystick, bạn phải liệt kê các Input Device mà đã cài đặt trên hệ thống.

Liệt kê Joystick

Liệt kê các Device làm cho DirectInput yêu cầu mỗi thiết bị lại một lần nữa tìm kiếm các chuẩn mà nó thiết lập. Ví dụ, nếu bạn gọi *EnumDevices* như sau:

```
hr= g_lpDI->EnumDevices(DI8DEVCLASS_GAMECTRL,
EnumDevicesCallback,
NULL,
DIEDFL_ATTACHEONLY);
```

Sau đó những thiết bị trả về cho hàm *EnumDevicesCallback* sẽ chỉ có thể là dạng *DI8DEVICECLASS_GAMECTRL*, đây là đích thực những gì mà bạn cần khi tìm kiếm cho joystick.

Kiểm soát một Joystick

Bàn phím và chuột gây ra ngắt phần cứng báo hiệu cho hệ thống rằng có dữ liệu Input mới đang hiện hành. Điều mà hầu hết các joystick cần là thỉnh thoảng chúng được *kiểm soát*. Thời hạn kiểm soát có liên quan tới việc kiểm tra Device để phát hiện Input mới. Sau khi một thiết bị đã được kiểm soát, bạn có thể giới hạn Input hợp lệ mới từ chúng.

Chú ý:
Joystick và gamepad sử dụng cấu trúc định trước `DIDATAFORMAT` và `DIJOYSTATE2`

Joystick là những thiết bị số không hoàn chỉnh, chúng cũng bao gồm một bộ phận analog. Thông thường, joystick sử dụng Digital Input cho các nút bấm, có nghĩa là chúng là một trong hai kiểu: lên hoặc xuống, và chúng sử dụng Analog Input cho chính cần diều khiển của mình. Kiểu Analog Input cho phép bạn nhận biết khoảng cách mà joystick đã dịch chuyển.
Một dịch chuyển nhỏ của cần điều khiển hướng về phía bên phải cũng sẽ gửi đi một giá trị nhỏ tới điều khiển chương trình, ngược lại nếu đẩy cần điều khiển hoàn toàn sang phải sẽ gửi đi một giá trị khá lớn. Độ lớn của giá trị này được xác định bởi đặc tính phạm vi của thiết bị.
Đặc tính phạm vi thường thiết lập cho phần Analog của cần điều khiển và nó bao gồm các giá trị lớn nhất và giá trị nhỏ nhất mà thiết bị sẽ tạo. Ví dụ, thiết lập hạn nhỏ nhất của phạm vi tới -1000 và lớn nhất tới 1000 thì nó chỉ cho phép game của bạn có những giá trị mà rơi vào trong khoảng giới hạn này. Dịch chuyển cần điều khiển bằng mọi cách sang trái sẽ đưa giá trị về -1000, ngược lại nếu dịch chuyển nó sang phải sẽ làm tăng giá trị về phía 1000. Bạn có thể thiết lập giới hạn của thiết bị tới bất kỳ giá trị nào mà làm nên cảm giác thật cho game của bạn.

Thiết lập phạm vi của một cần điều khiển.

Để thiết lập phạm vi đặc tính cho phần Analog của cần điều khiển, bạn phải sử dụng hàm `EnumObjects`. Như bạn đã biết từ trước, hàm `EnumObjects` làm việc tương tự như `EnumDevices` nhưng nó gửi cho hàm callback của nó các detail trên các bộ phận khác nhau của Device. Một ví dụ hàm callback được chỉ ra dưới đây:

```
/******************************************************************
EnumObjCallback
******************************************************************/
BOOL CALLBACK EnumObjCallback(const DIDEVICEOBJECTINSTANCE *pdidoi, VOID*
pContext)
{
//nếu đối tượng này là một axis type object, ta thử thiết lập phạm vi
if (pdidoi->dwType & DIDFT_AXIS)
{
            //tạo một cấu trúc DIPROPRANGE
DIPROPRANGE diprg;

//mỗi cấu trúc cần một cấu trúc kiểu DIPROPHEADER được gán
diprg.diph.dwSize = sizeof(DIPROPRANGE);
diprg.diph.dwHeaderSize = sizeof(DIPROPHEADER);
diprg.diph.dwHow=DIPH_BYID;
diprg.diph.dwObj=pdidoi->dwType;    //chỉ định trục liệt kê

//giá trị lớn nhất và nhỏ nhất của phạm vi đang thiết lập ở đây
diprg.lMin=-100;
diprg.lMax=100;
```

```
HRESULT hr;

//thiết lập phạm vi cho trục
hr=g_JoystickDevice->SetProperty(DIPROP_RANGE, &diprg.diph);

//kiểm tra để biết được nếu thiết lập phạm vi đặc tính thành công
if FAILED(hr)
                  return DIENUM_STOP;
}
//Thông báo cho EnumObjects tiếp tục tới Object tiếp theo tsrong Device này
return DIENUM_CONTINUE;
}
```

Ở ví dụ này, trước tiên là kiểm tra để biết nếu đối tượng được truyền cho callback có *kiểu trục (axis)*. Một *axis object* là một kiểu biểu diễn phần điều khiển analog của joystick controller. Nếu một *axis Deivce* hợp lệ được sử dụng, chương trình sẽ thử thiết lập giá trị phạm vi cho chúng.
Đầu tiên một cấu trúc ***DIPROPRANGE*** được tạo sẽ giữ thông tin liên quan đến phạm vị. Cấu trúc ***DIPROPRANGE*** được định nghĩa như sau:

```
Typedef struct DIPROPRANGE {
DIPROPRANGE diph;
LONG lMin;
LONG lMax;
} DIPROPRANGE, * DIPROPRANGE;
```

Biến thứ hai và thứ ba trong cấu trúc này: lMin và lMax trên thực tế biểu diễn giá trị giới hạn lớn nhất và nhỏ nhất. Bạn có thể thiết lập hai giá trị này tới bất cứ nơi đâu mà game của bạn cần, và biến lMin luôn nhỏ hơn giá trị lMax như đã biết.
Biến đầu tiên trong cấu trúc ***DIPROPRANGE*** là một cấu trúc khác: ***DIPROPHEADER***. Cấu trúc ***DIPROPHEADER*** cần thiết cho tất cả các cấu trúc đặc tính.

```
Typedef struct DIPROPHEADER{
DWORD dwSize;
DWORD dwHeaderSize;
DWORD dwObj;
DWORD dwHow;
} DIPROPHEADER, *DIPROPHEADER;
```

Cấu trúc ***DIPROPHEADER*** cần chỉ 4 biến được thiết lập. Biến thứ nhất ***dwSize*** biểu diễn kích thước của cấu trúc gửi kèm tính theo byte. Trong trường hợp này, nó là cấu trúc ***DIPROPRANGE***.
Biến thứ hai ***dwHeaderSize*** là kích thước của cấu trúc ***DIPROPHEADER***.
Biến thứ ba và thứ tư làm việc cùng nhau. Nội dung của biến ***dwHow*** biểu diễn kiểu của dữ liệu trong biến ***dwObj***. ***dwHow*** có thể là một trong những giá trị sau:

- DIPH_DEVICE – ***dwObj*** phải thiết lập về 0.
- DIPH_BYOFSET – ***dwObj*** là phần bù trong định dạng dữ liệu hiện tại
- DIPH_BYUSAGE – ***dwObj*** phải thiết lập về cách sử dụng trang HID và sử dụng các giá trị.
- DIPH_BYID – ***dwObj*** được thiết lập định dạng đối tượng. Bạn có thể tìm thấy nó trong cấu trúc ***DIDEVICEOBJECTINSTANCE*** mà được truyền cho hàm callback.

Cuối cùng, sau khi những cấu trúc đã được bổ xung đầy đủ. Hàm này sẽ áp dụng ***GUID*** của đặc tính để thiết lập tham số đầu tiên của nó và một địa chỉ tới cấu trúc chứa thông tin đặc tính mới.

Chú ý:

Vài thiết bị không cho phép phạm vi thay đổi. Đặc tính phạm vi chỉ được đọc (read-only).

Bạn có thể thay đổi những đặc tính khác của một Device theo cùng một phương thức nhờ đặc tính phạm vi nào được thay đổi. Các đặc tính còn lại sẽ tồn tại cho các thiết lập khác. Ví dụ, *DIPROP_DEADZONE* là giá trị phạm vi chỉ ra phần nào của joystick sẽ dịch chuyển mà không có tác dụng. *DIPROP_FFGAIN* thiết lập tăng tốc cho force feedback, và *DIPROP_AUTOCENTER* thông báo cho thiết bị là nó nên quay về tâm của chính nó hay không khi người sử dụng thoát khỏi.

