Chủ đề cuốn sách này

Hệ thống đồ hoạ OpenGL là một giao diện phần mềm đến phần cứng đồ hoạ (GL viết tắt của Graphics Library_thư viện đồ hoạ). Nó cho phép bạn tạo các chương trình tương tác mà tạo ra các hình ảnh màu các đối tượng ba chiều chuyển động. Với OpenGL, bạn có thể điều khiển công nghệ đồ hoạ máy tính để tạo các ảnh thực hay những ảnh không có thực theo những cách tưởng tượng khác nhau. Cuốn sách này giải thích cách lập trình với hệ thống đồ hoạ OpenGL để có được những hiệu ứng trực quan mà bạn mong muốn.

Cuốn sách này có những gì

Cuốn sách này bao gồm 14 chương. Năm chương đầu trình bày những thông tin cơ bản bạn cần phải biết để có thể vẽ một màu đúng cách và chiếu sáng đối tượng ba chiều trên màn hình.

Chương 1, "**Giới thiệu OpenGL**, " cung cấp một cái nhìn thoáng qua về những gì OpenGL có thể làm được. Nó cũng đưa ra một chương trình OpenGL đơn giản và giải thích bản chất chi tiết chương trình bạn muốn biết cho những chương tiếp theo.

Chương 2, "**Quản lí lệnh và vẽ các đối tượng hình học,**" giải thích làm thế nào để tạo một mô tả hình học ba chiều của một đối tượng mà cuối cùng được vẽ trên màn hình.

Chương 3, "**Quan sát**," mô tả các mô hình ba chiều đó được biến đổi như thế nào trước khi vẽ sang màn hình hai chiều. Bạn có thể điều khiển các biến đổi này để chỉ ra một quan sát cụ thể của một mô hình.

Chương 4, "**Màu sắc**," mô tả làm thế nào để chỉ rõ màu và các phương pháp tô bóng được sử dụng để vẽ một đối tượng.

Chương 5, "**Ánh sáng**," giải thích cách điều khiển các điều kiện ánh sáng bao quanh một đối tượng và đối tượng đó phản ứng như thế nào với ánh sáng (như là, cách nó phản chiếu hay hấp thụ ánh sáng). Ánh sáng là một chủ đề quan trọng, vì đối tượng sẽ không thể thấy trong không gian ba chiều trừ phi nó được chiếu sáng.

Các chương còn lại giải thích cách để tối ưu và thêm các đặc điểm phức tạp cho cảnh ba chiều của bạn. Bạn có thể không chọn cải tiến các đặc điểm này cho đến khi bạn thấy thành thạo hơn với OpenGL. Chủ đề cải tiến đặc biệt được chú thích trong văn bản khi chúng xuất hiện.

Chương 6, "**Trộn, khử răng cưa, mờ, và khối đa giác**", mô tả kĩ thuật cần thiết để tạo một pha trộn cảnh thực alpha (để tạo các đối tượng trong suốt), khử răng cưa (để loại bỏ răng cưa), hiệu ứng không khí (để mô phỏng sương mù hay khói), và khối đa giác (để xoá tạo tác trực quan khi chiếu nổi bật các cạnh của đa giác phủ kín)

Chương 7, "**Danh sách hiển thị**", trình bày cách lưu trữ một tập hợp các lệnh để tiến hành cho lần sau. Bạn sẽ muốn sử dụng đặc điểm này để tăng tính thực thi chương trình của bạn.

Chương 8, "**Vẽ điểm ảnh, Bitmaps, phông chữ, và các hình ảnh**," trình bày cách làm việc với tập hợp dữ liệu hai chiều như bitmaps hay hình ảnh. Một sử dụng điển hình cho bitmap là mô tả kí tự theo phông chữ.

Chương 9, "**Ánh xạ kết cấu**", giải thích cách để ánh xạ các hình ảnh một và hai chiều mà được gọi là kết cấu lên trên các đối tượng ba chiều. Nhiều hiệu ứng kì diệu có thể được kích hoạt thông qua ánh xạ kết cấu.

Chương 10, "**Đệm khung,"** mô tả tất cả vùng đệm có thể mà chúng tồn tại trong một cài đặt OpenGL và bạn có thể điều khiển chúng như thế nào. Bạn có thể sử dụng các vùng đệm cho các hiệu ứng đó như là chủ đề khử mặt khuất, mẫu tô, mặt nạ, chuyển động mờ, và phạm vi độ sâu.

Chương 11, " **Khảm và bậc hai**," chỉ ra làm thế nào để sử dụng khảm và thủ tục bậc hai trong GLU (OpenGL Utility Library_ thư viện tiện ích OpenGL)

Chương 12, "**Cách đánh giá và NURBS**," giới thiệu các kĩ thuật cải tiến cho các đường cong và bề mặt tổng quát một cách hiệu quả.

Chương 13, "**Lựa chọn và phản hồi**," giải thích cách bạn có thể sử dụng cơ chế lựa chọn của OpenGL để lựa chọn một đối tượng trên màn hình. Nó cũng giải thích cơ chế phản hồi, chúng cho phép bạn thu thập thông tin OpenGL vẽ ra thay vì việc nó phải được sử dụng để vẽ trên màn hình.

Chương 14, "**Bây giờ đó là những gì bạn biết**"mô tả cách sử dụng OpenGL theo một số cách thông minh và bất ngờ để tạo ra kết quả thú vị. Những kĩ thuật này được vẽ từ nhiều năm kinh nghiệm với hai OpenGL và tiền kĩ thuật OpenGL, the Silicon Graphics IRIS Graphics Library.

Thêm nữa, có một số phụ lục mà bạn sẽ tìm thấy hữu ích.

Phụ lục A, **"Trình tự hoạt động,"**, đưa ra một kĩ thuật tổng quát của các hoạt động OpenGL thực hiện, mô tả ngắn gọn chúng theo thứ tự mà chúng xuất hiện như một ứng dụng thực thi.

Phụ lục B, **"Các biến trạng thái,"** danh sách các biến trạng thái mà OpenGL duy trì và mô tả làm thế nào để thu được giá trị của chúng

Phụ lục C, **"OpenGL và các hệ thống Window,"** mô tả ngắn gọn các thủ tục có sẵn trong các thư viện cụ thể hệ thống Window, chúng được mở rộng để hỗ trợ kết xuất OpenGL. Giao diện hệ thống Window đến X Window System, Apple MacIntosh, IBM OS/2, và Microsoft Windows NT và Windows 95 được trình bày ở đây.

Phụ lục D, **"Cơ sở của GLUT: The OpenGL Utility Toolkit,"** trình bày thư viện mà điều khiển các hoạt động hệ thống cửa sổ, GLUT có thể linh động và nó tạo ra các đoạn mã ví dụ ngắn hơn và dễ hiểu hơn.

Phụ lục E, **"Tính toán các vectơ pháp tuyến,"** chỉ cho bạn biết cách tính vectơ pháp tuyến như thế nào cho những kiểu khác nhau của các đối tượng hình học.

Phụ lục F, **"Hệ toạ độ đồng nhất và các ma trận biến đổi,"** giải thích một số phép toán đằng sau các biến đổi ma trận

Phụ lục G, **"Mẹo lập trình,"** danh sách một số mẹo lập trình dựa trên ý định của các nhà thiết kế OpenGL mà bạn có thể thấy hữu ích.

Phụ lục H, **"OpenGL bất biến,"** mô tả một cài đặt OpenGL phải tạo các giá trị điểm ảnh chính xác khi nào và ở đâu trong đặc tả OpenGL.

Phụ lục I, **"Các bản màu,"** gồm các bản màu mà xuất hiện trong bản in của cuốn sách này.

Cuối cùng, một bản chú giải thuật ngữ khó mở rộng định nghĩa các từ khoá sử dụng trong cuốn sách này.

Phiên bản này có gì mới

Với câu hỏi, "xuất bản lần này có gì mới" câu trả lời là "khoảng 100 trang", những thông tin được thêm như sau:

- Thông tin chi tiết về đặc điểm mới sau đây của OpenGL phiên bản 1.1 đã được thêm
  - Mảng Vertex
  - Cải tiến kết cấu, bao gồm các đối tượng kết cấu (bao gồm cư trú và ưu tiên), định dạng ảnh kết cấu bên trong, kết cấu các phần ảnh nhỏ, kết cấu uỷ nhiệm, và sao chép kết cấu từ vùng đệm khung hệ thống.
  - offset đa giác.
  - Phép toán logic trong chế độ RGBA
- Các ví dụ chương trình đã được chuyển đổi thành GLUT của Mark Kilgard, chúng viết tắt của Graphics Library Utility Toolkit. GLUT là một bộ công cụ cửa sổ phổ biến, chúng được chứng minh và đã được chuyển đổi đến các hệ thống cửa sổ khác nhau.

Chi tiết hơn về một số chủ đề mà đã xuất bản lần đầu, đặc biệt là OpenGL Utility (GLU) Library An entire chapter on GLU tessellators and quadrics

- Một chương hoàn toàn về khảm GLU và bậc hai
- Một phần (trong Chương 3) với việc sử dụng **gluProject()** và **gluUnProject()**, chúng bắt chước hoặc đảo ngược lại hoạt động của quy trình xử lí hình học (đây đã là chủ đề thảo luận thường xuyên của nhóm thông tin Internet trên OpenGL, *comp.graphics.api.opengl*)
- Một thông tin mở rộng (và lược đồ lớn hơn) về các hình ảnh

Thay đổi các thuộc tính GLU NURBS Error handling and vendor-specific extensions to OpenGL

- Điều khiển lỗi và mở rộng nhà cung cấp cụ thể cho OpenGL.
- Phụ lục C mở rộng bao gồm giao diện OpenGL đến một số các hệ thống window/hệ điều hành.

Phụ lục của xuất bản lần đầu trên OpenGL Utility Library đã bị xoá, và thông tin của nó được tích vào các chương khác.

- Một chỉ số thông tin quan trọng hơn và lớn hơn nhiều
- Sửa lỗi và sắp đặt lại chủ đề nhỏ. Di chuyển chương danh sách hiển thị là thay đổi dễ nhận thấy nhất.

Bạn nên biết gì trước khi đọc cuốn sách này

Cuốn sách này giả sử rằng bạn biết lập trình trong ngôn ngữ C và bạn có một số kiến thức nền toán học (hình học, lượng giác, đại số tuyến tính, tính toán và hình học vi phân)

Thậm chí nếu bạn có một chút hoặc không kinh nghiệm với kĩ thuật đồ hoạ máy tính, bạn có thể theo được hầu hết trình bày trong cuốn sách này. Tất nhiên, đồ hoạ máy tính là một môn học lớn, do đó bạn có thể muốn nâng cao kiến thức môn học của bạn với việc đọc bổ sung.

- *Computer Graphics: Principles and Practice* by James D. Foley, Andries van Dam, Steven K.Feiner, và John F. Hughes (Reading, MA: Addison-Wesley, 1990) – Cuốn sách này là cuốn bách khoa toàn thư của môn đồ hoạ máy tính. Nó chứa phong phú thông tin nhưng có lẽ tốt nhất để đọc là sau khi bạn có một số kiến thức về môn học này.
- *3D Computer Graphics: A User's Guide for Artists and Designers* by Andrew S. Glassner (NewYork: Design Press, 1989) – Cuốn sách này là phi kĩ thuật, ít giới thiệu về đồ hoạ máy tính. Nó tập trung vào hiệu ứng trực quan mà có thể được kích hoạt thay vì các kĩ thuật cần thiết để kích hoạt chúng.

Mỗi khi bạn bắt đầu lập trình với OpenGL, bạn có thể muốn có *OpenGL Reference Manual* bởi OpenGL Architecture Review Board (Reading, MA: Addison-Wesley Developers Press, 1996), nó được thiết kế như một cuốn sách đồng hành với cuốn sách này. *Reference Manual* cung cấp một kĩ thuật quan sát của OpenGL hoạt động như thế nào trên dữ liệu mà chúng mô tả một đối tượng hình học hay một hình ảnh để tạo một ảnh trên màn hình. Nó cũng bao gồm mô tả đầy đủ từng tập hợp các lệnh OpenGL liên quan- các biến sử dụng bởi các lệnh, các giá trị ngầm định cho các biến đó, và những gì các lệnh phải hoàn thiện.

Nhiều cài đặt OpenGL có cùng tài liệu trực tuyến, theo dạng của các trang *man* hay tài liệu trợ giúp khác, và nó chắc chắn cập nhật theo ngày. Cũng có một phiên bản khác http khác trên web; tham khảo Silicon Graphics OpenGL Web Site

(http://www.sgi.com/Technology/openGL)

OpenGL thực sự là một đặc tả độc lập phần cứng của một giao diện lập trình, và bạn sử dụng cài đặt cụ thể trên một loại phần cứng cụ thể. Cuốn sách này giải thích cách lập trình như thế nào với mọi cài đặt OpenGL. Tuy nhiên, do cài đặt có thể khác nhau một chút – khi thi hành và khi cung cấp thêm, đặc điểm tuỳ ý, ví dụ bạn có thể muốn tích hợp hay không tài liệu bổ sung có sẵn cho cài đặt

cụ thể bạn đang sử dụng. Thêm nữa, bạn có thể có các tiện tích liên quan OpenGL, bộ công cụ, hỗ trợ lập trình và gỡ lỗi, chương trình mẫu và chương trình giới thiệu có sẵn với hệ thống của bạn.

Làm thế nào để có đoạn mã mẫu

Cuốn sách này chứa nhiều chương trình mẫu để minh hoạ việc sử dụng kĩ thuật lập trình OpenGL cụ thể. Các chương trình này sử dụng OpenGL Utility Toolkit của Mark Kilgard (GLUT). GLUT được dẫn chứng tài liệu trong *OpenGL Programming for the X Window System* viết bởi Mark Kilgard (Reading, MA: Addison-Wesley Developers Press, 1996). Phần "thư viện liên quan OpenGL" trong Chương 1 và phục lục D cung cấp thêm thông tin về việc sử dụng GLUT. Nếu bạn truy cập vào Internet, bạn có thể thu được đoạn mã nguồn cho cả chương trình mẫu và GLUT miễn phí thông qua nặc danh ftp (file transfer protocol - giao thức chuyển đổi tệp tin)

Đối với các ví dụ nguồn trong cuốn sách này, truy cập tệp tin này

```
ftp://sgigate.sgi.com/pub/opengl/opengl1_1.tar.Z
```

Các tệp tin bạn nhận được này được nén trong đuôi *tar*. Để giải nén và bung các tệp tin, gõ

```
uncompress opengl1_1.tar
```

```
tar xf opengl1_1.tar
```

Đối với mã nguồn của Mark Kilgard cho phiên bản X Window System của GLUT, bạn cần biết hầu hết các phiên bản hiện nay làm được những gì. Tên tệp tin sẽ là *glut-i.j.tar.Z,* với *i* là số lần sửa lại chính và *j* là số sửa đổi nhỏ trong phiên bản mới nhất. Kiểm tra thư mục cho số bên phải, sau đó truy cập tệp tin này

```
ftp://sgigate.sgi.com/pub/opengl/xjournal/GLUT/glut-i.j.tar.Z
```

Tệp tin này cũng phải được giải nén và trích chọn bằng việc sử dụng lệnh *tar*. Các chương trình mẫu và thư viện GLUT được tạo như thư mục con từ bất kì vị trí nào trong cấu trúc thư mục tệp tin.

Cổng khác của GLUT (ví dụ, đối với Microsoft Windows NT) là đàn hồi. Một địa điểm tốt để bắt đầu nghiên cứu đối với phát triển mới nhất GLUT và OpenGL, tổng quát, là trang Web OpenGL của Silicon Graphics

```
http://www.sgi.com/Technology/openGL
```

Nhiều cài đặt của OpenGL có thể cũng bao gồm đoạn mã mẫu như phần của hệ thống. Mã nguồn này hầu như chắc chắn là nguồn tốt nhất cho cài đặt của bạn, bởi vì nó có thể được tối ưu trong hệ thống của bạn. Đọc tài liệu OpenGL cụ thể trong máy của bạn để hiểu đoạn mã mẫu có thể tìm thấy ở đâu.

Lỗi viết

Mặc dù cuốn sách này là lí tưởng và hoàn hảo, có một số lỗi viết chỉ ra trong trang web Silicon Graphics OpenGL

http://www.sgi.com/Technology/openGL

Tác giả khá chắc chắn ở đây sẽ có một số chú thích nhỏ để trấn an người đọc về chất lượng cuốn sách này

Các quy ước về kiểu

Các quy ước về kiểu được sử dụng trong cuốn sách này:

Bold – tên câu lệnh, thủ tục và các ma trận

Italics – Các biến, đối số, các tên biến, chiều không gian, thành phần ma trận và từ khoá xuất hiện đầu tiên.

Regular – Kiểu liệt kê và định nghĩa hằng.

Các ví dụ mã được đặt ra văn bản theo phông chữ đơn cách, và tóm tắt lệnh được dùng chung với các hộp xám.

Trong một tóm tắt lệnh, dấu ngoặc móc được sử dụng để chỉ lựa chọn giữa các kiểu hệ thống. Trong ví dụ sau đây, **glCommand** có bốn hậu tố: s, i, f, và d, chúng có nghĩa cho các kiểu hệ thống GLshort, GLint, GLfloat, và Gldouble. Trong hàm mẫu ban đầu **glCommand**, TYPE là một kí tự đại diện mà biểu diễn kiểu dữ liệu biểu thị bởi hậu tố

*void **glCommand**{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);*

Lời cảm ơn

Lần xuất bản thứ hai của cuốn sách này cần đến sự trợ giúp của nhiều các nhân. Việc thúc đẩy cho lần xuất bản thứ hai bắt đầu với Paula Womack và Tom McReynolds của Silicon Graphics, người công nhân sự cần thiết cho việc xem lại và cũng phân bổ một số tài liệu mới. John Schimpf, OpenGL Product Manager ở Silicon Graphics, là công cụ trong việc sửa đổi và chạy. Cám ơn nhiều người tại Silicon Graphics: Allen Akin, Brian Cabral, Norman Chin, Kathleen Danielson, Craig Dunwoody, Michael Gold, Paul Ho, Deanna Hohn, Brian Hook, Kevin Hunter, David Koller, Zicheng Liu, Rob Mace, Mark Segal, Pierre Tardif, và David Yu đối với việc xây dựng xâm nhập và những câu hỏi vô nghĩa. Cám ơn Dave Orton và Kurt Akeley với sự trợ giúp mức thi hành. Cám ơn Kay Maitz và Renate Kempf đối với trợ giúp tác phẩm tài liệu. Và cám ơn Cindy Ahuna, luôn để mắt đến thức ăn miễn phí. Đặc biệt cám ơn nhà phê bình, người tình nguyện và đọc kĩ càng suốt 600 trang tài liệu kĩ thuật mà là phiên bản lần thứ hai: Bill Armstrong của Evans & Sutherland, Patrick Brown của IBM, Jim Cobb của Parametric Technology, Mark Kilgard của Silicon Graphics, Dale Kirkland của Intergraph, and Andy Vesper của Digital Equipment. Tính siêng năng cần cù của họ tạo một cải tiến lớn chất lượng của cuốn sách này. Cám ơn Mike Heck của Template Graphics Software, Gilman Wong của Microsoft, và Suzy Deffeyes của

IBM với sự đóng góp của họ cho thông tin kĩ thuật trong phụ lục C Thành công liên tiếp của OpenGL nợ nhiều sự tận tâm của nhà kiến trúc OpenGL tham gia Review Board (ARB). Họ hướng dẫn sự mở rộng của OpenGl chuẩn và cập nhật đặc tả để phản xạ cần đến và mong muốn của công nghiệp đồ họa. Những người cộng tác tích cực của OpenGL ARB include Fred Fisher của AccelGraphics; Bill Clifford, Dick Coulter, và Andy Vesper của Digital Equipment Corporation; Bill Armstrong của Evans & Sutherland; Kevin LeFebvre và Randi Rost của Hewlett-Packard; Pat Brown và Bimal Poddar của IBM; Igor Sinyak của Intel; Dale Kirklvà của Intergraph; Henri Warren của Megatek; Otto Berkes, Drew Bliss, Hock San Lee, và Steve Wright của Microsoft; Ken Garnett của NCD; Jim Cobb của Parametric Technology; Craig Dunwoody, Chris Frazier, và Paula Womack của Silicon Graphics; Tim Misner và Bill Sweeney của Sun Microsystems; Mike Heck của Template Graphics Software; và Andy Bigos, Phil Huxley, và Jeremy Morris của 3Dlabs.