Đọc từ joystick

Joystick cũng giống như các Input Devices khác, cần sủ dụng hàm *GetDeviceState*. Trong trường hợp cần điều khiển và gamepad, bộ nhớ đệm phải lưu giữ nguồn dữ liệu vào là một trong hai kiểu *DIOYSTATE* hoặc là *DIOYSTATE2*. Sự khác nhau chính giữa hai cấu trúc này là số object trên 1 Joystick Device được đọc. Cấu trúc *DIOYSTATE* cho phép chỉ hai Analog Device, ngược lại cấu trúc *DIOYSTATE2* có thể điều khiển nhiều hơn.
Vì Input từ Joystick không phải là một phần tuyệt đối, bạn có thể giữ lại bất kỳ một dich chuyển nào trước đó. Ví dụ, nếu bạn đang sử dụng joystick để điều khiển dịch chuyển của một sprite xung quanh màn hình, bạn cần giữ lại trong một biến riêng biệt vị trí hiện tại X và Y. khi new input được đọc từ joystick, nguồn dữ liệu mới sẽ được điền vào vị trí hiện tại X và Y. Ví dụ sau minh họa điều này:

```
//có hai biến lưu vị trí hiện tại của sprite
LONG curX;
LONG curY;

//đây là vị trí của sprite thiết lập theo mặc định.
curX=320;
curY=240;
while (1)
{
sử dụng cấu trúc DIOYSTATE để lưu dữ liệu từ joystick
DIOYSTATE2 js;
//đầu tiên là kiểm sóat joystick
g_joystickDevice->Poll();

//lấy nguồn dữ liệu vào hiện tại từ thiết bị.
g_joystickDevice->GetDeviceState( sizeof(DIOYSTATE), &js);

//điền giá trị mới vào vị trí hiện tại của X,Y
curX+=js.lX;
curY+=js.lY;

//vẽ sprite với vị trí mới cập nhật
}
```

Đây là một phần mã nguồn nhỏ đầu tiên kiểm soát thiết bị cần điều khiển để đưa ra Input mới. Sau đó new Input được đưa vào cấu trúc *DIJOYSTATE2*. Cuối cùng *lX* và *lY* được điền vào vị trí hiện tại *X*,*Y* của sprite. Biến *lX* và *lY* biểu diễn Input trả về từ điều khiển analog đầu tiên.

Bạn có thể tìm thấy ví dụ đầy đủ về đọc từ joystick trong thư mục chapter9\example4 trên đĩa CD-ROM.

Hỗ trợ nhiều thiết bị Input Devices.

Hầu hết các game trên console cho phép nhiều người người chơi. PCs cũng vậy. Với khả năng gắn nhiều gamepad hoặc joystick bằng cổng USB, game trên PC chỉ bị giới hạn bởi bạn có thể có sáng kiến gì mà thôi. Trong phần này, tôi sẽ giải thích quá trình cần thiết để hỗ trợ nhiều thiết bị.
Như từ trước bạn đã gọi, mỗi Input Device cần DirectInput Device riêng cho mình. Vì điều này, chương trình của bạn cần có khả năng lưu nhiều DirectInput Device. Tạo một trong hai kiểu: ma trận hoặc vecto của đối tượng *IdirectInputDevice8* cho phép bạn làm điều này.
Bước tiếp theo là liệt kê các Device đã cài đặt. Ví dụ nếu game của bạn cần hỗ trợ 4 gamepad, bạn phải gọi *EnumDevices* và thu thập thông tin trả về thông qua hàm callback cho mỗi Device của gamepad. Sau đó bạn sẽ lưu được dữ liệu cho mỗi Device mà callback của bạn đã lưu. Sau khi đã tạo tất cả các Device, bạn sẽ có truy cập để làm tất cả những gì mà bạn muốn.

Đoạn chương trình sau chỉ ra ví dụ cơ bản của quá trình này.

```
#define NUM_DEVICES 4
//4 thiết bị DirectInput
LPDIRECTiNPUTdEVICE8 devices[NUM_DEVICES];

//the DirectInput Object
LPDIRECTINPUT8    g_lpDI = NULL;
Int curCount =0;  //lưu số thiết bị hiện tại mà bạn có

Int APIENTRY WinMain(HINSTANCE hInts, HINSTANCE, LPSTR, int)
{
//biến lưu giá trị trả về
HRESULT hr;

//tạo DirectInput Object
hr= DirectInput8Create (hInstance,
DIRECTINPUT_VERSION,
                              IID_IdirectInput8,
(void**) &g_lpDI,
NULL);
//gọi hàm EnumDevice
hr= g_lpDI->EnumDevices(DI8DEVCLASS_GAMECTRL,
EnumDevicesCallback,
NULL,
DiEDFL_ATTACHEONLY);

//kiểm tra giá trị trả về của   hàm EnumDevices
if FAILED (hr)
return false;

//làm gì đó với thiết bị ở đây
}

/*************************************************************
*EnumDevicesCallback
*************************************************************/

BOOL CALLBACK EnumDevicesCallback( const DIDEVICEINSTANCE, VOID* pContext)
```

```
{
//biến giữ giá trị trả về
HRESULT hr;

//gọi   CreateDevice cho thiết bị trả về này
hr= g_lpDI->CreateDevice(pdidInstance->guidInstance, &devices[curCount],
NULL);

//nếu việc gọi hàm CreateDevice lỗi, ngưng quá trình liệt kê các thiết bị.
if( FAILED (hr))
      {
            return DIENUM_CONTINUE;
}
else {
curCount ++;
if(curCount >= NUM_DEVICES)
            return DIENUM_STOP;
}
//tiếp tục liệt kê
return DIENUM_CONTINUE;
}
```

Hàm callback này không làm gì nhiều. Nó thử gọi *`CreateDevice`* trên mỗi Input Device mà được truyền cho nó. Nếu một thiết bị có thể đọc được, nó tăng giá trị biến đếm và giữ lại chờ tiếp. Chương trình hiện tại hỗ trợ 4 thiết bị. Nếu hơn 4 thiết bị gamepad cần dùng đến, thì kích thước của ma trận lưu DirectInput Device phải thay đổi. Nếu bạn không muốn biết bao nhiêu thiết bị bạn phải có hoặc bạn muốn giữ lại mọi thứ ở dạng động (dynamic) hãy sử dụng vecto đối tượng *`IDIRECTINPUTDEVICE8`*.

Dành lại một Input Device

Thỉnh thoảng trong khi học làm game, Input device bị mất. Nếu game của bạn đã thiết lập cooperative level cho mỗi thiết bị thành truy cập không độc quyền, thì ứng dụng khác phải bắt đầu hạn chế truy cập của bạn tới thiết bị này. Trong trường hợp như vậy, bạn cần dành lại thiết bị trước khi bạn có thể tiếp tục đọc từ nó và sử dụng Input của nó.
Khi truy cập tới một thiết bị đã bị mất, giá trị trả về từ hàm *`GetDeviceState`* sẽ bằng *`DIERR_INPUTLOST`*. Khi việc này xảy ra, bạn cần gọi hàm *`Acquire`* trong vòng lặp cho đến khi truy cập tới thiết bị được hoàn lại.
Đoạn chương trình sau minh họa cách dành lại thiết bị một truy cập đã bị mất như thế nào:

```
HRESULT hr;       //biến giữ giá trị trả về
//đây là vòng lặp chính, đọc dữ liệu từInput Device mỗi trạng thái.

While(1)
{
//gọi hàm GetDeviceState và save lại giá trị trả về
hr= DI_Device->GetDeviceState (sizeof(DIMOUSESTATE), (LPVOID)&mouseState);

//kiểm tra trạng thái trả về để biết thiết bị có còn truy cập được không
if FAILED (hr)
{
//thử dành lại Input Device
hr= DI_Device->Acquire();
//tiếp tục vòng lặp cho đến khi thiết bị dành lại đươc
while (hr==DIER_INPUTLOST)
```

```
            hr=DI_Device->Acquire();
//chỉ trả về và không làm gì với trạng thái này
continue;
}
//kiểm tra Input và làm gì đó với nó
}
```

chú ý:

Hầu hết các game cần nhiều Input Device để nhiều người chơi cùng một lúc. Bằng cách tạo nhiều DirectInput Device, bạn có thể hỗ trợ nhiều thiết bị riêng biệt.

Làm sạch DirectInput

DirectInput giống như Direct3D điều nó cần là bạn xử lý các đối tượng bạn tạo trong khi hoàn thành ứng dụng của bạn. Trong việc tính toán đến đối tượng DirectInput, bạn cũng phải xóa bỏ mọi dành lại các thiết bị mà bạn đã lấy được điều khiển trước đó. Nếu bạn quên xóa bỏ các Input Device mà bạn đã sử dụng, khi game kết thúc, những thiết bị này vẫn còn bị khóa bởi hệ thống vì thế bạn không thể sử dụng chúng nứa. Mặc dù một joystick hoặc gamepad bị khóa không ảnh hưởng nhiều lắm, nhưng với chuột và bàn phím thì có thể làm cho bạn phải restart lại máy tính để lấy chúng trở lại.

Hàm *`Unacquire`* xử lý một thiết bị mà dành lại được trước đó qua DirectInput.

```
HRESULT Unacquire(VOID);
//Unacquire là một phương thức mà giao diện DirectInput Device cung cấp
//Đoạn ví dụ sau xóa bỏ thiết bị Input và xử lý cả DirectInput Device và
//DirectInput Object.

//kiểm tra bạn có một DirectInput Object hợp lệ hay không
if (DI_Object)
{
//kiểm tra để biết bạ có một DirectInput Device hợp lệ không
if (DI_Device)
{
//xóa bỏ Input Device
DI_Device-> Unacquire();
//xử lý DirectInput Device
DI_Device->Release();

//đưa biến DirectInput Device về NULL
DI_ Device =NULL;
}
//xử lý DirectInput Object
DI_Object->Release();
//đưa biến DirectInput Object về NULL
DI_ Object =NULL;
}
```

Ở đây bạn nên hiểu biết rõ ràng về gán và đọc từ Input Device chuẩn qua DirectInput. Trong phần tiếp theo, bạn sẽ học làm thế nào để sử dụng force feedback để nhúng player vào thế giới mà bạn tạo.

Force Feedback

Từ khi phóng thích sự khởi tạo của game đồ họa console trên thị trường, các gamer đã trở nên quen thuộc với khái niệm force feedback. Force feedback là khả năng gửi những mức chuyển động khác nhau tới một Input Device. Gamepad dành cho hệ thống console thường hỗ trợ force feedback, ngược lại, feedback trong Input Device của PC vẫn đang còn rất ít.