Xuất bản lần thứ hai của cuốn sách này sẽ không thể có được nếu không có xuất bản lần đầu, và cũng không thể có được nếu không có sự sáng tạo của OpenGL. Cám ơn kiến trúc sư đầu ngành của OpenGL: Mark Segal và Kurt Akeley. Sự thừa nhận đặc biệt với những người tiên phong đã góp phần to lớn cho thiết kế ban đầu và chức năng của OpenGL: Allen Akin, David Blythe, Jim Bushnell, Dick Coulter, John Dennis, Raymond Drewry, Fred Fisher, Chris Frazier, Momi Furuya, Bill Glazier, Kipp Hickman, Paul Ho, Rick Hodgson, Simon Hui, Lesley Kalmin, Phil Karlton, On Lee, Randi Rost, Kevin P. Smith, Murali Sundaresan, Pierre Tardif, Linas Vepstas, Chuck Whitmer, Jim Winget, và Wei Yen.

Việc thu thập tập hợp các bản màu không có nghĩa là kì công. Trình tự của các bản dựa trên toàn bộ ảnh (Bản 1 đến Bản 9) được tạo bởi Thad Beier, Seth Katz, và Mason Woo. Bản 10 đến Bản 12 là hình ảnh của các chương trình được tạo bởi Mason. Gavin Bell, Kevin Goldsmith, Linda Roy, và Mark Daly tạo chương trình con ruồi cho Bản 24. Mô hình đối với Bản 25 được tạo bởi Barry Brouillette của Silicon Graphics; Doug Voorhies, cũng của Silicon Graphics, thực hiện một số xử lí ảnh cho ảnh cuối cùng. Bản 26 được tạo bởi John Rohlf và Michael Jones, cả hai đều của Silicon Graphics. Bản 27 được tạo bởi Carl Korobkin của Silicon Graphics. Bản 28 là hình ảnh từ một

chương trình viết bởi Gavin Bell với sự đóng góp của nhóm Open Inventor ở Silicon Graphics - Alain Dumesny, Dave Immel, David Mott, Howard Look, Paul Isaacs, Paul Strauss, và Rikk Carey. Bản 29 và 30 là các hình ảnh từ chương trình mô phỏng ảo được tạo bởi nhóm Silicon Graphics IRIS Performer - Craig Phillips, John Rohlf, Sharon Clay, Jim Helman, và Michael Jones từ một sản phẩm cơ sở dữ liệu cho Silicon Graphics bởi Paradigm Simulation. Bản 31 là một hình ảnh bay trên bầu trời, tiền thân của Performer, cái được tạo bởi John Rohlf, Sharon Clay, và Ben Garlick, tất cả của Silicon Graphics

Một số người khác đóng vai trò đặc biệt quan trọng trong việc viết cuốn sách này. Nếu chúng đã liệt kê các tên khác như là các tác giả trong phần đầu của cuốn sách này, Kurt Akeley và Mark Segal sẽ có ở đây, như một tên danh dự. Họ hỗ trợ định nghĩa cấu trúc và các mục đích của cuốn sách, cung cấp mục chính của tài liệu, xem xét lại nó khi tất cả những người khác đã quá mệt mỏi để làm điều đó, và cung cấp với tất cả sự hài hước và hỗ trợ suốt quá trình xử lí. Kay Maitz cung cấp những tác phẩm vô giá và hỗ trợ thiết kế. Kathy Gochenour rất hào hiệp tạp ra nhiều minh họa cho cuốn sách. Susan Riley sao chép chỉnh sửa bản viết tay, có vẻ chúng là một công việc can đảm.

Và bây giờ, mỗi tác giả có thể dành 15 phút để Andy Warhol nói lời cảm ơn.

Tôi xin cảm ơn người quản lí của tôi tại Silicon Graphics - Dave Larson và Way Ting – và các thành viên trong nhóm của tôi- Patricia Creek, Arthur Evans, Beth Fryer, Jed Hartman, Ken Jones, Robert Reimann, Eve Stratton (aka Margaret-Anne Halse), John Stearns, và Josie Wernecke về sự trợ giúp của họ trong quá trình lâu dài này. Cuối cùng nhưng chắc chắn không kém, tôi muốn cảm ơn đến sự đóng góp cho công trình chuyên sâu và khó giải thích để hiểu: Yvonne Leach, Kathleen Lancaster, Caroline Rose, Cindy Kleinfeld, và my parents, Florence và Ferdinvà Neider. - JLN

Ngoài ra, bố mẹ tôi, Edward và Irene Davis, tôi cũng xin gửi lời cảm ơn đến những người đã dạy tôi hầu hết những gì tôi biết về máy tính và đồ họa máy tính- Doug Engelbart và Jim Clark. - TRD

Tôi cũng xin cảm ơn những người đã và đang là thành viên của Silicon Graphics , những điều kiện tiện nghi và khai sáng đã cung cấp cần thiết cho bài báo của tôi vào cuốn sách này: Gerald Anderson, Wendy Chin, Bert Fornaciari, Bill Glazier, Jill Huchital, Howard Look, Bill Mannel, David Marsland, Dave Orton, Linda Roy, Keith Seto, và Dave Shreiner. Đặc biệt quan tâm đến Karrin Nicol, Leilani Gayles, Kevin Dankwardt, Kiyoshi Hasegawa, và Raj Singh đối với sự dìu dắt của họ trong sự nghiệp của tôi. Tôi cũng bày tỏ lòng biết ơn đến nhóm của tôi ở đội bóng gậy trên băng những lúc giải trí khi bắt đầu viết cuốn sách này. Cuối cùng, tôi xin cảm ơn gia đình tôi, đặc biệt là mẹ tôi, Bo, và sau là bố tôi, Henry- MW.

Chương 1 Giới thiệu OpenGL

Mục đích chương

Sau khi đọc

xong chương này, bạn có thể nắm được:

- Đánh giá một cách tổng quát chính xác OpenGL làm được những gì
- Đồng nhất hoá các mức khác nhau của độ phức tạp kết xuất
- Hiểu được cấu trúc cơ bản của một chương trình OpenGL
- Nhận dạng cú pháp lệnh OpenGL.
- Đồng nhất hoá trình tự hoạt động của quy trình kết xuất OpenGL.
- Nêu tổng quát đồ hoạ hoạt cảnh như thế nào trong một chương trình OpenGL

Chương này giới thiệu OpenGL. Nó có các phần quan trọng sau đây:

- " OpenGL là gì ?" giải thích OpenGL là gì, những gì nó làm được và không làm được và cách làm việc của nó.
- "Một đoạn mã OpenGL" đưa ra một chương trình OpenGL nhỏ và giải thích ngắn gọn về nó. Phần này cũng định nghĩa một chút các từ cơ bản đồ hoạ máy tính.
- "Cú pháp lệnh OpenGL" Giải thích một số quy ước và chú thích được sử dụng trong các lệnh OpenGL.
- "OpenGL là một máy trạng thái" mô tả sử dụng các biến trạng thái trong OpenGL và các lệnh cho truy vấn, trạng thái không thể và có thể.
- "Quy trình kết xuất OpenGL" chỉ ra một trình tự điển hình của hoạt động cho việc xử lí hình học và dữ liệu hình ảnh.
- "Các thư viện liên quan OpenGL" mô tả tập hợp thủ tục liên quan OpenGL, bao gồm một thư viện phụ trợ bằng văn bản cho cuốn sách này để đơn giản hoá các ví dụ chương trình.
- "Hoạt cảnh" Giải thích một cách tổng quát làm thế nào để tạo bức ảnh trên màn hình mà có có thể di chuyển.

OpenGL là gì?

OpenGL là một giao diện phần mềm đến phần cứng đồ hoạ. Giao diện này bao gồm khoảng 150 câu lệnh phân biệt mà bạn sử dụng để chỉ rõ các đối tượng và các hoạt động cần thiết để tạo ứng dụng tương tác ba chiều. OpenGL được thiết kế như một tổ chức hợp lí, giao diện độc lập phần cứng được cài đặt trên nhiều nền phần cứng khác nhau. Để kích hoạt đặc tính này, không có lệnh nào để thực hiện công việc cửa sổ hay chứa đầu vào người dùng được chứa trong OpenGL; thay vào đó, bạn phải làm việc thông qua mọi hệ thống cửa sổ để điều khiển phần cứng cụ thể mà bạn đang sử dụng. Cũng như thế, OpenGL không cung cấp các lệnh mức cao để mô tả các mô hình của các đối tượng ba chiều. Các lệnh đó có thể cho phép bạn chỉ rõ hình dạng khá phức tạp như là xe ô tô, các bộ phận cơ thể, máy bay hay phân tử. Với OpenGL, bạn phải xây dựng mô hình mong muốn của bạn từ một một hợp nhỏ các đối tượng *hình học nguyên thuỷ*- các điểm, đường thẳng, và đa giác.

Một thư viện phức tạp mà cung cấp các đặc điểm này tất nhiên có thể xây dựng trong tầm điều khiển của OpenGL. Thư viện tiện ích OpenGL - OpenGL Utility Library (GLU) cung cấp nhiều đặc điểm mô hình, như là bề mặt bậc hai và các đường cong, mặt cong NURBS. GLU là một thành phần chuẩn của mọi cài đặt OpenGL. Cũng thế, có một mức cao hơn, bộ công cụ hướng đối tượng, Open Inventor, chúng được xây dựng ở chóm OpenGL, và có sẵn một cách riêng biệt cho nhiều cài đặt của OpenGL (xem "Các thư viện liên quan OpenGL" để biết thêm thông tin về Open Inventor)

Bây giờ bạn đã biết những cái mà OpenGL không thể, đây là những gì nó có thể. Xem các bản màu- chúng minh hoạ việc sử dụng điển hình của OpenGL. Chúng chỉ ra cảnh tô trong cuốn sách này, *kết xuất* (cái mà thường được gọi là vẽ) bởi một máy tính sử dụng OpenGL theo nhiều cách liên tục phức tạp hơn. Danh sách sau đây mô tả một cách tổng quát các bức tranh này đã được tạo như thế nào.

"Bản 1" chỉ ra toàn bộ cảnh hiển thị như một mô hình khung dây- đó là, xem tất cả các đối tượng trong cảnh được tạo bởi dây. Mỗi đường thẳng của dây tương ứng với cảnh của một đối tượng nguyên thuỷ (điển hình là một đa giác). Ví dụ, bề mặt của bàn được vẽ từ các đa giác tam giác mà được sắp đặt như lát mỏng của bánh.

Chú ý rằng bạn có thể nhìn phần chia của các đối tượng mà được làm tối đi nếu các đối tượng được đặc hơn khung dây. Ví dụ, bạn có thể nhìn thấy toàn bộ mô hình của các quả đồi bên ngoài cửa sổ mặc dù hầu hết các mô hình này thường ẩn bởi các bức tường của căn phòng. Quả cầu xuất hiện gần như đặc vì nó bao gồm hàng trăm khối màu, và bạn thấy các đường thẳng khung dây cho tất cả các cạnh của mọi khối, thậm chí cả những mặt sau của hình cầu. Cách hình cầu được vẽ cho bạn một ý tưởng về các đối tượng phức tạp có thể được tạo như thế nào bằng việc tập hợp các đối tượng ở mức thấp hơn.

"Bản 2" Chỉ ra một phiên bản *dấu hiệu độ sâu* của cảnh khung dây tương tự. Chú ý rằng các đường thẳng xa mắt hơn là mờ hơn, giống như chúng ở trong thế giới thực, theo cách đó đưa một dấu hiệu độ sâu trực quan. OpenGL sử dụng hiệu ứng không khí (được xem như là sương) để kích hoạt dấu hiệu độ sâu.

"Bản 3" chỉ ra phiên bản *khử răng cưa* của cảnh khung dây. Khử răng cưa là một kĩ thuật để giảm các cạnh zic zắc (thường được gọi là răng cưa) được tạo khi xấp xỉ các cạnh trơn với việc sử dụng *pixels (các điểm ảnh)*- viết tắt của phần tử ảnh (*picture elements*)- chúng bị giới hạn bởi một lưới hình chữ nhật. Các đường zic zắc này hầu hết luôn nhìn thấy bởi các đường thẳng gần như ngang và gần như dọc.

"Bản 4" chỉ ra một tô bóng phẳng (*flat-shaded*), phiên bản không chiếu sáng của cảnh. Các đối tượng trong cảnh được chỉ ra là chất rắn. Chúng xuất hiện "phẳng" trong cảnh với chỉ có một màu được sử dụng để kết xuất cho mỗi đa giác, do đó chúng không xuất hiện một cách liền mạch. Không có hiệu ứng từ bất kì nguồn sáng nào.

"Bản 5" chỉ ra một chiếu sáng, phiên bản tô bóng mịn (*smooth-shaded*) của cảnh. Chú ý rằng cảnh trông thật và ba chiều hơn khi các đối tượng được tô bóng để tương ứng với nguồn sáng trong phòng như thể các đối tượng được gọt mịn.

"Bản 6" thêm chiếu sáng (*shadows*) và kết cấu (*textures*) vào phiên bản trước của cảnh. Chiếu sáng không phải là một đặc điểm xác định rõ ràng của OpenGL (không có lệnh chiếu sáng), nhưng bạn có thể tự tạo chúng bằng việc sử dụng kĩ thuật được mô tả trong Chương 14. *Ánh xạ kết cấu* cho phép bạn áp dụng ảnh hai chiều vào đối tượng ba chiều. Trong cảnh này, bề mặt đỉnh bàn là ví dụ điển hình nhất của ánh xạ kết cấu. Thớ gỗ trên cửa sổ và bề mặt bàn là tất cả kết cấu ánh xạ, cũng như bờ tường và bề mặt đồ chơi (trên mặt bàn).

"Bản 7" chỉ ra một đối tượng *di chuyển tạo vệt mờ (motion-blurred)* trong cảnh. Nhân sư (hay chó, phụ thuộc vào cách nhìn Rorschach của bạn) xuất hiện để bắt giữ chuyển động phía trước, để lại một vệt trong phần di chuyển của nó.

"Bản 8" chỉ ra cảnh vẽ đối với việc phủ cuốn sách từ điểm quan sát khác. Bản này minh hoạ ảnh thực sự là một ảnh chụp của các mô hình đối tượng ba chiều.

"Bản 9" quay lại việc sử dụng sương, chúng đã được hiểu trong "Bản 2," chỉ ra sự có mặt của phần tử khói trong không khí. Chú ý rằng làm thế nào để vẫn là hiệu ứng trong "Bản 2" giờ sẽ tác động mạnh hơn trong "Bản 9."

"Bản 10" chỉ ra *hiệu ứng phạm vi độ sâu*, chúng mô phỏng sự bất lực của thấu kính camera để duy trì tất cả đối tượng trong một cảnh chụp tại tiêu điểm. Camera tập trung một điểm cụ thể trong cảnh. Các đối tượng mà gần hơn hay xa hơn đáng kể điểm đó được mờ đi một chút.

Bản màu cho bạn một ý tưởng về những gì bạn có thể thực hiện được với hệ thống đồ hoạ OpenGL. Danh sách sau đây mô tả ngắn gọn các hoạt động đồ hoạ chủ yếu mà OpenGL thực hiện để kết xuất một ảnh trên màn hình (xem "Quy trình kết xuất OpenGL") để biết thêm chi tiết về trình tự các hoạt động này.

1. Xây dựng hình dạng từ các hình học nguyên thuỷ, do đó việc tạo mô tả toán học của đối tượng (OpenGL coi các điểm, đường thẳng, đa giác, ảnh và bitmap là các nguyên thuỷ)
2. Sắp xếp các đối tượng trong không gian ba chiều và lựa chọn các vị trí hợp lí để quan sát cảnh được tạo.
3. Tính toán màu sắc cho tất cả các đối tượng. Màu có thể được thiết kế rõ ràng bởi ứng dụng, được xác định từ điều kiện ánh sáng chỉ ra, được chứa bởi việc dán một kết cấu trên các đối tượng, hay một số kết hợp của ba hành động này.
4. Chuyển đổi các mô tả toán học của các đối tượng và các thông tin liên quan đến màu của chúng đến điểm ảnh trên màn hình. Xử lí này được gọi là phân hoá điểm.
5. Trong các giai đoạn này, OpenGL có thể thực hiện các thao tác khác, như là khử phần đối tượng mà bị khuất bởi các đối tượng khác. Thêm nữa, sau khi cảnh được phân hoá điểm nhưng trước khi vẽ nó trên màn hình, bạn có thể thực hiện một số thao tác trên dữ liệu ảnh nếu bạn muốn.

Trong một số cài đặt (như với X Window System), OpenGL được thiết kế để làm việc thậm chí nếu máy tính hiển thị đồ hoạ bạn tạo không phải là máy tính mà chạy chương trình đồ hoạ của bạn. Điều này có thể là trường hợp nếu bạn làm việc trên môi trường mạng máy tính nơi mà rất nhiều máy tính được kết nối đến nhau bởi mạng kĩ thuật số. Trong tình huống này, máy tính mà chạy chương trình của bạn và các lệnh vẽ OpenGL được gọi máy trạm và máy tính mà nhận các lệnh đó và thực hiện việc vẽ gọi là máy chủ. Cú pháp để truyền các lệnh OpenGL (gọi là *giao thức*) từ máy trạm đến máy chủ luôn giống nhau, do đó các chương trình OpenGL có thể làm việc thông qua mạng thậm chí nếu máy trạm và máy chủ là các loại máy tính khác nhau. Nếu một chương trình OpenGL không chạy trên mạng, và chỉ có một máy tính, và nó đóng vai trò cả máy trạm và máy chủ.

Một đoạn mã ngắn của OpenGL

Do bạn có thể làm nhiều thứ với hệ thống đồ hoạ OpenGL, một chương trình OpenGL có thể bị rắc rối. Tuy nhiên, cấu trúc cơ bản của một chương trình hữu ích có thể đơn giản: Công việc của nó là khởi tạo các lệnh mà điều khiển OpenGL kết xuất như nào và chỉ ra đối tượng được kết xuất.

Trước khi bạn xem đoạn mã OpenGL, hãy tìm hiểu một số thuật ngữ. *Kết xuất*, từ mà bạn đã từng sử dụng, là quá trình mà máy tính tạo hình ảnh từ các mô hình. Các *mô hình*, hay đối tượng, được xây dựng từ các đối tượng hình học nguyên thuỷ- điểm, đường thẳng, và đa giác- chúng được chỉ ra bằng các đỉnh của chúng.