Hiệu ứng Force feedback

Force feedback Device thực hiện dao động của chúng dựa trên các hiệu ứng. Một hiệu ứng force feedback được làm từ một hoặc nhiều force thực hiện trên controller. Force trong DirectInput là nút nhấn hoặc resistance felt trên controller trong thời gian sử dụng một hiệu ứng. Các hiệu ứng có một vài kiểu khác nhau:

- Constat Force – một force liên tục đều đặn trong một điều khiển đơn.
- Ramp Force – một force làm tăng hoặc làm giảm cường độ một cách đều đặn trong mọi thời điểm.
- Pertodic Effect – a pulsating force
- Conditional – một hiệu ứng được gây ra như một phản ứng tới một chuyển động riêng biệt.

Mỗi force có một *độ lớn*, hay còn gọi là *cường độ* và một khoảng thời gian tồn tại hay còn gọi là chiều dài của thời gian. Khi thay đổi độ lớn cho phép bạn tăng, giảm độ rung hoặc reinstance the user feels in your game.

Chú ý:
Lạm dụng quá nhiều force feedback hoặc sử dụng nó ở những thời điểm không thích hợp trong game sẽ làm người chơi có cảm giác khó chịu. Nên phải sử dụng force feedback một cách hợp lý

Để sử dụng thiết bị force feedback như gamepad trong game của bạn, bạn cần làm những việc sau đây:

1. Tạo DirectInput Object
2. Liệt kê các thiết bị game controller đã cài đặt mà hỗ trợ force feedback.
3. Tạo một thiết bị trên cơ sở gamepad.
4. Tạo hiệu ứng force feedback bạn muốn sử dụng.
5. Bắt đầu hiệu ứng

Liệt kê Input Device cho force feedback

Vì force feedback không phải là phổ biến trong game controller, nên bạn cần tìm kiếm một cách cụ thể các điểm đặc trưng này khi liệt kê Input Device. Trước đây chỉ có Flag mà bạn gửi tới `EnumDevices` là `DIED_ATTACHEONLY`, chỉ ra rằng hàm này sẽ trả về các thiết bị đã cài đặt cho callback. Nếu ta có cờ hiệu này, thì callback sẽ tiếp nhận cả các thiết bị force feedback và thiết bị nonfeedback. Vì bạn biết từ khi bắt đầu là bạn chỉ muốn tìm kiếm các thiết bị force feedback, nên bạn nên điền cờ hiệu `DIEDFL_FORCEFEEDBACK` cho hàm `EnumDevices`. Việc này báo cho `EnumDevices` chỉ được báo cáo lại các thiết bị force feedback hiện hành.

Ví dụ chương trình sau chỉ ra gọi cập nhật tới `EnumDevices`.

```
//biến sử dụng để lưu Input Device
```

```
LPDIRECTINPUTDEVICE8 FFDevice = NULL;

//liệt kê các thiết bị đã cài đặt, tìm kiếm game controller hoặc joystick
//dùng hỗ trợ force feedback.
HRESULT hr;
Hr= g_pDI->EnumDevices (DI8DEVCLASS_GAMECTRL,
                        FFDeviceCallback,
                        NULL,
                        DIEDFL_ATTACHEONLY | DIEDFL_FORCEFEEDBACK));
```

Hàm *EnumDevices* trên đã cập nhật lại với cờ hiệu *DIEDFL_FORCEFEEDBACK*.

Đoạn ví dụ sau chỉ ra hàm callback cần để tìm thiết bị force feedback

```
/*************************************************************
*FFDeviceCallback
*************************************************************/
BOOL CALLBACK FFDeviceCallback( const     DIDEVICEINSTANCE* pInst, VOID*
pContext)
{
HRESULT hr;
//tạo thiết bị
hr-g_pDI->CreateDevice(pInst->guidInstance, &FFDevice, NULL);
//thiết bị này không tạo đuoc, thì giữ lại để tìm kiếm cái khác.
If (FAILED (hr))
            Return DIENUM_CONTINUE;
//chúng ta tìm thấy một thiết bị, ngưng lại liệt kê
return DIENUM_STOP;
}
```

Ở đây chỉ những thiết bị force feedback hợp lệ được báo cho hàm callback. Callback sẽ thử tạo Device dựa trên cái đầu tiên mà nó gặp. Nếu callback thành công khi tạo Device, nó sẽ dừng lại liệt kê, còn nếu không thì liệt kê tiếp tục tìm một Device thích hợp.

Tạo hiệu ứng force feedback

Sau khi bạn tìm thấy vàđã tạo một DirectInput Device cho controller bạn sẽ sử dụng nó. Bạn cần tạo một đối tượng hiệu ứng. Đối tượng hiệu ứng force feedback của DirectInput tạo ra dựa trên giao diện *IdirectInputEffect*. Mỗi đối tượng *IdirectInputEffect* trình bày chi tiết hiệu ứng cho hệ thống.

Hiệu ứng trước tiên được tạo bởi việc điền vào đầy đủ thông tin trong cấu trúc *DIEFFECT*. Cấu trúc này mô tả diện mạo bên ngoài khác nhau của hiệu ứng, như là thời gian tồn tại mà nó ảnh hưởng và độ mạnh của hiệu ứng.
Cấu trúc *DIEFFECT* sau đó được gán cho tham số của hàm *CreateEffect*. Hàm *CreateEffect* đăng ký hiệu ứng với DirectInput và tải hiệu ứng xuống cho Device. sau khi hiệu ứng đã được tải xuống cho force feedback Device, nó đã sẵn sàng để sử dụng.

Chú ý:
Để một hiệu ứng tải xuống force deedback Device, Device phải được thiết lập cooperative level kiểu truy cập độc quyền. Force feedback Device không thể chia sẻ chức năng feedback giữa các ứng dụng khác nhau.

Tôi sẽ đưa cho bạn một mô tả ngắn gọn cấu trúc *DIEFFECT* và sử dụng hàm *CreateEffect* vì thế bạn có thể nhìn thấy quá trình một cách cụ thể hơn.
Bạn phải khai báo cấu trúc *DIEFFECT* cho mỗi đối tượng hiệu ứng mà bạn muốn tạo. Cấu trúc *DIEFFECT* được định nghĩa dưới đây:

```
Typedef struct DIEFFECT{
DWORD dwSize;
DWORD dwFlags;
DWORD dwDuration;
DWORD dwSamplePeriod;
DWORD dwGain;
DWORD dwTriggerRepeatInterval;
DWORD cAxes;
LPLONG lgdwAxes;
LPDWORD rglDirection;
LPDIENVELOPE lpEnvelope;
DWORD cbTypeSpecificParams;
LPVOID lpvTypeSpeciParams;
DWORD dwStartDelay;
} DIEFFECT, *LPDIEFFECT;
```

cấu trúc *DIEFFECT* gồm những giá trị sau:

- dwSize – kích thước của cấu trúc *DIEFFFECT* tính theo byte
- dwFlags – cờ hiệu mô tả một vài biến được sử dụng như thế nào
  - DIEFF_CARTESIAN – giá trị trong biến *rglDirection* được xem như tọa độ đề các.
  - DIEFF_OBJECTIDS – giá trị trong biến *dwTriggerButton* và *rgdwAxes* là các đối tượng nhận biết
  - DIEFF_OBJECTOFFSETS – giá trị trong biến *dwTriggerButton* và *rgdwAxes* là các khoảng trống định dạng dữ liệu
  - DIEFF_POLAR – giá trị trong biến *rglDirection* được xem như tọa độ cực.
  - DIEFF_SPHERICAL – giá trị trong biến *rglDirection* được xem như tọa độ cầu.
- dwDuration – khoảng thời gian tồn tại của hiệu ứng tính theo microsecond. Nếu khoảng thời gian tồn tại này tiếp tục diễn ra, hãy sử dụng kiểu giá trị *INFINITE*
- dwSampePeriod – tốc độ thử nghiệm của hiệu ứng playback. 0 chỉ ra rằng tốc độ thử nghiệm mặc định được sử dụng.
- dwGain – tốc độ của hiệu ứng.
- dwTriggerButton – chỉ ra nút được dùng để gây ra hiệu ứng. Giá trị này phụ thuộc vào giá trị trong biến *dwFlags*.
- cAxes – số trục mà hiệu ứng sử dụng
- rgdwAxes – con trỏ tới ma trận kiểu *DWORD* mà chứa IDs hoặc khoảng trống tới trục mà hiệu ứng sử dụng
- rglDirection – ma trận tọa độ tương ứng với kiểu tọa độ chọn cho biến *dwFlags*.
- lpEnvelope – một con trỏ tùy ý trỏ đến cấu trúc *DIENVELOPE*. Cấu trúc này xác định vỏ bọc để áp dụng cho hiệu ứng này. Khi không có hiệu ứng nào cần kiểu này bạn có thể sử dụng giá trị *NULL*
- cbTypeSpecificParams – số byte của tham số bổ xung cho mỗi kiểu hiệu ứng.
- lpvTypeSpecificParams – biến này lưu tham số đã thảo luận trong biến trước. Biến này có thể lưu bất kỳ cấu trúc đã định nghĩa sau:
  - DIEFT_CUSTOMFORCE – một cấu trúc kiểu *DICUSTOMERFORCE* được truyền.
  - DIEFT_PERIODIC – một cấu trúc kiểu *DIPERIODIC* được sử dụng.

- DIEFT_CONSTANCTFORCE – một hằng số cấu trúc force, ***DICONSTANTFORCE*** được truyền.
- DIEFT_RAMPFORCE – một cấu trúc ramp force của ***DIAMPFORCE*** được sử dụng.
- DIEFT_CONDITION – một cấu trúc kiểu ***DICONDITION*** được truyền

▪ dwStartDelay – thời gian tính theo microsecond mà thiết bị đợi trước khi chạy một hiệu ứng.