Hình ảnh kết xuất cuối cùng bao gồm các điểm ảnh được vẽ trên màn hình; một điểm ảnh là phần tử nhỏ nhất nhìn thấy mà phần cứng có thể đặt trên màn hình. Thông tin về các điểm ảnh (ví dụ, chúng được cho màu gì) được tổ chức trong bộ nhớ thành ma trận bít. Một ma trận bít là một vùng bộ nhớ mà giữ một bít thông tin cho tất cả các điểm ảnh trên màn hình; ví dụ, bít có thể chỉ ra một điểm ảnh màu đỏ được cho như thế nào. Ma trận bít tự tổ chức thành *vùng đệm khung*, chúng giữ tất cả các thông tin mà hiển thị đồ hoạ cần để điều khiển màu sắc và cường độ của tất cả các điểm trên màn hình.

Bây giờ hãy quan sát chương trình OpenGL. Ví dụ 1-1 tô một hình chữ nhật trắng trên nền đen, như chỉ ra trong Hình 1-1

Hình 1-1 : Hình chữ nhật trắng trên nền đen

Ví dụ 1-1 : Đoạn mã OpenGL

```
#include <whateverYouNeed.h>
main() {
InitializeAWindowPlease();
glClearColor (0.0, 0.0, 0.0, 0.0);
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();
glFlush();
UpdateTheWindowAndCheckForEvents();
}
```

Dòng đầu tiên của thủ tục **main()** khởi tạo một *cửa sổ* trên màn hình: thủ tục **InitializeAWindowPlease()** có nghĩa như một trình giữ chỗ cho các thủ tục hệ thống cửa sổ đặc thù, nói chung chúng không phải là các lời gọi OpenGl. Hai dòng tiếp theo là các câu lệnh OpenGl mà chúng xoá hết cửa sổ thành màu đen: **glClearColor()** thiết lập những màu gì cửa sổ sẽ bị xoá, và **glClear()** thực sự xoá cửa sổ. Mỗi khi màu xoá được thiết lập, cửa sổ được xoá thành màu đó bất cứ khi nào **glclear()** được gọi. Màu được xoá này có thể thay đổi với lời gọi khác **glclearColor().** Tương tự, lệnh **glColor()** thiết lập những màu sử dụng cho các đối tượng được vẽ. Trong trường hợp này, màu là trắng. Tất cả đối tượng vẽ sau điểm này đều sử dụng màu này cho đến khi có một lời gọi thiết lập màu khác.

Lệnh tiếp theo sử dụng trong chương trình, **glOrtho(),** chỉ ra hệ thống toạ độ OpenGL giả định khi nó vẽ ảnh cuối cùng và ảnh được ánh xạ như nào lên màn hình. Lời gọi tiếp theo, chúng được đặt trong cặp lệnh **begin()** và **end(),** định nghĩa đối tượng được vẽ, trong VD này, một đa giác có 4 đỉnh. Góc của đa giác được định nghĩa bởi các lệnh **glVertex3f()**. Bạn có thể đoán các đối số, chúng là các toạ độ (x, y, z), đa giác là một hình chữ nhật trên mặt phẳng z=0.

Cuối cùng, **glFlush()** đảm bảo rằng các lệnh vẽ được chạy thực sự thay vì được chứa trong một *bộ đệm* bằng việc đợi các lệnh OpenGL thêm vào. Thủ tục giữ chỗ **UpdateTheWindowAndCheckForEvents()** quản lí nội dung của cửa sổ và bắt đầu xử lí sự kiện.

Thực tế mà nói, đoạn mã OpenGL có cấu trúc không tốt lắm. Bạn có thể thắc mắc " điều gì xảy ra nếu tôi có gắng di chuyển hoặc thay đổi kích cỡ cửa sổ hay tôi cần thiết lập hệ toạ độ mỗi lần tôi vẽ hình chữ nhật". Chương tiếp theo, bạn sẽ thay thế cả hai lệnh **InitializeAWindowPlease()** và **Update The WindowvàCheck For Events()** bằng việc thực tế nhưng sẽ yêu cầu xây dựng lại mã để cho hiệu quả hơn.

Cú pháp lệnh OpenGL

Như bạn có thể được quan sát phương trình đơn giản của phần trước, lệnh OpenGL sử dụng tiền tố **gl** và viết hoa kí tự bắt đầu cho mỗi từ để tạo tên lệnh (ví dụ, nhớ lại **glClearColor()**). Tương tự, OpenGL định nghĩa hằng số bắt đầu với GL_, sử dụng tất cả các kí tự viết hoa, và sử dụng dấu gạch dưới cho các từ riêng biệt (như GL_COLOR_BUFFER_BIT)

Bạn cũng có thể nhận thấy một số kí tự dường như không liên quan đến một số tên lệnh (vd, **3f** trong **glColor3f()** và **glVertex3f()**). Một sự thực rằng phần **Color** của tên lệnh **glColor3f()** là đủ để định nghĩa lệnh để thiết lập màu hiện tại. Tuy nhiên, một lệnh nhiều hơn như thế được định nghĩa sao cho bạn có thể sử dụng các kiểu khác của đối số. Cụ thể, hậu tố chỉ số **3** chỉ ra rằng ba đối số được cho; các phiên bản khác của lệnh **Color** tạo bốn đối số. Hậu tố **f** chỉ ra rằng đối số là một số thực dấu chấm động. Có các định dạng khác nhau cho phép OpenGL chấp nhận dữ liệu người dùng theo định dạng dữ liệu riêng của họ.

Một số lệnh OpenGL chấp nhận 8 kiểu dữ liệu khác nhau cho đối số của họ. Kí tự được sử dụng như một hậu tố chỉ ra kiểu dữ liệu này cho cài đặt ISO C của OpenGL được chỉ ra trong Bảng 1-1, cùng với định nghĩa kiểu OpenGL tương ứng. Cài đặt cụ thể của OpenGL mà bạn đang sử dụng có thể không đi theo lược đồ một cách chính xác; một cài đặt trong C++ hay Ada, ví dụ, sẽ không cần đến.

Bảng 1-1 : Các hậu tố của lệnh và các kiểu dữ liệu đối số

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Do đó, hai lệnh

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

là tương đương, ngoại trừ lệnh đầu chỉ ra toạ độ của đỉnh là số nguyên 32 bít và lệnh thứ hai chỉ ra chúng là số thực dấu chấm động với độ chính xác đơn.

Chú ý: Cài đặt của OpenGL có sự mềm dẻo trong việc lựa chọn kiểu dữ liệu C nào để biểu diễn các kiểu dữ liệu OpenGL. Nếu bạn kiên quyết sử dụng các kiểu dữ liệu định nghĩa trong OpenGL trong toàn bộ ứng dụng của bạn, bạn sẽ phải tránh các kiểu không khớp khi chuyển đổi mã của bạn giữa các cài đặt khác nhau.

Một số các lệnh OpenGL có thể có một kí tự cuối cùng **v**, chúng chỉ ra rằng lệnh có một con trỏ đến một vectơ (hay mảng) của các giá trị thay vì một chuỗi các đối số riêng lẻ. Nhiều lệnh có cả hai phiên bản vectơ và không vectơ, nhưng một số lệnh chỉ chấp nhận các đối số riêng lẻ và những cái khác yêu cầu ít nhất một vài đối số được chỉ rõ như một vectơ. Các dòng sau đây chỉ ra cách bạn có thể sử dụng một phiên bản vectơ và không vectơ như thế nào để thiết lập màu hiện tại:

```
glColor3f(1.0, 0.0, 0.0);
GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Cuối cùng, OpenGL định nghĩa typedef Glvoid. Đây là một sử dụng thường xuyên nhất cho các lệnh OpenGL mà chấp nhận trỏ đến các mảng giá trị.

Phần còn lại của cuốn sách này (trừ các đoạn mã thực tế), các lệnh OpenGL chỉ được tham chiếu các tên cơ sở của chúng, và một dấu hoa thị kèm theo để chỉ ra ở đây có thể thêm vào tên lệnh. Ví dụ, **glColor*()** được hiểu tất cả những kiểu lệnh khác nhau mà bạn sử dụng để thiết lập màu hiện tại. Nếu bạn muốn tạo một điểm cụ thể về một phiên bản của một lệnh cụ thể, chúng ta gộp hậu tố cần thiết để định nghĩa phiên bản đó. Ví dụ, **glVertex*v()** ám chỉ đến tất cả các phiên bản vectơ của lệnh bạn sử dụng cho các đỉnh cụ thể.

OpenGL là một máy trạng thái

OpenGL là một máy trạng thái. Bạn đặt nó thành các trạng thái khác nhau (hay chế độ) mà sau đó duy trì kết quả cho đến khi bạn thay đổi chúng. Như bạn đã được thấy, màu hiện tại là một biến trạng thái. Bạn có thể thiết lập màu hiện tại là trắng, đỏ hay một màu khác bất kì, và sau đó tất cả các đối tượng được vẽ với màu đó cho đến khi bạn thiết lập màu hiện tại là một màu khác. Màu hiện tại là màu duy nhất của nhiều biến trạng thái mà OpenGL duy trì. Biến khác điều khiển những thứ như là biến đổi quan sát và phép chiếu hiện tại, mẫu đường thẳng và đa giác chấm, chế độ vẽ đa giác, quy ước đóng gói điểm ảnh, vị trí và đặc tính ánh sáng, và thuộc tính chất liệu của đối tượng được vẽ. Nhiều biến trạng thái được xem như các chế độ được kích hoạt hoặc bị hủy với lệnh **glEnable()** hay **glDisable()**.

Mỗi biến trạng thái hay chế độ có một giá trị ngầm định, và tại mọi điểm bạn có thể truy vấn hệ thống một giá trị hiện tại của biến.

Điển hình, bạn sử dụng một trong sáu lệnh sau đây để thực hiện điều này : **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, hay **glIsEnabled()**. Lệnh lựa chọn phụ thuộc vào việc bạn muốn đưa ra câu trả lời có kiểu dữ liệu nào. Một số biến trạng thái có một lệnh truy vấn cụ thể hơn (vd, **glGetLight*()**, **glGetError()**, hay **glGetPolygonStipple()**). Thêm nữa, bạn có thể lưu lại một tập hợp các biến trạng thái trên một ngăn xếp thuộc tính với **glPushAttrib()** hay **glPushClientAttrib()**, tạm thời sửa đổi chúng, và sau đó khôi phục giá trị với **glPopAttrib()** hoặc **glPopClientAttrib()**. Đối với các trạng thái thay đổi tạm thời, bạn nên sử dụng các lệnh này thay vì bất kì lệnh truy vấn nào, vì chúng sẽ hiệu quả hơn.

Xem Phụ luc B để hoàn thiện danh sách các biến trạng thái bạn có thể truy vấn. Đối với mỗi biến, mục lục cũng liệt kê một lệnh đề nghị **glGet*()** mà trả lại giá trị của biến, lớp thuộc tính mà nó thuộc về, và giá trị ngầm định của biến.

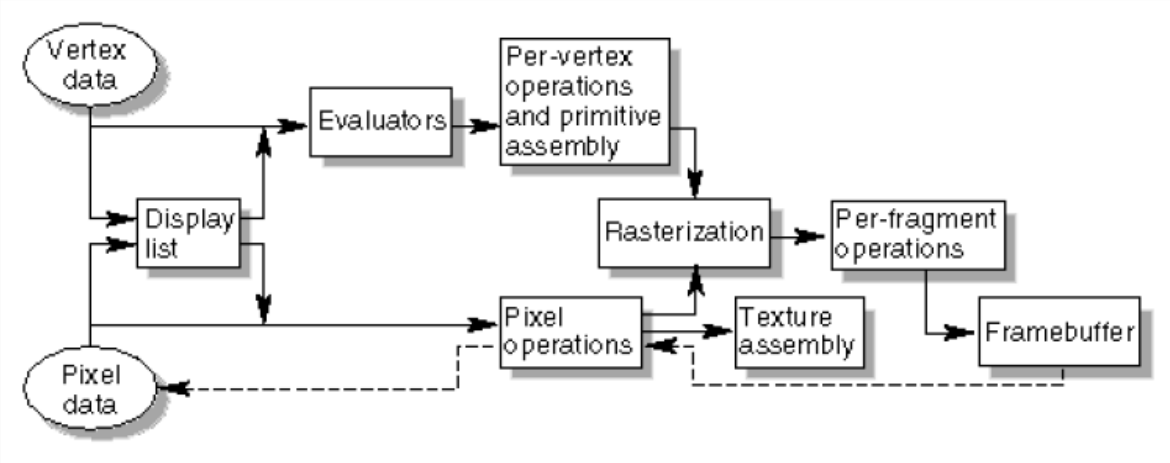
Quy trình kết xuất OpenGL

Hầu hết cài đặt của OpenGL có một trình tự hoạt động giống nhau, một loạt các xử lí giai đoạn được gọi là quy trình kết xuất OpenGL. Trình tự này, được chỉ ra trong Hình 1-2, không phải là quy luật bắt buộc về OpenGL được cài đặt như thế nào nhưng cung cấp một chỉ dẫn tin cậy để dự báo OpenGL sẽ làm gì.

Nếu bạn mới biết về đồ hoạ ba chiều, mô tả tới đây có thể giống như nước uống bên ngoài một vòi cứu hoả. Bạn có thể lướt qua điều này bây giờ, nhưng sẽ phải quay lại Hình 1-2 khi bạn đi qua mỗi chương trong cuốn sách này.

Lược đồ dưới đây chỉ ra phương pháp tiếp cận luồng lắp ráp Henry Ford, với OpenGL dùng đến xử lí dữ liệu. Dữ liệu hình học (đỉnh, đường thẳng và đa giác) chỉ ra đường dẫn thông qua dãy các hộp

mà chúng bao gồm các đánh giá và các phép toán trên đỉnh, trong khi dữ liệu điểm ảnh (điểm ảnh, ảnh và bitmap) được xét khác nhau cho các phần xử lí. Cả hai kiểu dữ liệu phải trải qua các bước cuối cùng như nhau (phân hoá điểm và các phép toán trên đoạn) trước khi dữ liệu điểm ảnh cuối cùng được ghi vào bộ đệm khung.



Hình 1-2 : Trình tự các hoạt động.

Bây giờ bạn sẽ xem chi tiết hơn về giai đoạn chính trong quy trình kết xuất OpenGL.

Danh sách hiển thị

Tất cả dữ liệu, cho dù nó mô tả hình học hay các điểm ảnh, có thể được lưu trong *danh sách hiển thị* để sử dụng hiện tại hoặc sau đó.(Luân phiên để duy trì dữ liệu trong danh sách hiển thị đang được xử lí dữ liệu ngay lập tức- cũng được biết đến như *chế độ tức thì*). Khi danh sách hiển thị được tiến hành, dữ liệu duy trì được gửi từ danh sách hiển thị cùng lúc nếu nó đã gửi bởi ứng dụng trong chế độ tức thì. (Xem Chương 7 để biết thêm thông tin về danh sách hiển thị)

Đánh giá

Tất cả đối tượng hình học nguyên thuỷ được mô tả cuối cùng bởi các đỉnh. Đường cong và mặt cong tham số có thể được mô tả ban đầu bởi các điểm điều khiển và các hàm đa thức được gọi là các hàm cơ bản. Việc đánh giá cung cấp một phương pháp để thu được các đỉnh sử dụng để biểu diễn bề mặt từ các điểm điều khiển. Phương pháp là một ánh xạ đa thức, chúng có thể tạo ra pháp tuyến bề mặt, toạ độ kết cấu, màu và giá trị toạ độ không gian từ các điểm điều khiển (xem Chương 12 có thêm kiến thức về đánh giá)

Các phép toán trên đỉnh

Đối với dữ liệu đỉnh, tiếp theo là giai đoạn "phép toán trên đỉnh", chúng chuyển đổi đỉnh thành nguyên thuỷ. Một số dữ liệu đỉnh (ví dụ, các toạ độ không gian) được biến đổi thành các ma trận dấu chấm động cỡ 4x4. Toạ độ không gian được chiếu từ một vị trí trong thế giới thực 3D đến một vị trí trên màn hình của bạn. (Xem Chương 3 để biết chi tiết về biến đổi các ma trận)

Nếu đặc điểm nâng cao được kích hoạt, giai đoạn này còn bận rộn hơn. Nếu kết cấu được sử dụng, toạ độ kết cấu có thể được tạo và biến đổi ở đây. Nếu ánh sáng được kích hoạt, tính toán ánh sáng được thực hiện với việc sử dụng biến đổi đỉnh, pháp tuyến bề mặt, vị trí nguồn sáng, thuộc tính vật chất và thông tin ánh sáng khác để tạo một giá trị màu.

Tập hợp nguyên thuỷ

Cắt xén, một phần chủ yếu của tổ chức nguyên thuỷ, là việc khử các phần hình học mà chúng giảm bớt một nửa không gian, được định nghĩa bởi một mặt phẳng. Cắt xén điểm đơn giản chỉ là đi qua hay loại bỏ đỉnh; cắt xén đường thẳng hay đa giác có thể thêm các đỉnh phụ thuộc vào việc đường và đa giác được xén như thế nào.

Trong một số trường hợp, điều này được sinh ra bởi phép chia phối cảnh, chúng làm cho đối tượng hình học ở xa sẽ nhỏ hơn những đối tượng ở gần. Sau đó thao tác cổng quan sát và độ sâu (toạ độ z) được áp dụng. Nếu chọn lọc được kích hoạt và nguyên thuỷ là một đa giác, nó sẽ có thể loại bởi một

kiểm tra chọn lọc. Phụ thuộc vào chế độ đa giác, một đa giác có thể được vẽ như các điểm hoặc các đường thẳng (xem "Chi tiết đa giác" trong Chương 2)

Kết quả của giai đoạn này là các hình học nguyên thuỷ đã hoàn thiện, chúng là các đỉnh được biến đổi và cắt xén với màu liên quan, độ sâu, và đôi khi các giá trị toạ độ kết cấu và nguyên tắc để bước phân hoá điểm.

Phép toán điểm ảnh

Trong khi dữ liệu hình học lấy một đường dẫn thông qua quy trình kết xuất OpenGL, dữ liệu điểm ảnh đi theo một đường khác. Điểm ảnh từ một mảng trong hệ thống bộ nhớ lúc đầu được bung ra từ một đa dạng của cú pháp thành một số hợp lí các thành phần. Tiếp theo dữ liệu được gọi, xiên, và xử lí theo ánh xạ điểm ảnh. Kết quả được kẹp lại và sau đó hoặc ghi vào bộ nhớ kết cấu hoặc gửi đến bước phân hoá điểm (xem "Quy trình thu ảnh" trong Chương 8).

Nếu dữ liệu điểm ảnh được đọc từ vùng đệm khung, phép toán chuyển đổi điểm ảnh (co dãn, xiên, ánh xạ và kẹp) được thực hiện. Sau đó các kết quả này được đóng gói thành cú pháp thích hợp và trả về một mảng trong bộ nhớ hệ thống.

Có các phép toán sao chép điểm ảnh đặc biệt để sao chép dữ liệu trong vùng đệm khung đến các phần khác của vùng đệm khung hay bộ nhớ kết cấu. Một truyền đơn được tạo thông qua các phép toán dịch điểm ảnh trước khi dữ liệu được ghi vào bộ nhớ kết cấu hay quay về vùng đệm khung.

Tập hợp kết cấu

Một ứng dụng OpenGL có thể mong muốn áp dụng hình ảnh kết cấu lên các đối tượng hình học để tạo cho chúng trông thực hơn. Nếu một số hình ảnh kết cấu được sử dụng, là khôn ngoan nếu đặt chúng thành các đối tượng kết cấu sao cho bạn có thể dễ sàng chuyển đổi giữa chúng. Một số cài đặt OpenGL có thể có tài nguyên đặc biệt để làm nhanh hiệu suất kết cấu. Đây có thể là đặc biệt, bộ nhớ kết cấu hiệu suất cao. Nếu bộ nhớ này có sẵn, đối tượng kết cấu có thể được ưu tiên để điều khiển việc sử dụng của hạn chế này và giá trị tài nguyên (xem Chương 9)

Phân hoá điểm

Phân hoá điểm là quy ước của cả hai hình học và dữ liệu điểm ảnh thành *đoạn*. Mỗi hình vuông đoạn tương ứng với một điểm ảnh trong vùng đệm khung, đường thẳng và đa giác chấm, độ rộng đường thẳng, kích cỡ điểm, mô hình tô bóng, và tính toán bao phủ để hỗ trợ việc khử răng cưa đều được xem xét như các đỉnh được kết nối thành các đường thẳng hay các điểm ảnh bên trong được tính toán đối với một đa giác kín. Các giá trị màu và độ sâu được thiết kế cho mỗi hình vuông đoạn.

Các phép toán trên đoạn

Trước khi giá trị được lưu trữ thực sự thành vùng đệm khung, một loạt các phép toán được thực hiện mà có thể thay đổi hay thậm chí loại bỏ đoạn. Tất cả các phép toán này có thể được kích hoạt hoặc bị hủy.

Phép toán đầu tiên, chúng có thể được bắt gặp là kết cấu, với một texel (phần tử kết cấu) được tạo từ bộ nhớ kết cấu cho mỗi đoạn và áp dụng đến đoạn. Sau đó tính toán mờ có thể được áp dụng, tiếp theo là thử nghiệm cắt kéo, thử nghiệm alpha, thử nghiệm mẫu tô và thử nghiệm vùng đệm chiều sâu (vùng đệm chiều sâu là đối với loại bỏ mặt khuất). Không một thử nghiệm kích hoạt có thể kết thúc quá trình liên tục của một hình vuông đoạn. Sau đó, việc trộn, phối màu, phép toán logic và tạo mặt nạ bởi mặt bạ bit có thể được thực hiện (xem Chương 6 và Chương 10). Cuối cùng, đoạn được xử lí hoàn toàn được vẽ thành vùng đệm thích hợp, nơi nó có cải tiến cuối cùng thành một điểm và kích hoạt vị trí còn lại cuối cùng của nó.

Thư viện liên quan OpenGL

OpenGL là một công cụ mạnh nhưng tập hợp các lệnh vẽ đều là các đối tượng hình học cơ bản, và tất cả bức vẽ ở mức cao phải thực hiện dưới dạng các lệnh này. Bởi vậy, các chương trình OpenGL phải sử dụng cơ chế nền tảng của hệ thống windows. Một số các thư viện tồn tại cho phép bạn đơn giản các công việc lập trình của bạn, bao gồm các thư viện sau:

- OpenGL Utility Library (GLU) bao gồm một số thủ tục sử dụng các lệnh OpenGL mức thấp để thực hiện các công việc như là thiết lập các ma trận hướng quan sát cụ thể và phép chiếu,

thực hiện khảm đa giác và kết xuất bề mặt. Thư viện này được cung cấp là một phần của tất cả cài đặt OpenGL. Các phần của GLU được mô tả trong *OpenGL Reference Manual*. Các thủ tục GLU hữu ích hơn được mô tả trong tài liệu này, nơi chúng liên quan đến những chủ đề đang được thảo luận, như là trong tất cả Chương 11 và trong phần "giao diện GLU NURBS" trong Chương 12. Các thủ tục GLU sử dụng tiền tố **glu.**

- Đối với mọi hệ thống window, có một thư viện mở rộng các chức năng của hệ thống cửa sổ để hỗ trợ kết xuất OpenGL. Đối với các máy sử dụng hệ thống Window X, OpenGL Extension đến hệ thống Window X (GLX) được cung cấp như là một hỗ trợ cho OpenGL. Thủ tục GLX sử dụng tiền tố **glX**. Đối với Microsoft Windows, các thủ tục WGL cung cấp Windows giao diện OpenGL. Tất cả các thủ tục WGL sử dụng tiền tố **wgl**. Đối với IBM OS/2, PGL là Presentation Manager cho giao diện OpenGL, và thủ tục của nó sử dụng tiền tố **pgl**.
  Tất cả những hệ thống thư viện mở rộng cửa sổ được mô tả chi tiết hơn trong Phụ lục C. Ngoài ra, thủ tục GLX cũng được mô tả trong *OpenGL Reference Manual*
- OpenGL Utility Toolkit (GLUT) là bộ công cụ độc lập hệ thống cửa sổ, được viết bởi Mark Kilgard, để giấu sự phức tạp của các hệ thống cửa sổ API khác nhau. GLUT là chủ đề của phần tiếp theo, và mô tả chi tiết hơn trong sách của Mark Kilgard *Programming for the X Window System* (ISBN 0-201-48359-9). Thủ tục GLUT sử dụng tiền tố **glut.** " Làm thế nào để có mã mẫu" **trong lời giới thiệu** mô tả làm thế nào để có được mã nguồn cho GLUT, với việc sử dụng ftp.
- Open Inventor là một bộ công cụ hướng đối tượng dựa trên OpenGL, chúng cung cấp các đối tượng và các phương pháp để tạo các ứng dụng đồ họa ba chiều. Open Inventor, chúng được viết trong C++, cung cấp các đối tượng trước khi xây dựng và một sự kiện mô hình liền kề cho tương tác người dùng, các thành phần ứng dụng mức cao cho việc tạo và sửa đổi cảnh ba chiều, và khả năng để in các đối tượng và trao đổi dữ liệu trong định dạng đồ họa khác. Open Inventor là tách biệt với OpenGL.

Include Files

Đối với tất cả ứng dụng OpenGL, bạn muốn khai báo các file gl.h trong mọi tệp tin. Hầu hết tất cả ứng dụng OpenGL sử dụng GLU (OpenGL Utility Library) có nghĩa là thư viện tiện OpenGL, chúng yêu cầu các file tiêu đề glu.h. Do đó hầu hết tất cả các file nguồn OpenGL bắt đầu bởi

```
#include <GL/gl.h>
#include <GL/glu.h>
```

Nếu bạn truy cập trực tiếp một thư viện giao diện cửa sổ để hỗ trợ OpenGL như là GLX, AGL, PGL, hay WGL, bạn phải thêm include tệp tin tiêu đề. Ví dụ, nếu bạn gọi GLX, bạn có thể cần thêm những dòng lệnh này đến mã lệnh của bạn

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

Nếu bạn sử dụng GLUT để quản lý các công việc trên cửa sổ của bạn, bạn cần include

```
#include <GL/glut.h>
```

Chú ý rằng glut.h tự động bao gồm gl.h, glu.h, và glx.h, do đó khai báo ba tập tin này là thừa. GLUT cho Microsoft Windows chứa các tệp tin tiêu đề thích hợp để truy cập WGL.

GLUT, bộ công cụ tiện ích OpenGL

Như bạn đã biết, OpenGL chứa các lệnh kết xuất nhưng được thiết kế độc lập với mọi hệ thống cửa sổ hay hệ điều hành. Do đó, nó không chứa các lệnh để mở cửa sổ hay đọc sự kiện từ bàn phím hoặc chuột. Tuy nhiên, không thể viết một chương trình đồ họa hoàn chỉnh mà không phải mở ít nhất một cửa sổ, và hầu hết các chương trình hay đều yêu cầu một bít cho đầu vào người dùng hay dịch vụ khác từ hệ điều hành hay hệ thống cửa sổ. Trong nhiều trường hợp, các chương trình hoàn chỉnh tạo ra các ví dụ thú vị nhất, nên cuốn sách này sử dụng GLUT để đơn giản hóa việc mở cửa sổ, dò đầu vào và tiếp tục. Nếu bạn có một cài đặt OpenGL và GLUT trên hệ thống của bạn, các ví dụ trong cuốn sách này sẽ chạy mà không cần thay đổi khi được liên kết với chúng.

Ngoài ra, do các lệnh vẽ OpenGL chỉ giới hạn trong việc tạo các hình học nguyên thủy đơn giản (điểm, đường thẳng, đa giác), GLUT chứa một số thủ tục tạo các đối tượng ba chiều phức tạp hơn như là hình cầu, đế hoa và ấm trà. Với cách này, việc hiển thị đầu ra chương trình có thể thú vị. (Chú ý rằng thư viện tiện ích OpenGL, GLU, cũng có các thủ tục bậc hai mà tạo một số đối tượng ba chiều giống như GLUT, ví dụ hình cầu, hình trụ, hay hình nón). GLUT có thể không thỏa mãn cho ứng dụng đầy đủ của OpenGL, nhưng bạn có thể thấy hữu ích khi bắt đầu học OpenGL. Phần còn lại của mục này mô tả ngắn gọn một tập nhỏ của các thủ tục GLUT sao cho bạn có thể theo dõi các chương trình ví dụ trong phần còn lại của cuốn sách này (xem Mục D để biết chi tiết hơn về tập con của GLUT, hay xem Chương 4 và 5 của *OpenGL Programming for the X Window System* để có thêm thông tin về những phần còn lại của GLUT).

Quản lí Window

Năm thủ tục thực hiện các công việc cần thiết để khởi tạo cửa sổ

glutInit(int *argc, char **argv) khởi tạo GLUT và xử lí các đối số dòng lệnh bất kì (đối với X, điều này có thể lựa chọn giống hiển thị và hình học). **glutInit()** có thể được gọi trước mọi thủ tục GLUT khác.

glutInitDisplayMode(unsigned int mode) chỉ ra là sử dụng mô hình màu RGBA hay chỉ số màu. Bạn có thể chỉ ra là bạn muốn cửa sổ vùng đệm single hay double (Nếu bạn đang làm việc trong chế độ chỉ số màu, bạn sẽ muốn tải màu nào đó vào ánh xạ màu; sử dụng **glutSetColor()** để thực hiện điều này.) Cuối cùng, bạn có thể sử dụng thủ tục này để chỉ ra rằng bạn muốn cửa sổ có một độ sâu liên kết, mẫu tô, và/hoặc vùng đệm tích lũy. Ví dụ, nếu bạn muốn một cửa sổ với vùng đệm double, mô hình màu RGBA và vùng đệm độ sâu, bạn có thể gọi **glutInitDisplayMode** (GLUT_DOUBLE| GLUT_RGB| GLUT_DEPTH).

- **glutInitWindowPosition** (int x, int y) chỉ ra vị trí màn hình ở góc trên bên trái cửa sổ của bạn.
- **glutInitWindowSize**(int width, int size) chỉ ra kích cỡ, tính theo điểm ảnh, cửa sổ của bạn.
- int **glutCreateWindow**(char *string) tạo một cửa sổ với một ngữ cảnh OpenGL. Nó trả về một định danh duy nhất cho một cửa sổ mới. Cảnh báo: cửa sổ sẽ không được hiển thị cho đến khi **glutMainLoop()** được gọi (xem phần tiếp theo).

Display Callback

glutDisplayFunc(void (*func)(void)) bạn sẽ thấy đây là hàm gọi lai sự kiện đầu tiên và quan trọng nhất. Bất cứ lúc nào GLUT xác định nội dung cửa sổ cần để hiển thị lại, hàm gọi lại được đăng kí bởi **glutDisplayFunc()** được chạy. Do đó, bạn nên đặt tất cả các thủ tục bạn cần để vẽ lại cảnh trong hàm gọi lại việc hiển thị.

Nếu chương trình của bạn thay đổi nội dung cửa sổ, đôi khi bạn sẽ phải gọi **glutPostRedisplay**(void), chúng gửi cho **glutMainLoop()** một lời gọi lại hiển thị đăng kí ở cơ hội tiếp theo của nó.

Chạy chương trình

Vấn đề cuối cùng bạn phải làm là gọi **glutMainLoop**(void). Tất cả các cửa sổ đã tạo bây giờ sẽ hiển thị, và việc kết xuất các cửa sổ đó sẽ có hiệu quả. Xử lí sự kiện bắt đầu, lời gọi lại hiển thị đã đăng kí được kích hoạt. Khi đi vào vòng lặp này, nó sẽ không bao giờ được thoát ra.

Ví dụ 1-2 chỉ ra bạn có thể sử dụng GLUT để tạo một chương trình đơn giản trong Ví dụ 1-1. Chú ý việc xây dựng lại mã. Để đạt hiệu quả tối đa, các thao tác mà chỉ cần gọi một lần (thiết lập màu nền và hệ thống toạ độ) nằm trong một thủ tục gọi là **init()**. Các thao tác kết xuất (và có thể tô lại) cảnh nằm trong thủ tục **display()**, chúng là lời gọi hiển thị GLUT đã đăng kí.

Ví dụ 1-2 : Một chương trình đơn giản sử dụng GLUT: hello.c

```
#include <GL/gl.h>
#include <GL/glut.h>
void display(void)
```

```
{
/* clear all pixels */
glClear (GL_COLOR_BUFFER_BIT);
/* vẽ đa giác trắng (hình chữ nhật)với góc tại
* (0.25, 0.25, 0.0)và(0.75, 0.75, 0.0)
*/
glColor3f (1.0, 1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();
/* don't wait!
* start processing buffered OpenGL routines
*/
glFlush ();
}
void init (void)
{
/* select clearing (background) color */
glClearColor (0.0, 0.0, 0.0, 0.0);
/* initialize viewing values */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
/*
* Khai báo kích cỡ cửa sốt, vị trí, và chế độ hiển thị
* (Vùng đệm đơn và RGBA). Mở cửa sổ với thanh tiêu đề "hello"
*. Gọi các thủ tục khởi tạo.
* đăng kí hàm callback để hiển thị đồ hoạ.
* chạy vòng lặp chính và xử lí các sự kiện.
*/
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition (100, 100);
glutCreateWindow ("hello");
init ();
glutDisplayFunc(display);
glutMainLoop();
```

```
return 0; /* ISO C requires main to return int. */
}
```

Điều khiển sự kiện đầu vào

Bạn có thể sử dụng các thủ tục này để đăng kí các lệnh gọi lại mà chúng được gọi khi các sự kiện cụ thể xảy ra.

glutReshapeFunc(void (*func)(int w, int h)) chỉ ra những hành động cần làm khi thay đổi kích cỡ cửa sổ.

glutKeyboardFunc(void (*func)(unsigned char key, int x, int y)) và **glutMouseFunc**(void (*func)(int button, int state, int x, int y)) cho phép bạn liên kết một phím của bàn phím hay một nút chuột với một thủ tục mà chúng được gọi khi một phím hay nút chuột được nhấn hoặc giải phóng.

glutMotionFunc(void (*func)(int x, int y)) đăng kí một thủ tục để gọi lại khi chuột di chuyển trong khi nút chuột đang được nhấn.

Quản lí một xử lí nền

Bạn có thể chỉ rõ một hàm được thực hiện nếu không có sự kiện khác đang chờ- ví dụ, khi một sự kiện vòng lặp không dùng đến-với **glutIdleFunc**(void (*func)(void)). Thủ tục này có một con trỏ đến hàm như một đối số duy nhất của nó. Pass là NULL (0) để vô hiệu hoá việc chạy của hàm.

Vẽ các đối tượng ba chiều

GLUT bao gồm một số thủ tục để vẽ các đối tượng ba chiều

cone	icosahedron	teapot
cube	octahedron	tetrahedron
dodecahedron	sphere	torus

Bạn có thể vẽ các đối tượng này như các khung dây hay các đối tượng có bề mặt đặc với pháp tuyến bề mặt định nghĩa. Ví dụ, thủ tục cho một hình hộp lập phương hay một hình cầu như sau:

void **glutWireCube**(GLdouble size);

void **glutSolidCube**(GLdouble size);

void **glutWireSphere**(GLdouble radius, GLint slices, GLint stacks);

void **glutSolidSphere**(GLdouble radius, GLint slices, GLint stacks);

Tất cả các mô hình này được vẽ với tâm ở gốc hệ toạ độ thực. (xem thông tin trên nguyên mẫu của tất cả các thủ tục vẽ này).

Hoạt cảnh

Một trong những thú vị nhất bạn có thể làm trên đồ họa máy tính là vẽ các bức tranh chuyển động. cho dù bạn là một kĩ sư cố gắng nhìn mọi phía các bộ phận máy móc bạn đang thiết kế hay không, một phi công học lái máy bay sử dụng một mô phỏng, hay đơn thuần một người đam mê trò chơi máy tính, rõ ràng hoạt cảnh là một phần quan trọng của đồ họa máy tính. Trong rạp chiếu phim, chuyển động được kích hoạt bằng việc tạo một chuỗi các bức ảnh và chiếu chúng với tốc độ 24 hình trong một giây trên màn hình. Mỗi hình được xê dịch vào vị trí sau thấu kính, màn chắn được mở ra, và hình được hiển thị. Màn chắn được đóng ngay lập tức trong khi phim được chuyển đến hình tiếp theo, sau đó hình đó được hiển thị, và cứ như thế. Mặc dù bạn đang xem 24 hình khác nhau trong mỗi giây, não của bạn trộn tất cả chúng thành một hoạt cảnh liên tục. (Phim của Charlie Chaplin ngắn chỉ 16 hình trên giây làm cho hình ảnh giật). Thực tế, các máy chiếu hiện đại hiển thị mỗi bức tranh hai lần ở tốc độ 48/ giây để giảm rung hình. Màn hình đồ họa máy tính làm tươi điển hình (vẽ lại bức tranh) xấp xỉ 60 đến 76 lần trên giây, và một số thậm chí chạt với tốc độ làm tươi 120/giây. Rõ ràng, 60 trên giây là mịn hơn 30, và 120 là tốt hơn 60. Tốc độ làm tươi nhanh hơn 120, tuy nhiên tránh giảm trở lại, do mắt người chỉ tốt đến thế.

Lí do chính mà phép chiếu hình ảnh chuyển động là mỗi hình được hoàn thiện khi nó được hiển thị. Giả sử bạn cố gắng thực hiện hoạt cảnh máy tính của phim triệu hình của bạn với một chương trình như thế này:

```
open_window();
for (i = 0; i < 1000000; i++) {
clear_the_window();
draw_frame(i);
wait_until_a_24th_of_a_second_is_over();
}
```

Nếu bạn thêm thời gian cần cho hệ thống của bạn để xóa màn hình và vẽ một hình đặc thù, chương trình này càng cho kết quả phức tạp hơn phụ thuộc vào nó việc xóa và vẽ. Giả sử việc vẽ xấp xỉ gần 1/24 giây. Phần vẽ đầu tiên được thấy với đầy đủ 1/24 giây và biểu diễn một ảnh đặc trên màn hình; phần vẽ giáp cuối được xóa ngay lập tức khi chương trình bắt đầu với hình tiếp theo. Chúng biểu diễn nhiều nhất một ảnh biến hóa, do hầu hết trong 1/24 giây mắt bạn đang quan sát một hình nền đã xóa thay cho những phần mà không đủ may mắn để vẽ cuối cùng. Vấn đề là chương trình này không hiển thị hoàn toán các hình được vẽ; thay vào đó, bạn quan sát việc vẽ như nó xảy ra.

Hầu hết các cài đặt OpenGL cung cấp vùng đệm kép- phần cứng hay phần mềm mà cung cấp hai vùng đệm màu hoàn chỉnh. Một cái được hiển thị trong khi cái kia đang được vẽ. Khi việc vẽ một khung được hoàn thành, hai vùng đệm được đổi cho nhau, sao cho bây giờ sẽ sử dụng cái mà đã được hiển thị để vẽ và ngược lại. Điều này giống như một máy chiếu phim với việc chỉ có hai khung trong một vòng lặp; trong khi một cái đang được chiếu trên màn hình, một họa sĩ đang cố xóa và vẽ lại khung đang không được nhìn thấy. Miễn là họa sĩ đủ nhanh, người quan sát thông báo không có sự sai khác giữa cài đặt này với việc tất cả các khung đã được vẽ và máy chiếu đơn giản đang hiển thị chúng từng cái. Với vùng đệm kép, tất cả các khung được chỉ ra duy nhất khi việc vẽ được hoàn thành; người quan sát không bao giờ thấy khung chưa được vẽ hoàn chỉnh.

Một phiên bản sửa đổi của chương trình có trước mà hiển thị hoạt cảnh đồ họa mịn có thể thấy như sau:

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++) {
clear_the_window();
draw_frame(i);
swap_the_buffers();
}
```

Làm tươi việc dừng

Đối với một số cài đặt OpenGL, ngoài việc đơn giản đổi chỗ các vùng đệm được quan sát và các vùng đệm được vẽ, thủ tục **swap_the_buffers()** đợi cho đến khi hết chu kì làm tươi màn hình hiện tại sao cho vùng đệm trước được hiển thị hoàn toàn. Thủ tục này cũng cho phép vùng đệm mới được hiển thị hoàn chỉnh, bắt đầu từ đầu. Giả sử rằng hệ thống của bạn có tốc độ làm tươi màn hình là 60 lần trên giây, điều này có nghĩa là tốc độ khung nhanh nhất bạn có thể kích hoạ là 60 khung trên giây (ftp), và nếu tất cả các khung của bạn được xóa và vẽ dưới 1/60 giây, hoạt cảnh của bạn sẽ chạy trơn với tốc độ đó.