Cấu trúc ***DIEFFECT*** hoàn thành được truyền cho hàm ***CreateEffect***. Hàm ***CreateEffect*** cần 4 tham số và được xác định giống như sau:

```
HRESULT CreateEffect(
REFGUID rguid,
LPCDEFFECT lpeff,
LPDIRECTINPUTEFFECT *ppdeff,
LPUNKNOWN punkOuter
);
```

Tham số thứ nhất có kiểu ***GUID*** thuộc dạng force được tạo. Ví dụ nếu bạn đang thử tạo một hiệu ứng force cố định, bạn hãy sử dụng ***GUID*** như ***GUID_ConstantForce***. Tham số thứ hai được truyền cho cấu trúc ***DIEFFECT***. Tham số thứ 3 là là địa chỉ của biến mà sẽ lưu hiệu ứng mới. Biến này phải thuộc dạng ***IdirectInputEffect***. Tham số cuối cùng của ***CreateEffect*** thường là ***NULL***.
Sau khi bạn có hiệu ứng đã tạo và sẵn sàng đi tiếp, bước tiếp theo sẽ là khởi chạy nó.

Bắt đầu một hiệu ứng.

Trước kia người sử dụng có thể cảm thấy hiệu ứng của bạn trong những hành động trong game. Playback của hiệu ứng force feedback được điều khiển thông qua hàm *Start*, nó là hàm nhớ của hàm *IdirectInputEffect*.

Hàm ***Start*** cần hai tham số.Tham số thứ nhất là số lần mà hiệu ứng được diễn ra. Tham số thứ hai là tập hợp các cờ hiệu mà gắn liền với cách diễn ra hiệu ứng như thế nào trong thiết bị.
Có hai kiểu cờ hiệu hợp lệ cho tham số thứ hai. Cả hai có thể được áp dụng. Nếu không cần có cờ hiệu nào bạn có thể thiết lập tham số này bằng 0.

▪ DIES_SOLO – hiệu ứng khác bất kỳ mà hiện tại bị dừng lại khi hiệu ứng này diễn ra.
▪ DIES_NODOWNLOAD – hiệu ứng không được tự động tải xuống thiết bị.

Chú ý:
Nếu hiệu ứng hiện tại đang diễn ra và hàm ***Start*** được gọi, thì hiệu ứng này được bắt đầu over from beginning.

Ví dụ này gọi hàm start thông báo cho DirectInput thực hiện hiệu ứng này một lần và chỉ ra rằng không có cờ hiệu nào được áp dụng.

```
g_pEffect->Start (1,0);
```

Sau khi gọi hàm *Start*, hiệu ứng bắt đầu diễn ra trên Device.Nếu hiệu ứng có một khoảng thời gian tồn tại, thì hiệu ứng kết thúc khi khoảng thời gian hoàn thành. Nếu khoảng thời gian tồn tại của hiệu ứng là vô hạn, bạn phải dừng ngay hiệu ứng.

Ngừng hiệu ứng.

Như tôi đã nhắc trước, nếu một hiệu ứng có một khoảng thờ gian tồn tại, nó kết thúc khi khoảng thờ gian được hoàn thành. Nhưng vì một điều gì đó, khoảng thời gian tồn tại của hiệu ứng thành vô hạn, hoặc người sử dụng nhấn phải nút Pause. Cả hai trường hợp trên cần phải kết thúc hiệu ứng ngay.

Việc này được hoàn thành thông qua hàm *Stop*. Hàm này không cần tham số nào và trả về, nó chỉ ra hàm thực hiện thành công hay không. Hàm Stop được khai báo sau đây:

```
HRESULT Stop(VOID);
```

Giá trị trả về là *DI_OK* có nghĩa là việc gọi tới hàm *Stop* đã thành công.

Tổng kết chương.

Input cũng giống như một phần không thể thiếu của bất kỳ game nào nên bạn nên dành sự chú ý đặc biệt đến nó trong toàn bộ vòng tròn khai triển. Khi game được xét duyệt, việc thi hành của Input có thể thực hiện hoặc thoát khỏi nó. Dành sự chú ý thích hợp cho hệ thống Input trong thời gian khai triển làm nâng cao kinh nghiệm của gamer.

Những vấn đề đã học.

Trong chương này, bạn đã học cách làm thế nào để sử dụng Input Devices. Bạn nên hiểu rõ những trọng điểm sau trogn chương này:

- Sử dụng thiết bị bàn phím và chuột như thế nào.
- Sự khác nhau giữa điều khiển analog và digital
- Hỗ trợ nhiều thiết bị như thế nào
- Tạo vào sử dụng hiệu ứng force feedback qua game controller như thế nào
- Cách thích hợp để giải phóng và ngừng hoạt động của DirectInput

Trong chương tiếp theo, bạn sẽ được giới thiệu DirectSound và cách sử dụng âm thanh và nhạc để làm nổi bật game của bạn.

Câu hỏi ôn tập

Bạn có thể tìm thấy câu trả lời cho những câu hỏi ôn tập và bài tập tự làm trong Appendix A

1. DirectInput cho phép những dạng Input Device nào?
2. Hàm nào tạo giao diện IdirectInput?

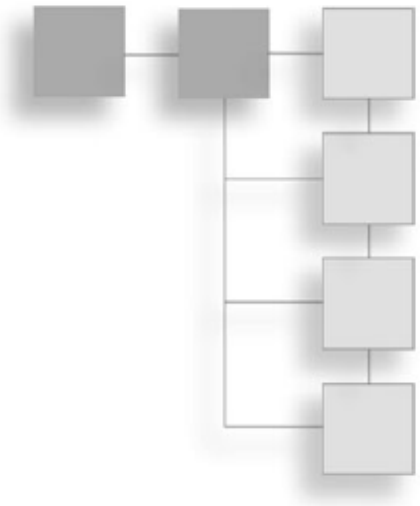
3. Sự thăm dò các Input Device trong hệ thống được gọi là gì?
4. Đọc từ bàn phím cần dạng bộ nhớ đệm nào?
5. Kiểu định dạng dữ liệu cho thiết bị Chuột là gì?

Bài tập tự làm

1. Hãy thay đổi ví dụ thiết bị chuột để dịch chuyển con trỏ của Windows
2. Thay đổi ví dụ gamepad để dọc từ nút điều khiển.

CHAPTER 10

DIRECTSOUND



Directsound

Directsound giúp game của bạn đến gần với cuộc sống. Khi bạn dùng những ưu điểm của nhạc nền và hiệu ứng âm thanh, thế giới game bạn tạo ra sẽ có một chiều sâu mới. Chương này sẽ giúp bạn học cách dùng âm thanh hiệu quả trong game.

Trong chương này:

- Directsound là gì?
- Sử dụng Directsound thế nào?
- Bộ đệm âm thanh là gì?
- Chạy một file âm thanh thế nào?
- Chạy lặp một đoạn âm thanh thế nào?
- Cài đặt và chỉnh âm lượng?

Âm thanh

Âm thanh rất quan trọng trong game. Nó dùng để cài đặt nhạc hiệu, building tension hoặc chào mừng vào cuối level. Âm thanh giúp bạn tạo ra một môi trường, từ tiếng xe đua chạy vòng quanh trường đua tới tiếng súng đạn rít qua đầu bạn. DirectX cung cấp cho bạn Directsound, giúp bạn dễ dàng thêm một âm thanh vào game.

DirectSound

Directsound cung cấp một giao tiếp lập trình ứng dụng(API) để phát lại âm thanh và âm nhạc. Trước đây, các nhà phát triển phải viết trình hỗ trợ cho các loại cạc âm thanh(soundcard) khác nhau vì họ có nhiệm vụ viết phần mềm cho từng loại. Với sự ra đời của DirectX và lớp trừu tượng hoá phần cứng của nó(hardware abstraction layer - HAL), nhà phát triển chỉ phải viết một tập hợp những hàm chung, hỗ trợ một lượng lớn cạc âm thanh.

DirectSound(DS) làm việc như thế nào?

Quản lý dữ liệu âm thanh thông qua dùng bộ đệm (buffers). Bộ đệm là một diện tích của bộ nhớ chứa dữ liệu âm thanh. Khi bạn cùng DS, bạn có thể có nhiều bộ đệm lưu giữ bất cứ dữ liệu âm thanh nào bạn muốn load. Sau đó bạn có thể điều khiển và chơi nó trong những bộ đệm đó. DS trộn chúng với nhau, và cho vào một bộ đệm đơn lẻ. Bộ đệm này chứa âm thanh cuối cùng mà người dùng nghe thấy.
Bộ đệm âm thanh có thể nằm ở bộ nhớ của cạc âm thanh hoặc bộ nhớ hệ thống.
Chú ý:
Bộ đệm trên bộ nhớ cạc âm truy cập nhanh hơn trên bộ nhớ hệ thống. Chúng ta nên chọn cách thứ 2 (dùng system memory) để làm bộ đệm âm thanh vì chúng sẽ không làm tốn bộ nhớ của cạc âm.
Như vậy, bộ đệm âm thanh là nơi chứa dữ liệu âm thanh. Ví dụ khi bạn load một file Wav để chạy, dữ liệu âm thanh trong file đó sẽ được đặt vào một bộ đệm âm thanh. Sau đó bạn có thể thay đổi, điều khiển, chạy dữ liệu bên trong bộ đệm đó.
Dưới đây là những kiểu bộ đệm âm mà DS dùng:
- Bộ đệm sơ cấp(primary buffer). Tất cả âm thanh được trộn trong bộ đệm sơ cấp. Cạc âm dùng âm thanh đã được hoà trộn trong đó để tạo âm thanh mà bạn nghe được.
- Bộ đệm thứ cấp(secondary buffer). Là những bộ đệm chứa tất cả dữ liệu âm mà game của bạn cần. DS giúp bạn chạy những âm thanh phức tạp bằng cách truy cập nhiều hơn một bộ đệm thứ cấp một cách đồng thời.
- Bộ đệm tĩnh(static buffer). Khi dữ liệu âm có kích thước giới hạn thì bạn có thể tạo một một bộ đệm tĩnh (kích thước cố định). Bộ đệm này cho phép load hoàn toàn một âm thanh riêng biệt vào bộ nhớ.
- Bộ đệm dòng (buffer). Có lúc âm thanh bạn muốn chơi quá lớn để cho vào bộ nhớ một lần. Trong trường hợp này, bạn cần một bộ đệm dòng. Bộ đệm dòng chỉ cho phép một phần của âm thanh được load vào bộ nhớ trước khi được phát. Sau khi âm thanh trong bộ đệm được phát, dữ liệu âm mới được load vào bộ nhớ đó.