Những gì thường xảy ra trên một hệ thống này là khung quá phức tạp để vẽ trong 1/60 giây, sao cho mỗi khung được hiển hơn một lần. Nếu, ví dụ, nó mất 1/45 giây để vẽ một khung, bạn có tốc độ 30ftp, và đồ hoạ của của bạn đứng im 1/30-1/45=1/90 giây cho mỗi khung, hoặc 1/3 thời gian.

Thêm nữa, tốc độ làm tươi hình ảnh là hằng số, chúng có thể có một số kết quả thực hiện không mong muốn, ví dụ, với 1/60 giây để làm tươi màn hình và một tốc độ khung không đổi, bạn có thể chạy 60 fps, 30 fps, 20 fps, 15 fps, 12 fps và hơn thế (60/1, 60/2, 60/3, 60/4, 60/5, ...). Điều này có nghĩa là nếu bạn viết một ứng dụng và thêm dần các đặc điểm vào (như một mô phỏng chuyến bay, và các cảnh mặt đất được thêm vào), thoạt nhìn mỗi đặc điểm bạn thêm vào không ảnh hưởng đến việc thực hiện tổng thể, bạn vẫn có 60ftp. Sau đó, tất cả đột ngột, bạn thêm một đặc điểm mới, và hệ thống không thể vẽ toàn bộ mọi thứ trong 1/60 của giây, do đó hoạt cảnh sẽ giảm từ 60 fps về 30

fps bởi vì nó mất thời gian đổi chỗ vùng đệm có thể lúc đầu. Một điều tương tự xảy ra khi thời gian vẽ một khung là lớn hơn 1/30 giây- hoạt cảnh giảm từ 30 xuống 20fps.

Nếu độ phức tạp của cảnh giống như thời gian ảo thuật (1/60 giây, 2/60 giây, 3/60 giây, và hơn thế trong ví dụ này) thì do biến đổi ngẫu nhiên, một số khung quá thời gian một chút và một số sớm hơn một chút. Thì tốc độ của khung là không đều, chúng có thể có nhiễu. Trong trường hợp này, nếu bạn không thể làm đơn giản cảnh sao cho tất cả các khung đủ nhanh, có thể tốt hơn nếu thêm một chủ ý, độ trễ nhỏ để chắc chắn tất cả chúng bị mất, cho một hằng số, chậm hơn, về tốc độ khung. Nếu các khung của bạn có độ sai khác lớn phức tạp, có thể cần đến một tiếp cận phức tạp hơn

Hoạt cảnh = Vẽ lại + đổi chỗ

Cấu trúc của chương trình hoạt cảnh thật không khác quá nhiều trong mô tả này. Thông thường, nó dễ hơn để vẽ lại toàn bộ vùng đệm từ đầu mỗi khung hơn tìm ra các phần yêu cầu vẽ lại. Điều này đặc biệt đúng với ứng dụng như là mô phỏng lái máy bay ba chiều khi hướng máy bay thay đổi một chút vị trí mọi thứ bên ngoài cửa sổ.

Trong hầu hết hoạt cảnh, đối tượng trong cảnh được vẽ lại đơn giản với biến đổi khác- điểm nhìn của người quan sát di chuyển, hay ô tô di chuyển xuống đường một chút, hay một đối tượng được quay một chút. Nếu việc tính lại quan trọng được yêu cầu cho thao tác không phải bản vẽ, tốc độ khung hình thường giảm xuống. Hãy nhớ, tuy nhiên, thời gian chờ sau thủ tục **swap_the_buffers()** thường được sử dụng cho những tính toán đó.

OpenGL không có lệnh **swap_the_buffers()** vì đặc điểm không có sẵn trên tất cả phần cứng và trong mọi trường hợp, nó phụ thuộc nhiều vào hệ thống cửa sổ. Ví dụ, nếu bạn sử dụng X Window System và truy cập nó trực tiếp, bạn có thể sử dụng thủ tục GLX dưới đây:

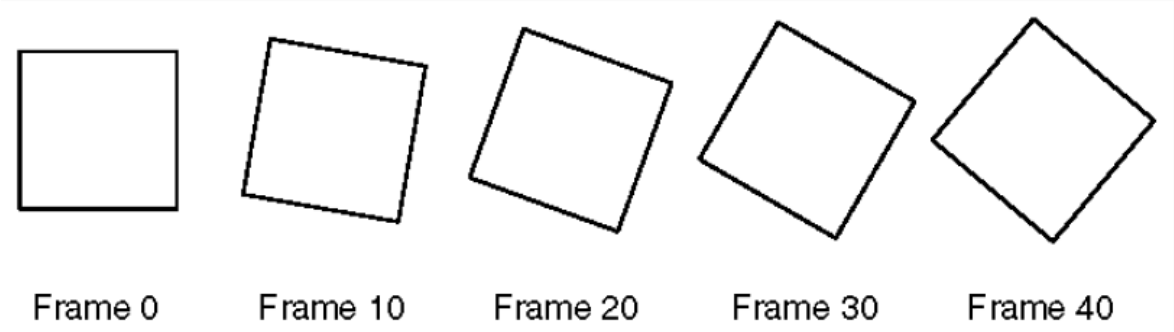
void glXSwapBuffers(Display *dpy, Window window);

(Xem phụ lục C về thủ tục tương đương cho các hệ cửa sổ.)

Nếu bạn đang sử dụng thư viện GLUT, bạn sẽ muốn gọi thủ tục này:

void glutSwapBuffers(void);

Ví dụ 1-3 minh hoạ sử dụng **glutSwapBuffers()** trong một ví dụ vẽ hình vuông quay như Hình 1-3. Ví dụ sau đây cũng chỉ ra sử dụng GLUT như thế nào để điều khiển một thiết bị đầu vào và bật /tắt một hàm chờ. Trong ví dụ này, nhấp thay đổi nút chuột để tắt/bật việc quay.



Hình1-3 : Hình vuông quay vùng đệm kép.

Ví dụ 1-3 : Chương trình Double-Buffered: double.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
```

```
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glPushMatrix();
glRotatef(spin, 0.0, 0.0, 1.0);
glColor3f(1.0, 1.0, 1.0);
glRectf(-25.0, -25.0, 25.0, 25.0);
glPopMatrix();
glutSwapBuffers();
}
void spinDisplay(void)
{
spin = spin + 2.0;
if (spin > 360.0)
spin = spin - 360.0;
glutPostRedisplay();
}
void reshape(int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(spinDisplay);
break;
case GLUT_MIDDLE_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(NULL);
break;
default:
break;
}
}
/*
```

```
* Request double buffer display mode.
* Register mouse input callback functions
*/
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (250, 250);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0;
}
```

Chương 2: Quản lí trạng thái và vẽ các đối tượng hình học trong OpenGL

Mục đích chương

Sau khi đọc chương này, bạn có thể làm được:

- Xoá cửa sổ với màu tuỳ ý
- Bắt bản vẽ bất kì đang chờ được hoàn thiện
- Vẽ với hình học nguyên thuỷ bất kì – điểm, đường thẳng, và đa giác trong không gian hai chiều và ba chiều
- Bật/ tắt trạng thái và truy vấn các biến trạng thái
- Điều khiển màn hình với các nguyên thủy của chúng - ví dụ, vẽ đường thẳng đứt nét hay phác thảo đa giác.
- Chỉ rõ vectơ pháp tuyến tại điểm thích hợp trên bề mặt của đối tượng rắn.
- Sử dụng *vertex arrays* để chứa và truy cập nhiều dữ liệu hình học chỉ với một số lời gọi hàm.
- Lưu và khôi phục một số biến trạng thái cùng một lúc.

Mặc dù bạn có thể vẽ những bức tranh phức tạp và thú vị bằng cách sử dụng OpenGL, tất cả đều được xây dựng từ một số ít các hình họa cơ bản. Điều này không nên quá ngạc nhiên - nhìn vào những gì Leonardo da Vinci hoàn thành chỉ với bút chì và cọ vẽ.

Ở cấp độ cao nhất của trừu tượng, có ba kiểu vẽ cơ bản: Xóa cửa sổ, vẽ một đối tượng hình học, và vẽ một đối tượng raster. Đối tượng raster, trong đó bao gồm những thứ như hình ảnh 2D, bitmap, và phông chữ ký tự, được đề cập trong *Chương 8*. Trong chương này, bạn tìm hiểu làm thế nào để xóa màn hình và để vẽ các đối tượng hình học, bao gồm cả điểm, đường thẳng, và đa giác phẳng.

Bạn có thể tự hỏi, *"Khoan đã. Tôi đã thấy rất nhiều đồ họa máy tính trong các bộ phim và trên truyền hình, có rất nhiều đường và bề mặt cong bóng đẹp mắt. Làm thế nào để vẽ chúng, nếu tất cả OpenGL chỉ có thể vẽ những đường thẳng và đa giác phẳng?"* Ngay cả những hình ảnh trên trang bìa của cuốn sách này bao gồm một bàn tròn và các đối tượng trên bàn có bề mặt cong. Nó chỉ ra rằng tất cả các đường và bề mặt cong bạn đã nhìn thấy được tính xấp xỉ bởi số lượng lớn các đa giác phẳng nhỏ hoặc đường thẳng, theo cách tương tự địa cầu trên trang bìa được xây dựng từ một tập lớn các khối hình chữ nhật. Quả địa cầu không xuất hiện là một bề mặt nhẵn vì các khối là tương đối lớn so với quả địa cầu. Ở cuối chương này, chúng tôi chỉ cho bạn làm thế nào để xây dựng các đường và bề mặt cong từ rất nhiều hình học nguyên thủy nhỏ.

Chương này có các phần chính sau đây:

"Bộ công cụ vẽ thiết yếu" giải thích làm thế nào để xóa cửa sổ và khối lượng bản vẽ sẽ được hoàn thành. Nó cũng cung cấp cho bạn thông tin cơ bản về kiểm soát màu của các đối tượng hình học và mô tả một hệ thống tọa độ.

"Mô tả các điểm, đường thẳng và đã giác " cho bạn thấy tập các đối tượng hình học nguyên thủy là gì và làm thế nào để vẽ chúng.

"Quản lí lệnh cơ bản" mô tả làm thế nào để bật và tắt một số trạng thái (chế độ) và truy vấn các biến trạng thái.

"Hiển thi các điểm, đường thẳng và đa giác " giải thích bạn có những điều khiển gì trên các chi tiết về việc đối tượng nguyên thủy được vẽ như thế nào- *Ví dụ:* điểm có đường kính như nào, đường thẳng là nét liền hay đứt, và đa giác là đường viền hay phủ kín.

"Các vectơ pháp tuyến" trình bày làm thế nào để xác định vectơ pháp tuyến cho các đối tượng hình học và (một cách ngắn gọn) những vectơ này để làm gì.

"Mảng đỉnh " cho bạn thấy làm thế nào để đưa nhiều dữ liệu hình học vào một vài mảng và làm thế nào, chỉ với một vài lời gọi hàm, để tô vẽ hình học nó mô tả. Việc làm giảm lời gọi hàm có thể làm tăng hiệu quả và hiệu suất của việc tô vẽ.

"Các nhóm thuộc tính" cho thấy làm thế nào để truy vấn các giá trị hiện tại của biến trạng thái và làm thế nào để lưu và khôi phục một số trạng thái liên quan đến nhiều giá trị cùng một lúc.

"Một số gợi ý cho việc xây dựng các mô hình đa giác của bề mặt tìm hiểu các vấn đề và kỹ thuật liên quan đến việc xây dựng xấp xỉ đa giác trên các bề mặt.

Một điều cần lưu ý khi bạn đọc phần còn lại của chương này là với OpenGL, trừ khi bạn chỉ định khác, mỗi khi bạn ra lệnh vẽ, các đối tượng xác định được vẽ. Điều này có vẻ hiển nhiên, nhưng trong một số hệ thống, trước tiên bạn phải tạo một danh sách những thứ để vẽ. Khi danh sách của bạn được hoàn tất, bạn nói với các phần cứng đồ họa vẽ các mục trong danh sách. Kiểu đầu tiên được gọi đồ họa *chế độ tức thì (immediate-mode)* và là kiểu OpenGL mặc định. Ngoài việc sử dụng chế độ tức thì, bạn có thể chọn để lưu một số lệnh trong một danh sách (được gọi là một danh sách hiển thị) cho việc vẽ sau này. Chế độ đồ họa tức thì điển hình dễ dàng cho chương trình, nhưng danh sách hiển thị thường hiệu quả hơn. Chương 7 cho bạn biết cách sử dụng danh sách hiển thị và tại sao bạn có thể muốn sử dụng chúng.

Dụng cụ vẽ thiết yếu

Phần này giải thích làm thế nào để xoá cửa sổ chuẩn bị cho việc vẽ, thiết lập màu của các đối tượng được vẽ, và bắt việc vẽ được hoàn thiện. Những chủ đề này không phải làm việc trực tiếp với các đối tượng hình học, nhưng mọi chương trình mà vẽ đối tượng phải giải quyết các vấn đề này.

Xoá của sổ

Việc vẽ trên màn hình máy tính khác với vẽ trên giấy là giấy bắt đầu từ tờ giấy trắng, và tất cả bạn phải làm là vẽ một bức tranh. Trên máy tính, bộ nhớ lưu giữ tranh luôn được phủ kín với bức tranh cuối bạn vẽ, do đó bạn cần phải xóa nó bởi một số màu nền trước khi bạn bắt đầu vẽ một cảnh mới. Màu bạn sử dụng cho nền phụ thuộc vào ứng dụng. Với một xử lí từ, bạn có thể xóa thành trắng (màu của giấy) trước khi bạn bắt đầu vẽ chữ. Nếu bạn đang vẽ một khung cảnh một không gian vũ trụ, bạn xóa thành màu đen của không gian trước khi bắt đầu vẽ ngôi sao, hành tinh, và người ngoài hành tin. Đôi khi bạn có thể không cần đến xóa màn hình chút nào; ví dụ, nếu ảnh là bên trong một phòng, toàn bộ cửa sổ đồ họa bao phủ như bạn vẽ tất cả trên các bức tường.

Ở điểm này, bạn có thể băn khoăn tại sao chúng ta cứ nói mãi về *xóa* cửa sổ- tại sao không vẽ một hình chữ nhật với màu phù hợp mà chúng phủ kín toàn bộ cửa sổ? đầu tiên, một lệnh đặc biệt để xóa cửa sổ có thể hiệu quả hơn nhiều một lệnh vẽ đa năng. Thêm nữa, như bạn sẽ thấy trong Chương 3, OpenGL cho phép bạn thiết lập hệ thống tọa độ, quan sát vị trí, và hướng quan sát tùy ý, sao cho nó có thể khó để tính toán một kích cỡ và vị trí phù hợp cho hình chữ nhật xóa cửa sổ. Cuối cùng, trên nhiều máy, phần cứng đồ họa bao gồm đa vùng đệm thêm vùng đệm chứa màu của các điểm ảnh mà được hiển thị. Các vùng đệm khác này thi thoảng phải được xóa, và việc có một lệnh đơn mà có thể xóa mọi kết hợp của chúng là thuận tiện (xem Chương 10 cho một trình bày về tất cả các vùng đệm có thể).

Bạn cũng phải biết màu của các điểm ảnh được lưu trữ như thế nào trong phần cứng đồ họa được gọi là *mặt phẳng bít* (*bitplanes*). Có hai phương pháp lưu trữ. Hoặc là các giá trị đỏ, xanh lục, xanh dương và alpha (RGBA) cho một điểm ảnh có thể được lưu trữ trữ trực tiếp trong mặt phẳng bít, hay một giá trị chỉ số đơn mà tham chiếu đến bảng tra màu (color lookup table). Chế độ hiển thị màu RGBA được sử dụng phổ biến hơn, do đó trong hầu hết các ví dụ của cuốn sách này là sử dụng nó (xem Chương 4 để có thêm thông tin chi tiết về hai chế độ hiển thị này). Bạn có thể lờ đi tất cả các tham chiếu giá trị alpha cho đến Chương 6.

Ví dụ, các dòng mã xoá một cửa sổ chế độ RGBA thành màu đen

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Dòng đầu tiên thiết lập màu xóa là màu đen, và lệnh tiếp theo xóa toàn bộ cửa sổ là màu xóa hiện tại. Biến đơn **glClear()** chỉ ra vùng đệm nào được xóa. Trong trường hợp này, chương trình xóa chỉ là vùng đệm màu, nơi ảnh hiển thị trên màn hình được duy trì. Điển hình, bạn thiết lập màu xóa một lần, phần đầu trong ứng dụng của bạn, và sau đó bạn xóa các vùng đệm mỗi lần thấy cần thiết. OpenGL duy trì rãnh của màu xóa hiện tại như một biến trạng thái hơn là việc yêu cầu bạn chỉ rõ nó mỗi lần một vùng đệm được xóa.

Chương 4 và Chương 10 cho bạn biết các vùng đệm khác được sử dụng như thế nào. Bây giờ, tất cả những gì bạn cần biết là xoá chúng rất đơn giản, Ví dụ, để xoá cả hai bộ đệm màu và bộ đệm độ sâu, bạn sẽ sử dụng trình tự các lệnh sau đây

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Trong trường hợp này, lời gọi **glClearColor()** giống như lúc trước, lệnh **glClearDepth()** chỉ ra giá trị mà mọi điểm ảnh của vùng đệm độ sâu được thiết lập, và biến cho lệnh **glClear()** giờ sẽ bao gồm OR từng bít của tất cả các vùng đệm bị xoá. Tóm tắt dưới đây của **glClear()** bao gồm một bảng mà liệt kê các vùng đệm mà có thể bị xóa, tên của chúng, và ở chương có mỗi kiểu vùng đệm được trình bày

*void **glClearColor**(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);*

Thiết lập màu xoá hiện tại để sử dụng trong vùng đệm màu được xoá với chế độ RGBA (Xem chương 4 để có thêm thông tin trên chế độ RGBA). Các giá trị red, green, blue,và alpha được co lại nếu cần thiết trong miền [0,1]. Màu xoá ngầm định là (0, 0, 0, 0), nó có nghĩa là màu đen.

*void **glClear**(GLbitfield mask);*

Xoá vùng đệm chỉ rõ thành các giá trị xoá hiện tại của chúng. Đối số mặt nạ là kết hợp bitwise-Ored của các giá trị liệt kê trong Bảng 2-1.

Bảng 2-1 : Vùng đệm được xoá

Buffer	Name	Reference
Color buffer	GL_COLOR_BUFFER_BIT	Chapter 4
Depth buffer	GL_DEPTH_BUFFER_BIT	Chapter 10
Accumulation buffer	GL_ACCUM_BUFFER_BIT	Chapter 10
Stencil buffer	GL_STENCIL_BUFFER_BIT	Chapter 10

Trước khi đưa ra một lệnh để xóa đa vùng đệm, bạn phải thiết lập các giá trị đến mỗi vùng đệm là được xóa nếu bạn muốn một số cái gì đó khác so với các giá trị ngầm định của màu RGBA , giá trị độ sâu, màu tổng hợp và chỉ số mẫu tô. Thêm các lệnh **glClearColor()** và **glClearDepth()** để thiết lập các giá trị hiện tại để xóa vùng đệm màu và độ sâu, **glClearIndex()**, **glClearAccum()**, và **glClearStencil()** chỉ ra *chỉ số màu*, màu tổng hợp, và chỉ số mẫu tô sử dụng để xóa vùng đệm tương ứng. (Xem Chương 4 và Chương 10 để mô tả những vùng đệm này và sử dụng của chúng).
OpenGL cho phép bạn chỉ ra nhiều vùng đệm vì việc xóa nói chung là một thao tác chậm, do mọi điểm ảnh trong cửa sổ (có thể hàng triệu) được xét đến, và một số phần cứng đồ họa cho phép một tập các vùng đệm được xóa đồng thời. Phần cứng không hỗ trợ xóa đồng thời có thể thực hiện theo trình tự. Sự sai khác giữa

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

và

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

đó là mặc dù cả hai có kết quả cuối cùng như nhau, ví dụ đầu tiên có thể chạy nhanh hơn trên nhiều máy. Và chắc chắn nó sẽ không chạy chậm hơn.

Chỉ định một màu

Với OpenGL, mô tả hình dạng của một đối tượng đang được vẽ là độc lập với mô tả màu của nó. Khi một đối tượng hình học cụ thể được vẽ, nó được vẽ với việc sử dụng lược đồ màu chỉ định hiện

tại. Lược đồ màu có thể đơn giản như "vẽ mọi thứ màu đỏ trong xe chữa cháy" hay có thể phức tạp như "giả sử đối tượng được làm bằng nhựa màu xanh dương, nhưng có đèn pha màu vàng chiếu vào từ một hướng và những hướng còn lại có ánh sáng màu nâu đỏ hoạt động mức thấp ". Nói chung, một người lập trình OpenGL đầu tiên thiết lập màu hay lược đồ màu và sau đó vẽ đối tượng. Cho đến khi màu hay lược đồ màu bị thay đổi, tất cả các đối tượng được vẽ với màu đó hay sử dụng lược đồ màu đó. Phương pháp này hỗ trợ OpenGL kích hoạt hiệu suất vẽ cao hơn so với kết quả khi nó không duy trì màu hiện tại.

Ví dụ, một đoạn mã

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

Vẽ đối tượng A và B màu đỏ và đối tượng C màu xanh dương. Lệnh ở dòng thứ tư được thiết lập màu hiện tại là màu xanh lá cây bị yếu dần.

Màu sắc, ánh sáng và tô bóng là những chủ đề lớn với toàn bộ chương hay những phần lớn dành cho chúng. Để vẽ hình học nguyên thủy mà có thể thấy được, tuy nhiên, bạn cần một số kiến thức cơ bản làm thế nào thiết lập màu hiện tại; thông tin này được cung cấp trong đoạn tiếp theo, (Xem Chương 4 và Chương 5 để biết chi tiết về chủ đề này).

Để thiết lập một màu, sử dụng lệnh **glColor3f().** Nó có ba biến, tất cả chúng là các số dấu chấm động trong khoảng 0.0 đến 1.0. Các biến là, theo thứ tự, các thành phần đỏ, xanh lá cây và xanh dương. Bạn có thể cho rằng ba giá trị này chỉ ra một màu tổng hợp: 0.0 có nghĩa là không sử dụng bất kì thành phần nào trong đó, và 1.0 có nghĩa là sử dụng tất cả các thành phần mà bạn có thể. Do đó, câu lệnh

```
glColor3f(1.0, 0.0, 0.0);
```

tạo màu đỏ chói nhất mà hệ thống có thể vẽ, mà không có các thành phần xanh lá cây và xanh dương. Tất cả giá trị bằng 0 sẽ tạo thành màu đen; ngược lại, tất cả là một thì tạo thành màu trắng. Thiết lập tất cả ba thành phần là 0.5 sẽ cho một màu xám (màu giữa của đen và trắng). Đây là tám lệnh và các màu chúng sẽ thiết lập:

```
glColor3f(0.0, 0.0, 0.0); đen
glColor3f(1.0, 0.0, 0.0); đỏ
glColor3f(0.0, 1.0, 0.0); xanh lá cây
glColor3f(1.0, 1.0, 0.0); vàng
glColor3f(0.0, 0.0, 1.0); xanh dương
glColor3f(1.0, 0.0, 1.0); tía
glColor3f(0.0, 1.0, 1.0); xanh ngọc bích
glColor3f(1.0, 1.0, 1.0); trắng
```

Bạn có thể đã được thấy thủ tục thiết lập màu xóa, **glClearColor()**, có bốn biến, ba biến đầu của chúng phù hợp với các biến cho **glColor3f()**. Biến thứ tư là giá trị alpha; nó được bao trùm toàn bộ trong "Trộn" của Chương 6. Bây giờ, thiết lập biến thứ tự của **glClearColor()** là 0.0, đây là giá trị ngầm định của nó.

Bắt buộc hoàn thành của bản vẽ

Như bạn đã thấy trong "Quy trình kết xuất OpenGL " trong Chương 1, các hệ thống đồ họa hiện đại có thể được coi là một dây truyền lắp ráp. Bộ xử lí trung tâm chính phát ra một lệnh vẽ. Có lẽ phần cứng khác thực hiện biến đổi hình học. Cắt xén được thực hiện, theo sau là tô bóng và/hoặc kết cấu. Cuối cùng, các giá trị được viết thành mặt phẳng bít để hiển thị. Trong kiến trúc mức cao nhất, mỗi hoạt động này được thực hiện bởi một phần khác nhau của phần cứng mà nó đã được thiết kế để thực hiện công việc cụ thể của nó nhanh chóng. Trong một kiến trúc như thế, CPU không cần phải đợi một lệnh vẽ hoàn thiện trước khi phát ra lệnh tiếp theo. Trong khi CPU đang gửi một một đỉnh xuôi theo quy trình, biến đổi phần cứng đang làm việc với đỉnh cuối cùng được gửi, một đỉnh trước

đó đang được cắt xén, và cứ như thế. Trong một hệ thống như thế, nếu CPU đợi mỗi lệnh để hoàn thiện trước khi phát ra lệnh tiếp theo, thì hiệu suất hoạt động rất kém.
Ngoài ra, ứng dụng có thể được chạy trên nhiều máy. Ví dụ, giả sử chương trình chính đang chạy ở một nơi khác (trên máy gọi từ máy khách) và bạn đang quan sát kết quả vẽ trên trạm làm việc hay thiết bị cuối (máy chủ), chúng được kết nối bởi một mạng đến máy khách. Trong trường hợp đó, nó có thể vô cùng kém hiệu quả khi gửi mỗi lệnh trên mạng kế tiếp nhau, do chi phí đáng kể thường kết hợp với mỗi việc truyền trên mạng. Thông thường, máy khách lấy một tập hợp các lệnh thành một gói mạng đơn lẻ trước khi gửi nó. Thật không may, mã mạng lưới trên máy khách thường không có cách nào biết chương trình đồ họa kết thúc bản vẽ một khung hay cảnh. Trong trường hợp tồi nhất, nó đợi mãi mãi các lệnh vẽ thêm vào đủ để đầy một gói, và bạn không bao giờ thấy bản vẽ hoàn chỉnh.
Vì lí do này, OpenGL cung cấp lệnh **glFlush()**, chúng bắt ép các máy khách gửi các gói tin lên mạng kể cả khi nó có thể chưa đầy.
Ở nơi không có mạng và tất cả các lệnh được chạy chính xác ngay lập tức trên máy chủ, **glFlush(),** có thể không hiệu quả. Tuy nhiên, nếu bạn đang viết một chương trình mà bạn muốn làm việc đúng cách với cả hai trường hợp có và không có mạng, sử dụng một lời gọi đến **glFlush()** ở phần cuối mỗi khung hay cảnh.
Chú ý rằng **glFlush()** không đợi để vẽ hoàn thiện, nó chỉ bắt buộc bản vẽ khi bắt đầu chạy, do đó đảm bảo rằng tất cả các lệnh trước đó *chạy* trong thời gian có hạn cho dù là các lệnh kết xuất không được chạy thêm nữa. Có các tình huống khác mà **glFlush()** có ích.

- Phần mềm kết xuất mà xây dựng ảnh trong hệ thống bộ nhớ và không muốn cập nhật liên tục màn hình.
- Việc cài đặt tập hợp thiết lập của các lệnh kết xuất để trả dần chi phí khởi động. Ví dụ mạng truyền nói ở trên là một minh họa cho điều này.

*void **glFlush**(void);*
Ép buộc các lệnh OpenGL phát ra trước để bắt đầu chạy, do đó đảm bảo rằng chúng hoàn thiện trong thời gian có hạn
Một số lệnh- ví dụ, lệnh đổi chỗ các vùng đệm trong chế độ vùng đệm kép tự động tự động phát ra các lệnh chờ trên mạng trước khi chúng có thể xuất hiện.
Nếu **glFlush()** không hiệu quả với bạn, hãy thử **glFinish().** Lệnh này phát trên mạng như **glFlush()** và sau đó đợi thông báo từ phần cứng đồ họa hay mạng chỉ ra rằng bản vẽ hoàn thiện trong vùng đệm khung. Bạn có thể cần sử dụng **glFinish()** nếu bạn muốn các công việc đồng bộ - ví dụ, để chắc chắn kết xuất ba chiều của bạn trên màn hình trước khi bạn sử dụng Display PostScript để vẽ các nhãn trên đỉnh của hiển thị. Ví dụ khác cũng đảm bảo rằng bản vẽ là hoàn thiện trước khi nó bắt đầu chấp nhận đầu vào người dùng. Sau khi bạn phát ra một lệnh **glFinish()**, quá trình đồ họa của bạn bị khóa cho đến khi nó nhận được thông báo từ phần cứng đồ họa rằng bản vẽ được hoàn thiện.
Lưu ý rằng quá lạm dụng **glFinish()** có thể giảm hiệu suất ứng dụng của bạn, đặc biệt nếu bạn đang chạy trên mạng, do nó yêu cầu liên lạc khứ hồi. Nếu **glFlush()** hiệu quả cho nhu cầu của bạn, sử dụng nó thay vì **glFinish()**.
*void **glFinish**(void);*
Bắt ép tất cả các lệnh OpenGL phát ra trước đó phải hoàn thành. Lệnh này không trả về cho đến khi tất cả các lệnh trước đó được thực hiện hoàn toàn.

Hệ tọa độ thiết yếu

Bất cứ khi nào bạn khởi tạo mở một cửa sổ hay sau đó di chuyển và chỉnh kích cỡ cửa sổ đó, hệ thống cửa sổ sẽ gửi một sự kiện để thông báo cho bạn. Nếu bạn đang sử dụng GLUT, thông báo được tự động; mọi thủ tục đã được đăng kí đến **glutReshapeFunc()** sẽ được gọi. Bạn phải đăng kí một hàm callback mà sẽ

- Thiết lập lại vùng hình chữ nhật mà sẽ là bức tranh hiển thị mới.
- Định nghĩa hệ tọa độ đến các đối tượng sẽ được vẽ.

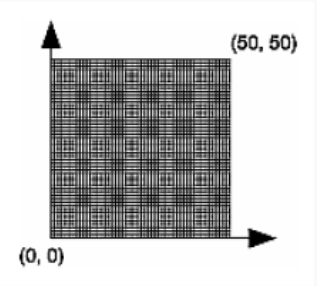
Trong Chương 3 bạn sẽ thấy các hệ tọa độ ba chiều định nghĩa như thế nào, nhưng bây giờ, chỉ tạo một hệ tọa độ hai chiều cơ bản, đơn giản để bạn có thể vẽ một số đối tượng. Gọi **glutReshapeFunc**(**reshape**)**,** với **reshape()** là hàm được chỉ ra sau đây trong Ví dụ 2-1

Ví dụ 2-1 : Hàm Reshape Callback

```
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

Bên trong của GLUT sẽ truyền cho hàm này hai đối số: chiều rộng và chiều cao, tính theo điểm ảnh, với tạo mới, di chuyển và chỉnh kích cỡ cửa sổ. **glViewport()** điều chỉnh hình chữ nhật điểm ảnh để vẽ là toàn bộ cửa sổ mới. Ba thủ tục tiếp theo điều chỉnh hệ tọa độ để vẽ sao cho góc dưới bên trái là (0.0), và góc trên bên phải là (w, h) (Xem Hình 2-1)

Để giải thích nó theo cách khác, hãy tưởng tượng một miếng giấy vẽ. Giá trị của w và h trong **reshape()** đại diện có bao nhiêu cột và hàng tính theo ô vuông trên tờ giấy vẽ của bạn. Sau đó bạn phải đặt các tọa độ trên tờ giấy vẽ. Thủ tục **gluOrtho2D()** đặt gốc tọa độ, (0, 0), ở vị trí ô vuông thấp nhất và trái nhất, và định nghĩa mỗi ô vuông biểu diễn một đơn vị. Bây giờ, khi bạn tô vẽ các điểm, đường thẳng và đa giác trong phần còn lại của chương này, chúng sẽ hiện trên bề mặt giấy này theo số ô vuông đoán được dễ dàng. (Bây giờ, giữ tất cả đối tượng của bạn là hai chiều)



Hình 2-1 : Định nghĩa hệ tọa độ với w = 50, h = 50

Mô tả điểm, đường thẳng và các đa giác

Phần này giải thích làm thế nào để mô tả hình học nguyên thủy OpenGL. Tất cả các hình học nguyên thủy được mô tả dưới dạng tọa độ *đỉnh* của chúng mà định nghĩa các các điểm của chính chúng, các điểm đầu cuối của đoạn thẳng, hay góc của đa giác. Phần tiếp theo trình bày các hình học nguyên thủy này được hiển thị như thế nào và bạn có điều khiển gì qua hiển thị của chúng.

Điểm, đường thẳng và đa giác là gì?

Chắc chắn bạn có một ý tưởng khá hay về những gì một nhà toán học định nghĩa thuật ngữ *điểm*, đường thẳng và đa giác. Nghĩa trong OpenGL tương tự, nhưng không hẳn là giống nhau.

Một sự khác biệt xuất phát từ hạn chế của những tính toán dựa trên máy tính. Trong mọi cài đặt OpenGL, tính toán dấu chấm động là hữu hạn chính xác, và chúng có những lỗi khi làm tròn. Do đó, tọa độ của các điểm, đường thẳng kém chính xác từ những vấn đề tương tự.

Một sự khác nhau quan trọng hơn xuất hiện từ hạn chế của hiển thị đồ họa điểm. Trên một hiển thị như thế, đơn vị hiển thị nhỏ nhất là một điểm ảnh và mặc dù các điểm ảnh có kích thước nhỏ hơn 1/100 độ rộng một inch, chúng vẫn rất lớn so với khái niệm nhỏ vô hạn trong toán học (đối với điểm) hay mỏng (đối với đường thẳng). Khi OpenGL thực hiện tính toán, nó giả sử các điểm được biểu diễn là các vectơ của số dấu chấm động. Tuy nhiên một điểm thường (không phải lúc nào cũng vậy) được vẽ như một điểm ảnh đơn lẻ, và nhiều điểm có các tọa độ sai khác nhỏ có thể được vẽ bởi OpenGL trên cùng một điểm ảnh.

Điểm

Một điểm được biểu diễn bởi một tập hơn các số dấu chấm động được gọi là một đỉnh. Tất cả các tính toán bên trong được thực hiện như thể các đỉnh là ba chiều.

Các đỉnh chỉ rõ bởi người dùng là hai chiều (đó là tọa độ duy nhất x và y) được thiết kế một tọa độ z bằng 0 bởi OpenGL.

Nâng cao

OpenGL làm việc trong hệ tọa độ thuần nhất của phép chiếu hình học ba chiều, do đó đối với tính toán bên trong, tất cả các đỉnh được biểu diễn với tọa độ dấu chấm động (x, y, z, w). Nếu w khác không, những tọa độ này tương ứng với điểm Oclit ba chiều (x/w, y/w, z/w). Bạn có thể chỉ ra tọa độ w trong lệnh OpenGL, nhưng hiếm khi chúng được thực hiện. Nếu tọa độ w không được chỉ rõ, nó được hiểu là 1.0 (Xem Phụ lục F để có thêm thông tin về hệ tọa độ đồng nhất)

Đường thẳng

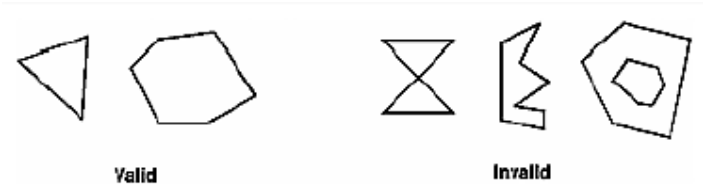
Trong OpenGL, thuật ngữ *đường thẳng* có nghĩa là *đoạn thẳng*, không giống kiểu của toán học là mở rộng vô hạn theo hai hướng. Có nhiều cách dễ dàng để chỉ ra một chuỗi các đoạn thẳng kết nối, hay thậm chí khép kín, một chuỗi các đoạn thẳng kết nối (xem Hình 2-2). Trong nhiều trường hợp, dường như, các đường thẳng tạo thành một chuỗi kết nối được chỉ ra dưới dạng các đỉnh tại điểm đầu cuối của chúng.

Hình 2-2 : Hai chuỗi kết nối của các đoạn thẳng

Đa giác

Đa giác là các vùng khép kín với một vòng khép kín đơn của các đoạn thẳng, ở đây các đoạn thẳng được chỉ rõ bởi các đỉnh với các điểm kết thúc của chúng. Đa giác thường được vẽ với các điểm ảnh phủ kín bên trong, nhưng bạn cũng có thể vẽ chúng như một khung hay một tập hợp các điểm. (Xem "Chi tiết đa giác")

Nói chung, đa giác có thể là phức tạp, nên OpenGL tạo một số hạn chế mạnh với những gì tạo nên một đa giác nguyên thủy. Đầu tiên, cạnh của đa giác OpenGL không thể cắt nhau (toán học gọi đa giác thỏa mãn điều kiện này là *đa giác đơn giản*). Thứ hai, các cạnh của đa giác phải *lồi*, có nghĩa là chúng không có đỉnh lõm. Phát biểu chính xác, một vùng là đỉnh nếu, cho hai điểm bất kì trong vùng đa giác, đoạn thẳng nối chúng cũng phải nằm bên trong. Xem Hình 2-3 để thấy một số đa giác hợp lệ và không hợp lệ. OpenGL, tuy nhiên, không giới hạn số các đoạn thẳng tạo nên đường biên của một đa giác lồi. Chú ý rằng đa giác với các lỗ hổng không thể được mô tả. Chúng là đa giác không lồi, và chúng không thể được vẽ với một đường biên tạo bởi một vòng lặp đơn khép kín. Thấy rằng nếu bạn xét OpenGL với một đa giác phủ kín không lồi, nó có thể không được vẽ như bạn mong đợi. Ví dụ, trong hầu hết các hệ thống không có nhiều hơn so với phần lồi của đa giác sẽ được phủ kín. Trong một số hệ thống, ít hơn so với phần lồi có thể được phủ kín.