Dùng DirectSound

Trước khi bạn dùng DS, bạn cần biết những bước liên quan. Như những thành phần DX khác, DS cần được khởi tạo trước khi bạn sử dụng nó. Bước đầu tiên để dùng DS là tạo thiết bị DS. Thiết bị này được miêu tả bởi giao tiếp IDirectSound8, cái cung cấpcác phương thức để tạo các bộ đệm âm thanh, thu nhận khả năng của phần cứng xử lý âm thanh, và thiết lập mức độ hợp tác của cạc âm thanh.

Thiết bị Directsound

Thiết bị DS miêu tả một giao tiếp tới một bộ phận của phần cứng về âm thanh trong máy tính của bạn. Để DS hoạt động, bạn phải lựa chọn loại cạc âm thanh và tạo thiết bị DS để miêu tả nó. Bởi thường một máy chỉ có một cạc âm nên DS cho phép bạn tạo thiết bị DS dựa trên cạc âm thanh mặc

định. Nếu máy của bạn có nhiều hơn một cạc, bạn phải liệt kê chúng và tìm ra cái mà chương trình của bạn cần.
Bạn tạo thiết bị DS bằng cách sử dụng hàm DirectSoundCreate8, định nghĩa như sau:

```
HRESULT WINAPI DirectSoundCreate8(
LPCGUID lpcGuidDevice,
LPDIRECTSOUND8 * ppDS8,
LPUNKNOWN pUnkOuter
);
```

Hàm này cần ba tham số:

- lpcGuidDevice. Nó miêu tả thiết bị âm thanh sẽ sử dụng. Tham số này có thể là DSDEVID_DefaultPlayback hoặc NULL. Dùng NULL khi bạn muốn dùng thiết bị âm thanh mặc định.
- ppDS8. Địa chỉ của biến sẽ lưu thiết bị DS mới được tạo ra.
- pUnkOuter. Giao tiếp IUnknown của đối tượng điều khiển. Giá trị này nên để NULL.

Một lệnh gọi chuẩn gọi hàm DirectSoundCreate8 sử dụng thiết bị âm thanh mặc định sẽ có dạng sau:

```
// biến giữ giá trị trả lại
HRESULT hr;
// biến lưu giữ thiết bị DS
LPDIRECTSOUND8 m_pDS;
// Thử tạo thiết bị DS
hr = DirectSoundCreate8( NULL, &m_pDS, NULL ) ;
// Kiểm tra giá trị trả lại để chắc chắn có một thiết bị đúng được tạo ra
if FAILED ( hr )
return false;
```

Nếu đoạn code trên ko thể tạo một thiết bị DS đúng, hàm này sẽ trả lại FALSE.

Liệt kê các thiết bị âm thanh

Thỉnh thoảng, bạn muốn liệt kê thiết bị âm thanh trong hệ thống. Nếu, ví dụ như, thiết bị âm thanh mặc định ko có đủ tất cả các hàm mà game cần thì bạn có thể tìm một thiết bị khác trong hệ thống của bạn.
Nếu bạn ko muốn dùng thiết bị mặc định, bạn phải liệt kê các thiết bị có sẵn trước khi gọi hàm DirectSoundCreate8. Khi qúa trình liệt kê hoàn thành, bạn sẽ cần GUID cho thiết bị, cái mà bạn sẽ truyền cho hàm DirectSoundCreate8 thay vì NULL.
Quá trình liệt kê được quản lý qua hàm DirectSoundEnumerate Như các thành phần trong DX, liệt kê các thiết bị cần một hàm callback. Hàm DirectSoundEnumerate gọi hàm callback mỗi khi một thiết bị âm thanh mới được phát hiện. Trong hàm callback, bạn có thể xác định khả năng của thiết bị và chọn xem có muốn dùng nó ko.
Hàm DirectSoungEnumerate được định nghĩa như sau:

```
HRESULT WINAPI DirectSoundEnumerate(
LPDSENUMCALLBACK lpDSEnumCallback,
LPVOID lpContext
);
```

Hàm DirectSoundEnumerate chỉ cần 2 tham số:

- lDSEnumCallback. Địa chỉ của hàm callback.
- lContext. Bất cứ dữ liệu nào bạn muốn truyền cho hàm callback.

Đoạn code sau cho một ví dụ gọi hàm DirectSoundEnumerate:

```
// Biến lưu đoạn code trả về
HRESULT hr;
// Gọi hàm DirectSoundEnumerate
hr = DirectSoundEnumerate(
(LPDSENUMCALLBACK)DSoundEnumCallback, 0);
// Kiểm tra đoạn code trả về để chắc chắc hàm được gọi thành công
if FAILED ( hr)
return false;
```

Hàm DirectSoundEnumerate callback

Hàm callback cung cấp cho DirectSoundEnumerate được gọi mỗi khi quá trình liệt kê tìm thấy một thiết bị mới. Nếu nhiều thiết bị được cài đặt trên hệ thống, hàm callback được gọi một lần cho từng cái.

Nhiệm vụ chính của hàm callback là cho code một cơ hội để tạo thiết bị DS và dùng nó để thu thập thông tin về thiết bị. Nếu bãn đang tìm kiếm một thiết bị âm thanh có khả năng bắt âm thanh chẳng hạn, bạn có thể kiểm tra khả năng của từng thiết bị truyền cho hàm callback để xem có khả năng này ko?

Hàm DirectSoundEnumerate cần hàm callback trong định dạng DSEnumCallback.

```
BOOL CALLBACK DSEnumCallback(
LPGUID lpGuid,
LPCSTR lpcstrDescription,
LPCSTR lpcstrModule,
LPVOID lpContext
);
```

Bạn phải khai báo hàm callback theo cách say. Hàm callback cần 4 tham số:

- lpGuid. Địa chỉ của GUID định nghĩa thiết bị âm thanh hiện tại. Nếu giá trị này là NULL, thì thiết bị đang được liệt kê là thiết bị đầu tiên.
- lpcstrDescription. Một biến NULL-terminated string cung cấp một dòng text miêu tả thiết bị hiện tại.

- lpcstrModule. Một biến NULL-terminated string cung cấp tên module của driver của DS cho thiết bị này.
- lpContext. Dữ liệu thêm vào được truyền cho hàm callback thông qua biến lpContext trong DirectSoundEnumerate.

Hàm DSEnumarate trả lại một giá trị boolean. Nếu giá trị là TRUE, hàm DirectSoundEnumerate tiếp tục liệt kê các thiết bị thêm vào. Nếu giá trị trả lại là FALSE, quá trình liệt kê kết thúc.

Chú ý:

Thiết bị đầu tiên thường được liệt kê hai lần, một với giá trị NULL được truền cho tham số lpGuid và lần hai là GUID của nó.

Ví dụ hàm callback say tạo một hộp thoại hiển thị tên của thiết bị âm thanh hiện tại và driver của nó.

```
/****************************************************************************
* DirectSoundEnumerate callback function
****************************************************************************/
BOOL CALLBACK DSCallback( GUID* pGUID,
LPSTR strDesc,
LPSTR strDrvName,
VOID* pContext )
{
// Biến tạm thời lưu thông tin về thiết bị
string tempString;
// Xây dựng biến String bằng thông tin cung cấp bởi hàm callback
tempString = “Device name = “;
tempString += strDesc;
tempString += “\nDriver name = “;
tempString += strDrvName;
// Hiển thị kết quả trên hộp thoại
MessageBox (NULL, tempString.c_str(), “message”, MB_OK );
// Tiếp tục liệt kê các thiết bị khác, trả lại TRUE
return true;
}
```

Một biến string tạm thời được tạo ra để lưu thông tin. Hàm trả về một giá trị TRUE, nên nó sẽ liệt kê tất cả các thiết bị âm thanh trong hệ thống. Toàn bộ ví dụ này nằm trong thư mục chapter10\example1 trên đĩa CD. Hình 10.1 chỉ ra kết quả:

Figure 10.1 Message box showing the sound device's name and driver.

Thiết lập mức độ hợp tác

Bởi DS cho bạn quyền truy cập tới thiết bị phần cứng, nên có cần xác định mức độ hợp tác. Tương tự như DirectInput, DS thử nhận quyền truy cập bình thường tới thiết bị. Với DirectInput, bạn có thể nhận quyền truy cập dành riêng tới một thiết bị, giới hạn quyền dùng nó trong ứng dụng của bạn. Với DS, bạn ko thể nhận quyền truy cập dành riêng với thiệt bị âm thanh, nhưng bạn có thể thiết lập mức độ ưu tiên cao nhất cho ứng dụng của bạn khi hệ điều hành dùng thiết bị âm thanh đó. Tất nhiên, vì bạn ko thể giành quyền giành riêng với cạc âm thanh, nên các ứng dụng khác(cả hệ điều hành) có thể gây ra các âm thanh của chúng.
Có ba mức độ hợp tác mà DS thiết lập được:
- DSSCL_NORMAL. Mức độ này hoạt động tốt với các ứng dụng khác bởi vẫn cho phép các sự kiện khác xảy ra. Bởi ứng dụng của bạn phải chia sẻ thiết bị, nên bạn ko thể thay đổi đinh dạng của primary buffer(bộ đệm sơ cấp).
- DSSCL_PRIORITY. Nếu bạn muốn nhiều sự điều khiển hơn với bộ đệm và âm thanh, bạn nên dùng mức độ này. Hầu hết games dùng nó.
- DSSCL_WRITEPRIMARY. Mức độ này giúp ứng dụng có thể viết lên primary buffer.