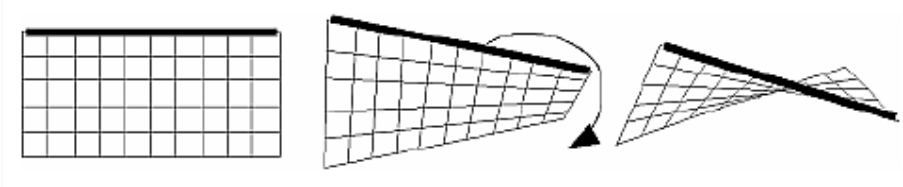
Hình 2-3 : Các đa giác hợp lệ và không hợp lệ

Lí do để OpenGL quy định những kiểu đa giác hợp lệ là để nó đơn giản hơn cho việc cung cấp nhanh phần cứng tô vẽ đa giác đối với những lớp hạn chế của các đa giác. Các đa giác đơn giản có thể được tô vẽ nhanh chóng. Những trường hợp phức tạp hơn thì khó để dò nhanh. Cho nên để hiệu quả nhất, OpenGL xóa những vật hình ngón tay của nó và giả sử rằng các đa giác là đơn giản.

Nhiều bề mặt trong thế giới thực chứa các đa giác không đơn giản, các đa giác không lồi hay các đa giác với lỗ hổng. Do tất cả các đa giác đó có thể được tạo từ tập hợp các đa giác lồi đơn giản. Một số thủ tục để xây dựng các đối tượng phức tạp hơn được cung cấp bởi thư viện GLU. Các thủ tục này lấy những mô tả phức tạp và khảm chúng, hay phân tách chúng thành các nhóm đa giác đơn giản hơn trong OpenGL mà sau đó có thể được tô vẽ.(Xem "Khảm đa giác" trong Chương 11 để biết thêm thông tin về các thủ tục khảm)

Do các đỉnh OpenGL luôn trong ba chiều, các điểm tạo thành ranh giới của một đa giác cụ thể không nhất thiết nằm trên cùng một mặt phẳng trong không gian. (Tất nhiên, chúng thực hiện trong nhiều

trường hợp- nếu tất cả tọa độ z là 0, ví dụ, hay nếu đa giác là một tam giác). Nếu các đỉnh của một đa giác không nằm trên cùng một mặt phẳng, thì sau phép quay khác nhau trong không gian, thay đổi mặt nhìn, và chiếu lên màn hình hiển thị, các điểm có thể không tạo thành một đa giác lồi đơn giản nữa. Ví dụ, tưởng tượng một *tứ giác* bốn điểm với các điểm chệch ngoài mặt phẳng một chút, và thấy chúng gần như dốc lên. Bạn có thể có được một đa giác không đơn giản mà giống như nơ hình con bướm, như chỉ ra trong Hình 2-4, chúng không đảm bảo được tô vẽ một cách chính xác. Tình huống này không phải tất cả là khác thường nếu bạn xấp xỉ bề mặt cong bởi tứ giác tạo thành các điểm nằm trên chính xác bề mặt. Bạn có thể luôn tránh vấn đề bằng cách sử dụng các tam giác, do ba điểm bất kì luôn nằm trên một mặt phẳng.



Hình 2-4 : Đa giác không phẳng biến đổi thành đa giác không đơn giản

Hình chữ nhật

Do hình chữ nhật là rất thông thường trong ứng dụng đồ họa, OpenGL cung cấp một lệnh vẽ nguyên thủy hình chữ nhật phủ kín **glRect*()**. Bạn có thể vẽ một hình chữ nhật như một đa giác, được mô tả trong "vẽ hình học nguyên thủy OpenGL ", nhưng cài đặt cụ thể của bạn trong OpenGL có thể đã tối ưu **glRect*()** cho việc vẽ hình chữ nhật.

*void **glRect**{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);*

*void **glRect**{sifd}**v**(TYPE*v1, TYPE*v2);*

Vẽ hình chữ nhật được định nghĩa bởi các điểm góc (x1, y1) và (x2, y2). Hình chữ nhật nằm trên mặt phẳng z=0 và có các cạnh song song với trục x và trục y. Nếu dạng vectơ của hàm được sử dụng. các góc được cho hai con trỏ đến dãy, mỗi chúng chứa một cặp (x, y).

Chú ý rằng mặc dù hình chữ nhật bắt đầu với một hướng cụ thể trong không gian ba chiều (trong mặt phẳng x-y và song song với các trục), bạn có thể thay đổi điều này bằng việc áp dụng phép quay hay phép biến đổi khác. (Xem Chương 3 để có thông tin về cách thực hiện biến đổi này)

Đường cong và bề mặt cong

Mọi đường cong và bề mặt cong trơn có thể được xấp xỉ- đến mức độ chính xác tùy ý- bởi các đoạn thẳng ngắn hay vùng đa giác nhỏ.

Do đó, việc phân chia đủ nhỏ các đường và bề mặt cong và sau đó xấp xỉ chúng với các đoạn thẳng liên tục hay các đa giác phẳng để chúng có hình dáng cong (xem Hình 2-5). Nếu bạn nghi ngờ điều này, tưởng tượng việc phân chia cho đến khi mỗi đoạn thẳng hay đa giác là nhỏ đến nỗi nó nhỏ hơn một điểm ảnh trên màn hình.

Hình 2-5 : Các đường cong xấp xỉ

Cho dù các đường cong không phải là các hình học nguyên thủy, OpenGL cung cấp một số hỗ trợ trực tiếp cho việc phân chia và vẽ chúng. (Xem Chương 12 để có thông tin về cách vẽ đường cong và bề mặt cong)

Xác định các đỉnh

Với OpenGL, tất cả các đối tượng hình học được mô tả cơ bản là một tập hợp các đỉnh có thứ tự. Bạn sử dụng lệnh **glVertex*()** để xác định một đỉnh.

*void **glVertex**{234}{sifd}[v](TYPEcoords);*

Xác định một đỉnh để sử dụng mô tả một đối tượng hình học. Bạn có thể cung cấp đến bốn tọa độ (x, y, z, w) đối với một đỉnh cụ thể hay ít hơn như là hai (x, y) bằng việc lựa chọn phiên bản thích hợp

*của lệnh. Nếu bạn sử dụng một phiên bản mà không xác định rõ z hay w, z được hiểu là 0 và w được hiểu là 1. Lời gọi **glVertex*()** chỉ có tác dụng giữa cặp **glBegin()** và **glEnd().***

Ví dụ 2-2 cung cấp một số ví dụ về việc sử dụng **glVertex*()**.

Ví dụ 2-2 : Sử dụng hợp lệ của glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

Ví dụ đầu biểu diễn một đỉnh với tọa độ ba chiều (2, 3, 0).

(Lưu ý rằng nếu nó không được xác định, tọa độ z được hiểu là 0) Tọa độ trong ví dụ thứ hai là (0.0, 0.0, 3.1415926535898) (các số dấu chấm động chính xác double). Ví dụ thứ ba biểu diễn đỉnh với tọa độ ba chiều (1.15, 0.5, -1.1). (Lưu ý rằng tọa độ x, y, và z được chia bởi tọa độ w). Trong ví dụ cuối cùng, dvect là một con trỏ đến mảng của ba số dấu chấm động có độ chính xác double.

Trên một số máy, dạng vectơ của **glVertex*()** hiệu quả hơn, do chỉ cần một biến đơn được truyền đến hệ thống phụ đồ họa.

Phần cứng đặc biệt có thể gửi toàn bộ một loạt các tọa độ trong một gói đơn lẻ. Nếu máy của bạn có đặc điểm này, nó cải tiến miền dữ liệu của bạn sao cho các tọa độ đỉnh được đóng gói trình tự trong bộ nhớ. Trong trường hợp này, có thể hiệu quả bằng việc việc sử dụng thao tác mảng đỉnh của OpenGL (Xem "mảng đỉnh")

Vẽ các đối tượng hình học nguyên thuỷ OpenGL

Bây giờ bạn đã hiểu các đỉnh được xác định như thế nào, bạn vẫn cần biết cách để chỉ cho OpenGL tạo một tập hợp các điểm, đường thẳng và đa giác từ các đỉnh đó. Để làm điều này, bạn gộp mỗi tập hợp các đỉnh giữa lời gọi **glBegin()** và một lời gọi **glEnd()**. Đối số truyền cho **glBegin()** xác định thứ tự của hình học nguyên thuỷ được xây dựng từ các đỉnh. Ví dụ, Ví dụ 2-3 xác định các đỉnh cho đa giác chỉ trong Hình 2-6.

Ví dụ 2-3 : Đa giác được tô màu

```
glBegin(GL_POLYGON);
glVertex2f(0.0, 0.0);
glVertex2f(0.0, 3.0);
glVertex2f(4.0, 3.0);
glVertex2f(6.0, 1.5);
glVertex2f(4.0, 0.0);
glEnd();
```

Hình 2-6 : Vẽ một đa giác hay một tập các điểm

Nếu bạn đã sử dụng GL_POINT thay cho GL_POLYGOL, hình học nguyên thủy sẽ đơn giản là năm điểm chỉ ra trong Hình 2-6. Bảng 2-2 trong hàm tóm tắt sau đây **glBegin()** liệt kê mười đối số có thể và kiểu tương ứng của hình học nguyên thủy

*void **glBegin**(GLenum mode);*

Đánh dấu việc bắt đầu một danh sách dữ liệu đỉnh mà mô tả một hình học nguyên thuỷ. Kiểu của nguyên thuỷ được chỉ ra bởi chế độ, chúng có thể giá trị bất kì được được chỉ ra trong Bảng 2-2.

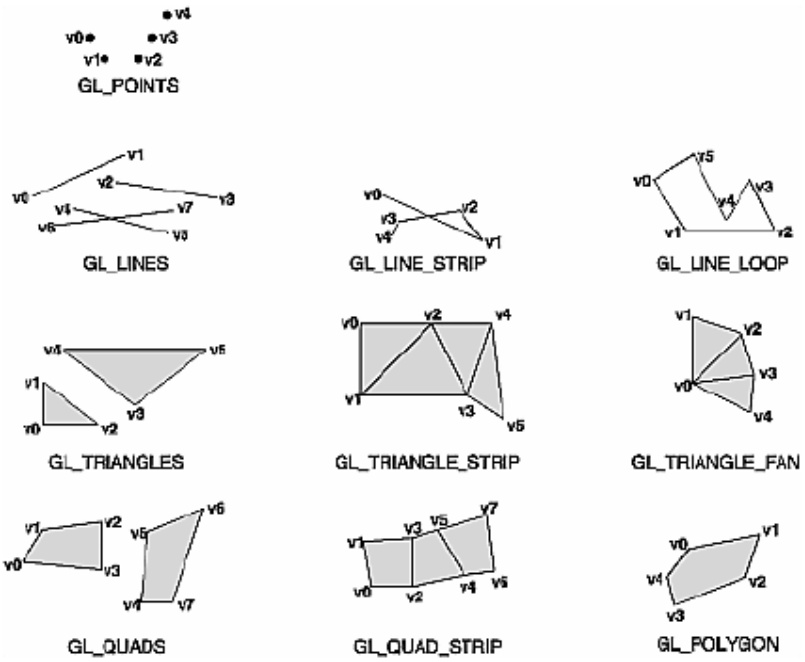
Bảng 2-2 : Tên và ý nghĩa hình học nguyên thủy

Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

void ***glEnd****(void);*

Đánh dấu kết thúc của danh sách dữ liệu đỉnh.

Hình 2-7 chỉ ra các ví dụ của tất cả các hình học nguyên thủy liệt kê trong Bảng 2-2. Các đoạn chỉ ra theo hình mô tả các điểm ảnh mà được vẽ cho mỗi đối tượng. Chú ý rằng ngoài các điểm, một số kiểu của đường thẳng và đa giác được định nghĩa. Tất nhiên, bạn có thể có nhiều cách để vẽ cùng một đối tượng. Phương pháp bạn lựa chọn phụ thuộc vào dữ liệu đỉnh của bạn.



Hình 2-7 : Các kiểu hình học nguyên thủy

Như bạn đã đọc các mô tả sau đây, giả sử rằng *n* đỉnh (v0, v1, v2, ... , vn-1) được mô tả giữa một cặp **glBegin()** và **glEnd().**

GL_POINTS Vẽ một điểm tại một trong *n* đỉnh

GL_LINES Vẽ một chuỗi các đoạn thẳng không kết nối. Các đoạn được vẽ giữa v0 và v1, giữa v2 và v3, và tiếp tục như thế. Nếu *n* là số lẻ, đoạn cuối được vẽ giữa vn-3 và vn-2, và vn-1 bị bỏ qua.
GL_LINE_STRIP Vẽ một đoạn thẳng từ v0 đến v1, sau đó từ v1 đến v2, và tiếp tục như thế, cuối cùng vẽ đoạn từ vn-2 đến vn-1.
Do đó, tất cả có *n*-1 các đoạn thẳng được vẽ. Không có đoạn nào được vẽ trừ khi n lớn hơn 1. Không hạn chế số đỉnh biểu diễn một dải đường thẳng (hay một đoạn thẳng khép kín); các đường thẳng có thể cắt nhau tùy ý.
GL_LINE_LOOP giống như GL_LINE_STRIP, ngoài việc đường thẳng cuối cùng được vẽ từ vn-1 đến v0, tạo thành một vòng khép kín.
GL_TRIANGLES Vẽ một chuỗi các tam giác (đa giác có ba cạnh) với việc sử dụng v0, v1, v2, sau đó đến v3, v4, v5, và tiếp tục như thế. Nếu *n* không chia hết cho 3, một hai đỉnh cuối cùng sẽ bị bỏ qua.
GL_TRIANGLE_STRIP Vẽ một chuỗi các tam giác đa giác có ba cạnh) với việc sử dụng v0, v1, v2, sau đó đến v2, v1, v3 (chú ý thứ tự), sau đó đến v2, v3, v4, và tiếp tục như thế. Thứ tự đảm bảo rằng các tam giác được vẽ cùng hướng sao cho dải tạo chính xác một phần của một bề mặt. Duy trì hướng là quan trọng cho một số thao tác, như là chọn lọc. (Xem "các bề mặt đa giác chọn lọc và đảo chiều") *n* phải lớn hơn hoặc bằng 3 cho mọi lần vẽ.
GL_TRIANGLE_FAN giống như GL_TRIANGLE_STRIP, ngoài trừ việc các đỉnh là v0, v1, v2, sau đó v0, v2, v3, sau đó v0, v3, v4, và tiếp tục như thế (xem Hình 2-7)
GL_QUADS Vẽ một chuỗi các tứ giác (là đa giác có bốn cạnh) với việc sử dụng v0, v1, v2, v3, sau đó v4, v5, v6, v7, và tiếp tục như thế.
Nếu *n* không phải là số chia hết cho 4, một, hai hoặc ba đỉnh cuối cùng bị bỏ qua.
GL_QUAD_STRIP Vẽ một chuỗi các tứ giác (đa giác bốn cạnh) bắt đầu với v0, v1, v3, v2, sau đó là v2, v3, v5, v4, sau đó là v4, v5, v7, v6, và tiếp tục như thế (xem Hình 2-7). *n* nhỏ nhất là 4 trước khi mọi thứ được vẽ. Nếu *n* là lẻ, đỉnh cuối cùng bị bỏ qua.
GL_POLYGON Vẽ một đa giác với việc sử dụng các điểm v0, ... , vn-1 là các đỉnh. *n* nhỏ nhất là 3, nếu không thì sẽ không có gì được vẽ. Ngoài ra đa giác chỉ rõ không được giao với chính nó và phải là lồi. Nếu các đỉnh không thỏa mãn các điều kiện này, kết quả là không xác định.

Giới hạn trong việc sử dụng glBegin() và glEnd()

Thông tin quan trong nhất về đỉnh là toạ độ của chúng, chúng được chỉ ra bởi lệnh **glVertex*().** Bạn cũng có thể cung cấp thêm dữ liệu đỉnh cho mỗi đỉnh-một màu, một vectơ pháp tuyến, toạ độ kết cấu hay mọi kết hợp của chúng-bằng việc sử dụng các lệnh đặc biệt.

Thêm nữa, một số lệnh khác là hợp lệ giữa cặp **glBegin()vàglEnd()**. Bảng 2-3 chứa một danh sách hoàn chỉnh của các lệnh hợp lệ đó.

Table 2-3 : Lệnh hợp lệ giữa glBegin() và glEnd()

Command	Purpose of Command	Reference
glVertex*()	set vertex coordinates	Chapter 2
glColor*()	set current color	Chapter 4
glIndex*()	set current color index	Chapter 4
glNormal*()	set normal vector coordinates	Chapter 2
glTexCoord*()	set texture coordinates	Chapter 9
glEdgeFlag*()	control drawing of edges	Chapter 2
glMaterial*()	set material properties	Chapter 5
glArrayElement()	extract vertex array data	Chapter 2
glEvalCoord*(), glEvalPoint*()	generate coordinates	Chapter 12
glCallList(), glCallLists()	execute display list(s)	Chapter 7

Không còn lệnh OpenGL khác hợp lệ giữa một cặp **glBegin()** và **glEnd()** và việc tạo hầu hết lời gọi OpenGL khác sinh ra một lỗi. Một số lệnh mảng đỉnh, như là **glEnableClientState()** và **glVertexPointer()**, khi gọi giữa **glBegin()** và **glEnd()**, được coi là không xác định nhưng không cần thiết tạo ra một lỗi. (Hơn nữa, thủ tục liên quan đến OpenGL, như là thủ tục **glX*()** là không xác định giữa **glBegin()** và **glEnd()**.) Các trường hợp này nên tránh, và việc gỡ lỗi chúng có thể khó khăn hơn.

Chú ý, tuy nhiên, chỉ các lệnh OpenGL là bị hạn chế, bạn có thể chắc chắn bao gồm cấu trúc ngôn ngữ lập trình khác (ngoại trừ cho các lời gọi, như là thủ tục **glX*()** như đã trình bày trước đây). Ví dụ, Ví dụ 2-4 vẽ phác thảo một đường tròn

Ví dụ 2-4 : Cấu trúc khác giữa glBegin() và glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
angle = 2*PI*i/circle_points;
glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

Chú ý: Ví dụ này không phải là cách hiệu quả nhất để vẽ một đường tròn, đặc biệt nếu bạn có ý định thực hiện nó nhiều lần. Các lệnh đồ họa sử dụng thường rất nhanh, nhưng đoạn mã này tính toán một góc và gọi các thủ tục **sin()** và **cos()** cho mỗi đỉnh; thêm nữa, có một vòng lặp ở trên. (Cách khác để tính toán các đỉnh của một đường tròn là sử dụng một thủ tục GLU; xem "Bậc hai: tô vẽ hình cầu, hình trụ, và đĩa" trong Chương 11.)