Mức độ hợp tác được thiết đặt bằng hàm SetCooperativeLevel. Giao tiếp IDirectSound8 hỗ trợ hàm này. Hàm này được định nghĩa như sau:

```
HRESULT SetCooperativeLevel(
HWND hwnd,
DWORD dwLevel
);
```

Hàm này cần hai tham số:
- hwnd. Kênh điều khiển(handle) của cửa sổ ứng dụng cái yêu cầu thay đổi mức độ hợp tác(cooperative level).
- Dwlevel. Một trong ba mức độ hợp tác đã nêu bên trên.

Ví dụ:

```
// biến lưu code trả về
HRESULT hr;
// biến chứa một thiết bị DS hợp lệ
LPDIRECTSOUND8 g_pDS = NULL;
hr = DirectSoundCreate8( NULL, & g_pDS, NULL ) ;
// Thiết lập mức độ hợp tác
hr = g_pDS->SetCooperativeLevel( hwnd, DSSCL_PRIORITY );
// Kiểm tra kết quả trả về
if FAILED ( hr )
return false;
```

Ở đoạn code trên, mức độ hợp tác được thiết lập là DSSCL_PRIORITY. Trước khi bạn gọi hàm SetCooperativeLevel, bạn phải có một con trả hợp lệ chứa một thiết bị DS.

Giờ khi đã thiết lập mức độ hợp tác rồi, bạn có thể tạo bộ đệm và load dữ liệu.

Files âm thanh

Bạn có thể load dữ liệu âm thanh trong DS vào một bộ đệm sơ cấp trước khi dùng. Bạn có thể load nhạc nền hoặc hiệu ứng âm thanh vào bộ đệm tĩnh(static buffer) hoặc bộ đệm luồng(streaming buffer)

Bộ đệm tĩnh là bộ đệm có độ dài cố định có một âm thanh đầy đủ được load vào đó. Bộ đệm luồng là bộ đêm cần khi âm thanh cần load lớn hơn chỗ chứa của bộ đệm. Trong trường hợp này, một bộ đệm nhỏ được sử dụng, và âm thanh được load vào và play liên tục. Đoạn tiếp theo bàn luận về việc bộ đệm được dùng như thế nào trong DS.

Bộ đệm thứ cấp(Secondary Buffer)

DS Sử dụng bộ đệm để chứa dữ liệu audio nó cần. Trước khi bạn có thể play một âm thanh, bạn phải tạo một bộ đệm thứ cấp để chứa nó. Sau khi bộ đệm được tạo ra, âm thanh được load toàn bọ vào đó(hoặc một phần với bộ đệm luồng) và say đó được play. DS cho phép tuỳ ý lượng bộ đệm thứ cấp được play đồng thời, tất cả được trộn lẫn trong bộ đệm sơ cấp.

Trước khi bạn có thể tạo một bộ đệm sơ cấp, bạn cần biết định dạng của loại âm thanh sẽ chứa trong đó. DS yêu cầu định dạng của bộ đệm phải giống định dạng âm thanh nó chứa. Ví dụ âm thanh là file 16-bit WAV cần 2 kênh âm, thì bộ đệm cần theo định dạng này.

Hầu hết mọi lúc, các âm thanh cho game cùng một định dạng chung, cho phép bạn biết trước định dạng nào bộ đệm cần. Nếu bạn có nhiệm vụ viết một trình generic audio player, bạn sẽ ko thể đảm bảo mọi files âm thanh bạn load có cùng định dạng.

Định dạng của bộ đệm trong DS được miêu tả bằng cấu trúc WAVEFORMATEX. Cấu trúc này định nghĩa như say:

```
typedef struct {
  WORD wFormatTag;
  WORD nChannels;
  DWORD nSamplesPerSec;
  DWORD nAvgBytesPerSec;
  WORD nBlockAlign;
  WORD wBitsPerSample;
  WORD cbSize;
} WAVEFORMATEX;
```

Cấu trúc này bao gồm bảy biến:

- wFormatTag. Kiểu audio dạng sóng(waveform audio). Với dữ liệu 1 hay 2 kênh PCM, giá trị này nên để là: WAVE_FORMAT_PCM.
- nChannels. Số lượng kênh cần tới.
- nSamplesPerSec. Tốc độ mẫu.
- nAvgBytesPerSec. Tốc độ tryền dữ liệu trung bình mỗi giây.

- nBlockAlign. Sự căn chỉnh về các bytes. Bạn quyết định giá trị cần ở đây bằng cách nhân số kênh với bits per sample rồi chia cho 8.
- wBitsPerSample. Số bits per sample. Giá trị này là 8 hoặc 16.
- cbSize. Số bytes thêm vào để nối thêm dữ liệu vào cấu trúc này.

Bạn có thể tạo một cấu trúc chuẩn WAVEFORMATEX nếu bạn biết định dạng của dữ liệu file WAV bạn sử dụng. Nhưng nếu ko chắc chắn, bạn có thể tạo cấu trúc này và điền vào nó say khi mở file audio.
Cấu trúc WAVEFORMATEX chỉ là một phần của thông tin bạn cần khi tạo một bộ đệm thứ cấp. Bên cạnh ghi rõ định dạng của bộ đệm, bạn phải biết thêm các thông tin, như kích thước của dữ liệu audio và bộ đệm sẽ chứa.
Bạn cần một cấu trúc thứ hai để miêu tả bộ đệm thứ cấp: DSBUFFERDESC. Cấu trúc này định nghĩa như sau:

```
typedef struct {
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwBufferBytes;
  DWORD dwReserved;
  LPWAVEFORMATEX lpwfxFormat;
  GUID guid3DAlgorithm;
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Cấu trúc này có 6 biến thành phần:

- dwSize. Kích thước của cấu trúc DSBUFFERDESC tính bằng byte.
- dwFlags. Một tập hợp DWORD của flags, ghi rõ khả năng của bộ đệm.
- dwBufferBytes. Kích thước bộ đệm mới. Đây là số bytes dữ liệu âm mà bộ đệm này có thể chứa.
- dwReserved. Một giá trị dành riêng mà buộc phải = 0.
- lpwfxFormat. Địa chỉ của cấu trúc WAVEFORMATEX.
- Guid3Dalgorithm. Một định danh GUID cho biết thuật toán two-speaker virtualization sử dụng.

Tham số dwFlags được miêu tả chi tiết ở bảng 10.1
Bộ đệm, bên cạnh việc có một định dạng, còn có các điều khiển. Chúng cho phép bạn điều chỉnh âm lượng, tần số, và sự dịch chuyển(movement)???. Bạn phải nói rõ các kiểu điều khiển bạn muốn trong cấu trúc DSBUFFERDESC nêu trên.
Bảng 10.1 DSBUFFERDESC Flags

Giá trị	Miêu tả
DSBCAPS_CTRL3D	Bộ đệm này có điều khiển 3D
DSBCAPS_CTRLFREQUENCY	Bộ đệm này có điều khiển tần số âm
DSBCAPS_CTRLFX	BĐ hỗ trợ hiệu ứng.
DSBCAPS_CTRLPAN	BĐ có thể thay đổi âm lượng kênh

	phải và kênh trái(pan) âm thanh
DSBCAPS_CTRLPOSITIONNOTIFY	Đây là vị trí thông báo bộ đệm
DSBCAPS_CTRLVOLUME	Bạn có thể điều chỉnh âm thanh
DSBCAPS_GLOBALFOCUS	Nếu cờ này được bật thì khi người dùng chuyển focus sang ứng dụng khác, âm thanh vẫn bật
DSBCAPS_LOCDEFER	Bạn có thể đặt bộ đệm vào bộ nhớ phần cứng hoặc phần mềm vào lúc chạy
DSBCAPS_LOCHARDWARE	Bộ đệm dùng hardware mixing. Nếu cờ này bật và ko đủ bộ nhớ, lời gọi bộ đệm thất bại
DSBCAPS_LOCSOFTWARE	Bộ đệm đặt trong bộ nhớ phần mềm, và software mixing được sử dụng
DSBCAPS_MUTE3DATMAXDISTANCE	Âm thanh trong bộ đệm này được giảm xuống khi vị trí ảo của nó càng xa.
DSBCAPS_PRIMARYBUFFER	Đây là bộ đệm sơ cấp
DSBCAPS_STATIC	Bộ đệm được đặt trên bộ nhớ phần cứng on-board
DSBCAPS_STICKYFOCUS	Khi bạn chuyển focus sang ứng dụng khác, bạn vẫn có thể nghe bộ đệm có stickyfocus. Bộ đệm thường sẽ ko kêu khi khi điều này xảy ra

Tạo bộ đệm thứ cấp(secondary buffer)

Giờ bạn đã tạo cấu trúc DSBUFFERDESC, bạn đã sẵn sàng tạo bộ đệm thứ cấp thực sự. Bộ đệm thứ cấp được tạo với lời gọi CreatSoundBuffer, định nghĩa như sau:

```
HRESULT CreateSoundBuffer(
LPCDSBUFFERDESC pcDSBufferDesc,
LPDIRECTSOUNDBUFFER * ppDSBuffer,
LPUNKNOWN pUnkOuter
);
```

Hàm này chỉ cần 3 tham số:

- pcDSBufferDesc. Địa chỉ một cấu trúc đã xác định DSBUFFERDESC.
- ppDSBuffer. Địa chỉ biến sẽ chứa bộ đệm mới tạo.
- pUnkOuter. Địa chỉ tới điều khiển giao tiếp IunKnown của đối tượng. Giá trị này nên để là NULL.

Một lời gọi mẫu của hàm như sau:

```
// định nghĩa một cấu trúc WAVEFORMATEX
WAVEFORMATEX wfx;
// Khởi trị cấu trúc tất cả về 0.
```

```
ZeroMemory( &wfx, sizeof(WAVEFORMATEX) );
// Thiết lập định dạng là WAVE_FORMAT_PCM
wfx.wFormatTag = (WORD) WAVE_FORMAT_PCM;
// Thiết lập số kênh âm thanh là 2
wfx.nChannels = 2;
// Thiết lập tốc độ mẫu là 22050
wfx.nSamplesPerSec = 22050;
// Tính giá trị nBlockAlign
wfx.wBitsPerSample = 16;
wfx.nBlockAlign = (WORD) (wfx.wBitsPerSample / 8 * wfx.nChannels);
// Tính giá trị the nAvgBytesPerSec
wfx.nAvgBytesPerSec = (DWORD) (wfx.nSamplesPerSec *
wfx.nBlockAlign);
// Định nghĩa một cấu trúc DSBUFFERDESC
DSBUFFERDESC dsbd;
// Khởi trị cấu trúc tất cả về 0
ZeroMemory( &dsbd, sizeof(DSBUFFERDESC) );
// Thiết lập kích thước cấu trúc
dsbd.dwSize = sizeof(DSBUFFERDESC);
// Thiết lập các cờ
dsbd.dwFlags = 0;
// kích thước bộ đệm
dsbd.dwBufferBytes = 64000;
// GUID của thuật toán
dsbd.guid3DAlgorithm = GUID_NULL;
// địa chỉ cấu trúc WAVEFORMATEX
dsbd.lpwfxFormat = &wfx;
// Định nghĩa biến lưu bộ đệm mới tạo
LPDIRECTSOUNDBUFFER DSBuffer = NULL;
// Tạo bộ đệm mới
hr = g_pDS->CreateSoundBuffer( &dsbd, &DSBuffer, NULL );
// Kiểm tra code trả về để chắc chắc lời gọi hàm CreatSoundBuffer thành
công
if FAILED (hr)
return NULL;
```

Nếu lời gọi hàm CreatSoundBuffer thành công, biến DSBuffer sẽ là một bộ đệm DS hợp lệ. Trong ví dụ trên, định dạng của cấu trúc WAVEFORMATEX đã được hard-coded, buộc tất cả các files được load vào bộ đệm này phải có kiểu định dạng xác định và chỉ tối đa dài 64000 bytes.

Load một file âm thanh vào bộ đệm

Giờ bạn đã tạo bộ đệm, bạn cần load dữ liệu âm thanh vào nó. Load dữ liệu âm thanh vào bộ đệm cần bạn trước hết phải mở file chứa dữ liêu đó,

rồi copy nội dung của nó vào bộ đệm đã tạo. Với bộ đệm tĩnh(static buffer), tất cả dữ liệu được copy vào bộ đệm.
Bởi bộ đệm âm thanh là một diện tích bộ nhớ điều khiển bởi DS, bạn phải khoá(lock) nó trước khi viết lên nó. Khoá bộ đệm sẽ chuẩn bị bộ nhớ để được viết vào. Sau khi khoá bộ đệm, chương trình có thể load âm thanh vào nó. Khi bạn hoàn thành việc load, bạn phải nhớ mở khoá(unlock) nó. Mở khoá bộ đệm cho phép DS chế tác nội dung của nó một lần nữa.
Khoá(Lock) bộ đệm
Khoá bộ đệm âm thanh cho code một cơ hội chế tác, thay đổi dữ liệu âm thanh trong bộ đệm. Khoá bộ đệm cần hàm Lock, như sau:

```
HRESULT Lock(
  DWORD dwOffset,
  DWORD dwBytes,
  LPVOID * ppvAudioPtr1,
  LPDWORD pdwAudioBytes1,
  LPVOID * ppvAudioPtr2,
  LPDWORD pdwAudioBytes2,
  DWORD dwFlags
);
```

Hàm Lock yêu cầu 7 tham số:
- dwOffset. Biến này chỉ định vị trí trong bộ đệm nên được bắt đầu lock.
- dwBytes. Số bytes bên trong bộ đệm để lock.
- ppvAudioPtr1. Biến này nhận một con trỏ tới phần đầu của bộ đệm bị lock.
- pdwAudioBytes1. Biến này nhận số bytes trong khối nhớ của con trỏ ppvAudioPtr1.
- ppvAudioPtr2. Biến này nhận một con trỏ tới phần thứ hai của bộ đệm bị lock. Nếu bạn điền vào cả bộ đệm với dữ liệu âm thanh, biến này nên để NULL.
- pdwAudioBytes2. Biến này nhận số bytes trong khối nhớ của con trỏ ppvAudioPtr2. Biến này nên để NULL nếu bạn điền vào cả bộ đệm với dữ liệu âm thanh.
- dwFlags. Đây là các cờ(Flags) chỉ định lock nên diễn ra như thế nào:

. DSBLOCK_FROMWRITECURSOR. Bắt đầu lock từ con trỏ ghi(write cursor).
. DSBLOCK_ENTIREBUFFER. Lock cả bộ đệm. Nếu cờ này được bật, biến dwBytes sẽ được bỏ qua.

Mở khoá(unlock) bộ đệm
Lúc này, bạn được tuỳ ý đọc dữ liệu âm thanh và load nó vào bộ đệm. Sau khi quá trình này hoàn thành, bạn có thể mở khoá(unlock) bộ đệm bằng hàm Unlock, như sau:

```
HRESULT Unlock(
```

```
  LPVOID pvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID pvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Hàm Unlock cần 4 tham số:

- pvAudioPtr1. Địa chỉ của giá trị tham số ppvAudioPtr1 dùng trong hàm Lock.
- dwAudioBytes1. Số bytes viết vào pvAudioPtr1.
- pvAudioPtr2. Địa chỉ của giá trị tham số ppvAudioPtr2 dùng trong hàm Lock.
- dwAudioBytes2. Số bytes viết vào pvAudioPtr2.

Đọc từ dữ liệu âm thanh vào bộ đệm

Đọc dữ liệu âm thanh vào bộ đệm thứ cấp (secondary buffer) có thể rất phức tạp. Để giải thích dễ hiểu hơn, tôi sẽ giải thích quá trình này qua lớp CWaveFile trong các lớp DS framework. DS Framework cung cấp một cách đơn giản để load dữ liệu âm thanh sử dụng định dạng file WAV. Đây là định dạng âm thanh mặc định của Windows, có đuôi là WAV.

Chú ý:

Các lớp DS framework được khai báo trong file dsutil.cpp và dsutil.h, cung cấp các hàm chung gắn liền với DS. Bạn có thể tìm thấy chúng trong thư mục Samples\C++\Common\Src và Samples\C++\Common\Inc, trong folder mà bạn đã cài DirectX Software Development Kit (SDK).

Bước đầu tiên trong việc load một file WAV vào bộ đệm là tạo một đối tượng CWaveFile. Đối tượng này cung cấp cho bạn các phương thức để mở, đóng và đọc một files WAV. Dòng code sau chỉ ra cách tạo một đối tượng CWaveFile.

```
CWaveFile wavFileObj = new CWaveFile( );
```

Tiếp theo, sử dụng phương thức Open được cung cấp bởi lớp CWaveFile, bạn có thê nhận quyền truy cập vào file WAV bạn muốn dùng. Đoạn code sau sử dụng hàm Open và kiểm tra xem file WAV có chứa dữ liệu ko?

```
// Mở 1 file dạng WAV có tên test.wav
wavFile->Open("test.wav", NULL, WAVEFILE_READ );
// Kiểm tra để chắc chắn kích thước dữ liệu trong file là hợp lệ
if( wavFile->GetSize( ) == 0 )
return false;
```

Đoạn code trên mở một file tên là test.wav để đọc. Sau đó nó kiểm tra kích thước của dữ liệu bên trong file, nếu file ko chứa dữ liệu, nó dừng việc đọc. Bước tiếp theo là tạo một bộ đệm thứ cấp (secondary buffer) để chứa dữ liệu. Bước này đã được hướng dẫn ở trên. Sau khi tạo bộ đệm, bạn phải khoá(lock) nó trước khi bạn có thể viết dữ liệu WAV vào nó. Đoạn code tiếp theo giải thích cách dùng hàm Lock để chuẩn bị một bộ đệm đọc toàn bộ một file WAV.

```
HRESULT hr;
VOID* pDSLockedBuffer = NULL; // con trỏ tới vùng nhớ bộ đệm bị khoá
DWORD dwDSLockedBufferSize = 0; // kích thước bộ đệm DS bị khoá
// Bắt đầu từ phần đầu của bộ đệm
hr = DSBuffer->Lock( 0,
                     // Nhận một bộ đệm 64000 bytes
64000,
                     // Biến này lưu một con trỏ tới phần mở đầu
                  // của bộ đệm
                  &pDSLockedBuffer,
                     // Biến này lưu kích thước của bộ đệm bị khoá
                     &dwDSLockedBufferSize,
                     NULL,// Ko cần cái thứ hai
                     NULL, // Ko cần cái thứ hai
                     // Khoá cả bộ đệm
                     DSBLOCK_ENTIREBUFFER);
                     // Kiểm tra code trả về để chắc chắc lock
                  // thành công
if FAILED (hr)
  return NULL;
```

Đoạn code trên lock một bộ đệm bằng cờ DSBLOCK_ENTIREBUFFER. Cờ này khiến bộ đệm bị lock toàn bộ. Biến DSBuffer phải là một DirectSoundBuffer hợp lệ.

Giờ bộ đệm đã được lock đúng, bạn có thể ghi dữ liệu WAV vào nó. Một lần nữa tôi lại dùng phương thức có trong lớp CWaveFile. Trước khi bạn đọc dữ liệu WAV vào bộ đệm, bạn phải reset lại dữ liệu WAV về phần đầu. Bạn làm điều này bằng cách dùng phương thức ResetFile. Tiếp theo bạn dùng phương thức Read để đưa dữ liệu WAV vào bộ đệm. Đoạn code tiếp theo reset lại file WAV để đọc và đưa dữ liệu vào bộ đệm.

```
HRESULT hr; // biến lưu kết quả trả lại
DWORD dwWavDataRead = 0; // biến chứa lượng dữ liệu đọc từ file
                         // WAV
wavFile->ResetFile( ); // Reset lại file WAV về đầu file
// Đọc file
hr = wavFile->Read( ( BYTE* ) pDSLockedBuffer,
dwDSLockedBufferSize,
&dwWavDataRead );
// Kiểm tra để chắc chắn đã thành công
if FAILED (hr)
return NULL;
```

Biến wavFile phải chứa một đối tượng CWaveFile hợp lệ trước khi sử dụng. Đầu tiên, hàm ResetFile được gọi, tiếp theo là lời gọi hàm Read. Hàm Read cần ba tham số. Tham số thứ nhất là con trỏ tới vùng bộ nhớ

chứa bộ đệm để copy dữ liệu vào. Tham số tiếp theo là kích thước của vùng bộ đệm đã khoá, tham số cuối nhận khối lượng dữ liệu đọc từ file WAV, tính bằng bytes.
Sau khi gọi hàm Read, bộ đệm được điền bằng dữ liệu từ file WAV. Bạn giờ đây có thể an toàn mở khoá bộ đệm.
Play âm thanh trong bộ đệm
Giờ bạn đã có dữ liệu âm thanh hợp lệ chứa trong DirectSoundBuffer, bạn có thể play dữ liệu mà nó chứa. Sau toàn bộ các công việc cần để tạo bộ đệm và điền dữ liệu vào đó, giừo việc play nó rất đơn giản. Một hàm đơn giản là Play hoàn thành việc này. Hàm này là một phương thức cung cấp bởi đối tượng DirectSoundBuffer. Nó như sau:

```
HRESULT Play(
  DWORD dwReserved1,
  DWORD dwPriority,
  DWORD dwFlags
);
```

Hàm này cần ba tham số:
- dwReserved1. Một giá trị để dành trước phải là 0.
- dwPriority. Mức độ ưu tiên để play âm thanh. Nó có thể là bất cứ giá trị nào trong khoảng 0 và 0xFFFFFFFF. Bạn phải đặt mức độ ưu tiên về 0 nếu cờ DSBCAPS_LOCDEFER chưa được bật khi bộ đệm được tạo ra.
- dwFlags. Các cờ chỉ rõ âm thanh sẽ được play thế nào? Cờ duy nhất tôi sẽ giải thích ở đây là DSBPLAY_LOOPING. Cờ này khiến âm thanh lặp lại khi đạt đến cuối bộ đệm. Nếu âm thanh chỉ nên play một lần, tahm số dwFlags nên được truyền giá trị 0.

Đoạn code dưới đây khiến 1 bộ đệm âm thanh play nội dung của nó:

```
DSBuffer->Play( 0, 0, DSBPLAY_LOOPING);
```

Biến DSBuffer phải chứa một đối tượng DirectSoundBuffer hợp lệ, chứa dữ liệu. Trong trường hợp này, cờ DSBPLAY_LOOPING được truyền vào hàm khiến âm thanh lặp lại sau khi hoàn thành một lượt.
Dừng âm thanh
Thường sau khi play âm thanh, bạn ko cần lo gì nữa, trừ khi bạn bảo âm thanh lặp lại. Trong trường hợp này, bạn cần ra lệnh dừng âm thanh. Bạn làm điều này nhờ phương thức Stop, cung cấp bởi đối tượng DirectSoundBuffer, như sau:

```
HRESULT Stop( );
```

Hàm này ko cần tham số. Nó chỉ trả lại một kết quả cho biết hàm được gọi thành cồng hay ko?
Bạn có thể tìm thấy một ví dụ đầy đủ về load một file âm thanh và play nó trong thư mục chapter10\example2 trên đĩa.
Sử dụng các điều khiển của bộ đệm(buffer control)
Như đã đề cập, bộ đệm DS có thể điều khiển diện mạo của những âm thanh trong nó. Ví dụ, qua một bộ đệm, bạ có thể thay đổi âm lượng, tần

số, hoặc pan(chỉnh âm lượng giữa hai loa) một âm thanh. Trong đoạn này bạn sẽ học cách dùng chúng.

Thay đổi âm lượng

Bạn có thể thay đổi âm lượng của âm thanh qua bộ đệm chứa nó. Bạn có thể điều chỉnh âm lượng trong khoảng giá trị DSBVOLUME_MIN và DSBVOLUMEMAX. Giá trị DSBVOLUME_MIN miêu tả im lặng và DSBVOLUME_MAX miêu tả âm lượng gốc của âm thanh.

Chú ý:

DS ko hỗ trợ khuếch đại âm, vì vậy bạn ko thể tăng âm lượng được.

Bạn có thể điều chỉnh âm lượng của âm thanh qua hàm SetVolume như sau:

```
HRESULT SetVolume (
  LONG lVolume
);
```

Hàm này chỉ cần một tham số: lVolume. Bạn có thể cho giá trị của nó từ -10000(DSBVOLUME_MIN) và 0(DSBVOLUME_MAX).

Bạn cũng có thể nhận giá trị âm lượng mà một âm thanh đang được play bằng hàm Getvolume. Hàm này như sau:

```
HRESULT GetVolume (
  LPLONG plVolume
);
```

Hàm này cũng chỉ cần một tham số, là một con trỏ tới một biến sẽ nhận giá trị âm thanh hiện tại.

Chú ý:

Trước khi dùng hai hàm trên, bạn phải thiết lập bộ đệm cho phép dùng chúng, bằng cách bật cờ DSBCAPS_CTRLVOLUME trong cấu trúc DSBUFFERDESC khi bạn tạo bộ đệm thứ cấp (secondary buffer).

Panning âm thanh(chỉnh âm lượng giữa hai loa)

Bộ đệm DS cho phép một âm thanh được pan giữa hai loa. Pan là giảm âm lượng một loa và tăng ở bên kia. Âm thanh nghe như di chuyển vậy. Pan dùng một tư tưởng chung như hàm SetVolume. Loa trái và loa phải có thể tăng hoặc giảm âm lượng phụ thuộc hai giá trị: DSBPAN_LEFT và DSBPAN_RIGHT.

Giá trị đầu DSBPAN_LEFT, tương đương -10000, tăng âm lượng của loa trái tới full, trong khi làm im loa kia. Và giá trị DSBPAN_RIGHT, tương đương 10000, tăng âm lượng loa phải trong khi làm câm loa trái. Bằng cách dùng giá trị giữa DSBPAN_LEFT và DSBPAN_RIGHT, âm thanh có thể bị pan từ một loa sang loa kia.

Một giá trị thứ ba, là DSBPAN_CENTER, bằng 0, chỉnh cả hai tới full âm lượng.

Giá trị pan âm thanh có thể được đặt bằng hàm SetPan, như sau:

```
HRESULT SetPan(
  LONG lPan
```

);
Hàm này chỉ cần một tham số: lPan, nhận giá trị giữa DSBPAN_LEFT và DSBPAN_RIGHT(-10000->10000).
Nếu bạn muốn biết giá trị pan hiện tại, bạn dùng hàm GetPan:
HRESULT GetPan(
  LPLONG plPan
);
Hàm này cũng chỉ cần một tham số: plPan, là con trỏ tới một biến LONG nhận giá trị pan hiện tại.
Chú ý:
Trước khi dung hai hàm này, bạn phải thiết lập bộ đệm cho phép dùng. Bạn cần đặt cờ DSBCAPS_CTRLPAN trong cấu trúc DSBUFFERDESC khi tạo bộ đệm.

Tổng kết chương

Dùng những gì vừa học, bạn có thể play nhạc nền hoặc các hiệu ứng âm thanh đơn giản cho game. Bạn có thể mở rộng bài học này để play nhiều âm thanh đồng thời, hay tạo các nhạc động, có thể sửa và điều khiển trong game.
Trong chương sau, chúng ta sẽ đặt những thứ đã học với nhau để tạo một game đơn giản sử dụng mỗi phần đã nói trong sách.

Những thứ bạn đã học

Trong chương này, bạn đã học:
- DS dùng thế nào?
- Có những kiểu bộ đệm nào?
- Liệt kê các thiết bị âm thanh đã có trong hệ thống.
- Load và Play một file WAV.
- Điều khiển sự phát lại(playback) của một file âm thanh.

Câu hỏi ôn tập

Bạn có thể tìm câu trả lời trong phụ lục A “Answers to End-of-Chapter Exercises”.
1. Khi nào bạn cần dùng hàm DirectSoundEnumerate?
2. Ba mẩu dữ liệu quan trọng nào được truyền cho hàm callback liệt kê?(enumeration callback function)?
3. Định dạng của một bộ đệm có cần giống định dạng của dữ liệu nó chứa ko?
4. Mục đích bộ đệm sơ cấp (primary buffer)?
5. Giá trị nào được truyền cho hàm DirectSoundCreate8 để chỉ rõ thiết bị âm thanh mặc định được dùng?

Về phần bạn

1. Viết một cí dụ cho phéo điều chỉnh âm lượng âm thanh khi đang play nó?
2. Viết một ví dụ nhỏ cho phép âm thanh pan sử dụng các phím mũi tên.