Nếu bạn cần vẽ nhiều hình tròn, tính toán các tọa độ của các đỉnh một lần và lưu chúng trong một mảng và tạo một danh sách hiển thị (xem Chương 7), hay sử dụng một mảng đỉnh để tô vẽ chúng.

Trừ khi chúng được biên dịch thành một danh sách hiển thị, tất cả các lệnh **glVertex*()** sẽ xuất hiện giữa một số kết hợp **glBegin()** và **glEnd()**. (nếu chúng xuất hiện một nơi nào đó, chúng không thể hoàn thành). Nếu chúng xuất hiện trong một danh sách hiển thị, chúng chỉ được chạy khi chúng xuất hiện giữa một **glBegin()** và một **glEnd()**. (Xem Chương 7 để biết thêm thông tin về danh sách hiển thị.)

Mặc dù nhiều lệnh được chấp nhận giữa **glBegin()** và **glEnd()**, các đỉnh chỉ được tạo khi một lệnh **glVertex*()** được đưa ra. Tại khoảng thời gian **glVertex*()** được gọi, OpenGL gán các đỉnh kết quả màu hiện tại, các tọa độ kết cấu, thông tin vectơ pháp tuyến, và tiếp tục như thế. Để hiểu điều này, xem thứ tự đoạn mã sau đây. Điểm đầu tiên được vẽ là đỏ, điểm thứ hai và ba là xanh dương, mặc dù có thêm nhiều lệnh màu.

```
glBegin(GL_POINTS);
glColor3f(0.0, 1.0, 0.0); /* green */
glColor3f(1.0, 0.0, 0.0); /* red */
glVertex(...);
glColor3f(1.0, 1.0, 0.0); /* yellow */
glColor3f(0.0, 0.0, 1.0); /* blue */
glVertex(...);
glVertex(...);
glEnd();
```

Bạn có thể sử dụng mọi kết hợp của 24 phiên bản của lệnh **glVertex*()** giữa **glBegin()** và **glEnd()**, mặc dù trong ứng dụng thực, tất cả các lời gọi trong ví dụ cụ thể đều có dạng như nhau. Nếu dữ liệu đỉnh của bạn chỉ ra nhất quán và lặp (ví dụ, **glColor***, **glVertex***, **glColor***, **glVertex***,...) bạn có thể nâng cao việc thực hiện chương trình bởi việc sử dụng mảng đỉnh (xem "mảng đỉnh").

Quản lí trạng thái cơ bản

Nếu trong phần trước, bạn đã thấy một ví dụ của một biến trạng thái, màu RGBA hiện tại, và nó có thể được liên kết như thế nào với một nguyên thủy. OpenGL duy trì nhiều trạng thái và biến trạng thái. Một đối tượng có thể được tô vẽ với ánh sáng, kết cấu, khử mặt khuất, hiệu ứng sương mù, và một số trạng thái tác động bề mặt của nó.

Ngầm định, hầu hết các trạng thái này lúc đầu không hoạt động. Các trạng thái này có thể tốn kém cho kích hoạt; ví dụ, việc bật vào ánh xạ kết cấu gần như là giảm chậm tốc độ tô vẽ một nguyên thủy. Tuy nhiên, chất lượng của ảnh sẽ hoàn thiện và trông thực hơn, do khả năng đồ họa nâng cao.

Để bật và tắt nhiều trạng thái này, sử dụng hai lệnh đơn giản sau:

*void **glEnable**(GLenum cap);*

*void **glDisable**(GLenum cap);*

***glEnable()** bật khả năng thực hiện, và **glDisable()** là tắt. Có trên 40 giá trị liệt kê có thể được truyền như là một biến đến **glEnable()** hay **glDisable()**. Một số ví dụ cho điều này là GL_BLEND (chúng điều khiển trộn các giá trị RGBA), GL_DEPTH_TEST (chúng điều khiển so sánh độ sâu và cập nhật vùng đệm độ sâu), GL_FOG (chúng điều khiển sương mù), GL_LINE_STIPPLE (các mẫu đường thẳng) GL_LIGHTING (bạn đặt ý tưởng), và vân vân.*

Bạn cũng có thể kiểm tra nếu một trạng thái là kích hoạt hay tắt.

*GLboolean **glIsEnabled**(GLenum capability)*

Trả về GL_TRUE hay GL_FALSE, phụ thuộc khả năng truy vấn được kích hoạt hiện tại không.

Các trạng thái bạn vừa được thấy có hai thiết lập: bật và tắt. Tuy nhiên, hầu hết các thủ tục OpenGL thiết lập các giá trị cho các biến trạng thái phức tạp hơn. Ví dụ, thủ tục **glColor3f()** thiết lập ba trạng thái, chúng là phần của trạng thái GL_CURRENT_COLOR. Có năm thủ tục truy vấn sử dụng để tìm ra những giá trị nào được thiết lập cho nhiều trạng thái:

*void **glGetBooleanv**(GLenum pname, GLboolean *params);*

*void **glGetIntegerv**(GLenum pname, GLint *params);*

*void **glGetFloatv**(GLenum pname, GLfloat *params);*

*void **glGetDoublev**(GLenum pname, GLdouble *params);*

*void **glGetPointerv**(GLenum pname, GLvoid **params);*
Thu được các biến trạng thái Boolean, integer, floating-point, double-precision, hay con trỏ. Đối số pname là một hằng số biểu tượng mà nó chỉ ra biến trạng thái để trả về, và params là một con trỏ đến một mảng của kiểu chỉ định là nơi đặt dữ liệu trả về. Xem các bảng trong Mục lục B để có các giá trị hợp lí cho pname. Ví dụ, để đặt màu RGBA hiện tại, một bảng trong Mục lucc B đề xuất bạn sử dụng

***glGetIntegerv**(GL_CURRENT_COLOR, params) hay **glGetFloatv**(GL_CURRENT_COLOR, params). Một kiểu hoán đổi được thực hiện nếu cần thiết trả về các biến mong muốn như kiểu dữ liệu yêu cầu.*
Hầu hết xử lí các thủ tục truy vấn này, nhưng không phải tất cả, yêu cầu thông tin trạng thái thu được. (Xem "Các lệnh truy vấn" trong Mục lục B) đối với việc thêm 16 thủ tục truy vấn.

Hiển thị các điểm, đường thẳng, và đa giác

Ngầm định, một điểm được vẽ là một điểm đơn lẻ trên màn hình, một đường thẳng được vẽ liên tục và có độ rộng một điểm ảnh, các đa giác được vẽ là đặc. Các đoạn sau đây trình bày chi tiết làm thế nào thay đổi các chế độ hiển thị ngầm định này.

Chi tiết điểm

Để điều khiển kích cỡ một điểm tô vẽ, sử dụng **glPointSize()** và cung cấp kích cỡ mong muốn theo điểm ảnh như một đối số.
*void **glPointSize**(GLfloat size);*
Thiết lập độ rộng theo đơn vị điểm ảnh cho các điểm được tô vẽ; kích cỡ phải lớn hơn 0.0 và ngầm định là 1.0.
Tập các điểm ảnh thực sự trên màn hình mà chúng được vẽ với độ rộng điểm khác nhau phụ thuộc vào việc khử răng cưa được kích hoạt hay không. (Khử răng cưa là một kĩ thuật để làm trơn điểm và đường thẳng khi chúng được tô vẽ); xem "Khử răng cưa" trong Chương 6 để biết thêm chi tiết). Nếu việc khử răng cưa không kích hoạt (ngầm định), độ rộng phân số được làm tròn đến độ rộng nguyên, và vùng ô vuông căn lề theo màn hình của các điểm ảnh được vẽ. Do đó, nếu độ rộng là 1.0, kích cỡ ô vuông là 1x1 điểm ảnh; nếu độ rộng là 2.0, kích cỡ ô vuông là 2x2 điểm ảnh, và cứ như thế.
Nếu khử răng cưa được kích hoạt, một *group* vòng tròn các điểm ảnh được vẽ, và các điểm ảnh trên được biên thường được vẽ kém hơn cường độ cực đại gửi đến đỉnh một bề mặt trơn hơn. Trong chế độ này, độ rộng không nguyên không được làm tròn.
Hầu hết các cài đặt OpenGL hỗ trợ các kích cỡ điểm rất lớn. Kích cỡ cực đại đối với các điểm ảnh khử răng cưa là có thể truy vấn, nhưng thông tin tương tự là không có sẵn để hiểu, tức là các điểm răng cưa. Một cài đặt cụ thể, tuy nhiên, có thể giới hạn kích cỡ chuẩn, tức là các điểm không nhỏ hơn kích cỡ điểm răng cưa cực đại của nó, làm tròn đến giá trị nguyên gần nhất. Bạn có thể thu được giá trị dấu chấm động này bằng việc sử dụng GL_POINT_SIZE_RANGE với **glGetFloatv()**.

Chi tiết đường thẳng

Với OpenGL, bạn có thể chỉ ra các đường thẳng với độ rộng khác nhau và các đường thẳng mà được *vẽ đứt nét* trong nhiều cách khác nhau- nét chấm, nét gạch ngang, vẽ với các nét chấm và gạch luân phiên, và cứ như thế.

Độ rộng đường thẳng

*void **glLineWidth**(GLfloat width);*
Thiết lập độ rộng các điểm ảnh cho đường thẳng tô vẽ; độ rộng phải lớn hơn 0.0 và ngầm định là 1.0.
Tô vẽ thực tế của các đường thẳng bị tác động bởi chế độ khử răng cưa, theo cách tương tự như đối với điểm. (Xem "Khử răng cưa " trong Chương 6). Không có khử răng cưa, độ rộng của 1, 2 và 3 vẽ các đường thẳng độ rộng 1, 2, và 3. Nếu có khử răng cưa, độ rộng các đường thẳng có thể không nguyên, và các điểm ảnh trên đường biên thường được vẽ nhỏ hơn cường độ cực đại. Với các kích cỡ điểm, một cài đặt OpenGL cụ thể có thể hạn chế độ rộng của các đường thẳng không khử răng cưa đến độ rộng đường thẳng khử răng cưa cực đại của nó, làm tròn đến giá trị số nguyên gần nhất. Bạn có thể thu được giá trị dấu chấm động này bằng việc sử dụng GL_LINE_WIDTH_RANGE với **glGetFloatv()**.

Chú ý: Nhớ rằng đường thẳng ngầm định có độ rộng 1 điểm ảnh, nên chúng xuất hiện rộng hơn trên màn hình có độ phân giải thấp. Với màn hình máy tính, điều này thường không phải là vấn đề, nhưng nếu bạn đang sử dụng OpenGL để tô vẽ một máy vẽ độ phân giải cao, các đường thẳng độ rộng 1 điểm ảnh có thể gần như không nhìn thấy, bạn cần tính toán kích cỡ vật lí của các điểm ảnh.

Nâng cao

Với đường thẳng rộng không được khử răng cưa, độ rộng đường thẳng không được đo vuông góc với đường thẳng. Thay vào đó, nó được đo theo hướng y nếu giá trị tuyệt đối của độ dốc là nhỏ hơn 1.0; và ngược lại, nếu nó được đo theo hướng x. Việc tô vẽ của đường thẳng khử răng cưa chính xác tương đương với việc tô vẽ của một hình chữ nhật đặc với độ rộng đã được cho, canh giữa trên đúng đường thẳng.

Đường thẳng nét đứt

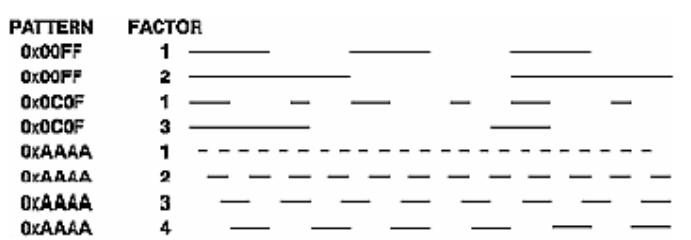
Để tạo các đường thẳng nét đứt (chấm hay gạch ngang), bạn sử dụng lệnh **glLineStipple()** để định nghĩa kiểu nét đứt, và sau đó bạn có thể vẽ đường thẳng với **glEnable()**.

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

*void **glLineStipple**(GLint factor, GLushort pattern);*

*Thiết lập mẫu nét đứt hiện tạo cho các đường thẳng. Đối số pattern là một chuỗi 16 bit với các giá trị 0 và 1, và việc lặp lại của nó là cần thiết để tạo nét đứt của một đường thẳng đã cho. Giá trị 1 có nghĩa thực hiện vẽ, và 0 là không vẽ, dựa trên cơ sở 1x1 điểm ảnh, bắt đầu với bít bậc thấp của pattern. pattern có thể được kéo dài bằng việc sử dụng factor, chúng nhân mỗi chuỗi con liên tục các bít 0 và 1. Do đó, nếu ba bít 1 liên tiếp xuất hiện trong pattern, chúng được kép dài đến sáu nếu thừa số là 2. Thừa số được kẹp giữa 1 và 255. Đường đứt nét có thể thực hiện bằng việc truyền GL_LINE_STIPPLE cho **glEnable()**; nó được huỷ kích hoạt bằng việc truyền đối số tượng tự đến **glDisable()**.*

Với ví dụ có trước và mẫu 0x3F07 (chúng dịch thành 0011111100000111 theo nhị phân), một đường thẳng sẽ được vẽ với 3 điểm ảnh bật, sau đó 5 điểm tắt, 6 điểm bật và 2 điểm tắt. (Nếu điều này được xem là chậm, nhớ rằng bít bậc thấp được sử dụng đầu tiên). Nếu thừa số là 2, mẫu sẽ được kéo dài ra: 6 điểm ảnh bật, 10 điểm tắt, 12 bật và 4 tắt. Hình 2-8 chỉ ra các đường thẳng được vẽ với các mẫu khác nhau và lặp lại hệ số. Nếu bạn không thể vẽ đường thẳng nét đứt, việc vẽ tiến hành như thể mẫu là 0xFFFF với hệ số 1. (Sử dụng **glDisable()** với GL_LINE_STIPPLE để huỷ nét đứt) Chú ý rằng nét đứt có thể được sử dụng kết hợp với độ rộng đường thẳng để tạo độ rộng các đường thẳng nét đứt.



Hình 2-8 : Các đường thẳng nét đứt

Một cách để hiểu về nét đứt là các đường thẳng đang được vẽ, mỗi lần mẫu được đổi thành bít 1, một điểm ảnh được vẽ (hay hệ số điểm ảnh được vẽ, nếu hệ số khác 1). Khi một chuỗi các đoạn thẳng kết nối được vẽ giữa một cặp **glBegin()** và **glEnd()**, pattern tiếp tục biến đổi một đoạn thành đoạn tiếp theo. Cách này, một pattern nét đứt liên tục đi hết một chuỗi các đoạn thẳng kết nối. Khi **glEnd()** được chạy, pattern được thiết lập lại, và nếu các đường thẳng được vẽ thêm trước khi đứt nét bị hủy- nét đứt khởi tạo lại lúc bắt đầu của pattern. Nếu bạn đang vẽ đường thẳng với GL_LINES. Pattern thiết lập lại cho mỗi đường thẳng độc lập.

Ví dụ 2-5 minh học kết quả của việc vẽ với một vặp các pattern đứt nét khác nhau và độ rộng đường thẳng. Nó cũng minh họa những gì xảy ra nếu các đường thẳng được vẽ là một chuỗi các đoạn cụ thể thay vì một dải đường thẳng kết nối đơn lẻ. Kết quả của việc chạy chương trình thể hiện trong Hình 2-9.

Hình 2-9 : Độ rộng các đường thẳng nét đứt

Ví dụ 2-5 : Các mẫu đường thẳng nét đứt: lines.c

```
#include <GL/gl.h>
#include <GL/glut.h>
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES);
glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
int i;
glClear (GL_COLOR_BUFFER_BIT);
/* select white for all lines */
glColor3f (1.0, 1.0, 1.0);
/* in 1st row, 3 lines, each with a different stipple */
glEnable (GL_LINE_STIPPLE);
glLineStipple (1, 0x0101); /* dotted */
drawOneLine (50.0, 125.0, 150.0, 125.0);
glLineStipple (1, 0x00FF); /* dashed */
drawOneLine (150.0, 125.0, 250.0, 125.0);
glLineStipple (1, 0x1C47); /* dash/dot/dash */
drawOneLine (250.0, 125.0, 350.0, 125.0);
/* in 2nd row, 3 wide lines, each with different stipple */
glLineWidth (5.0);
glLineStipple (1, 0x0101); /* dotted */
drawOneLine (50.0, 100.0, 150.0, 100.0);
glLineStipple (1, 0x00FF); /* dashed */
drawOneLine (150.0, 100.0, 250.0, 100.0);
glLineStipple (1, 0x1C47); /* dash/dot/dash */
drawOneLine (250.0, 100.0, 350.0, 100.0);
glLineWidth (1.0);
/* in 3rd row, 6 lines, with dash/dot/dash stipple */
/* as part of a single connected line strip */
glLineStipple (1, 0x1C47); /* dash/dot/dash */
glBegin (GL_LINE_STRIP);
for (i = 0; i < 7; i++)
glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
glEnd ();
/* in 4th row, 6 independent lines with same stipple */
for (i = 0; i < 6; i++) {
drawOneLine (50.0 + ((GLfloat) i * 50.0), 50.0,
50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
```

```
}
/* in 5th row, 1 line, with dash/dot/dash stipple */
/* and a stipple repeat factor of 5 */
glLineStipple (5, 0x1C47); /* dash/dot/dash */
drawOneLine (50.0, 25.0, 350.0, 25.0);
glDisable (GL_LINE_STIPPLE);
glFlush ();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (400, 150);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Chi tiết đa giác

Đa giác thường được vẽ bởi việc tô kín tất cả các điểm ảnh nằm trong đường biên, nhưng bạn cũng có thể vẽ chúng là các khung đa giác hay đơn giản là trỏ tại các đỉnh. Một đa giác đặc có thể phủ kín liên tục hay đứt nét theo một mẫu nào đó. Mặc dù các chi tiết chính xác được bỏ sót ở đây, các đa giác đặc được vẽ theo cách mà nếu các đa giác liền kề chung nhau một cạnh hay đỉnh, các điểm ảnh tạo các cạnh hay đỉnh được vẽ đúng một lần-chúng được bao gồm trong một đa giác duy nhất. Điều này được thực hiện sao cho các đa giác trong suốt một phần không có các đỉnh của chúng được vẽ hai lần, chúng sẽ tạo các cạnh đó xuất hiện tối hơn (hay sáng hơn, phụ thuộc vào màu bạn đang vẽ). Chú ý rằng nó có thể dẫn đến đa giác hẹp lại không có các điểm ảnh tô kín trong một hay nhiều hàng hoặc nhiều cột điểm ảnh. Các đa giác khử răng cưa thì phức tạo hơn so với điểm và đường thẳng. (Xem "Khử răng cưa" trong Chương 6 để biết thêm chi tiết).

Đa giác là điểm, khung, hay đặc

Một đa giác có hai mặt – trước và sau- và có thể được tô vẽ khác nhau phụ thuộc vào mặt nào đang đối diện với người quan sát. Điều này cho phép bạn có mô hình quan sát không có mặt trước của các đối tượng đặc nơi có một sự khác biệt rõ ràng giữa các phần mà chúng ở bên trong và những phần kia ở bên ngoài. Ngầm định, cả hai bề mặt trước và sau được vẽ trong cùng một hướng. Để thay đổi điều này, hay chỉ vẽ các khung hay các đỉnh, sử dụng **glPolygonMode()**.

*void **glPolygonMode**(GLenum face, GLenum mode);*

Điều khiển việc vẽ chế độ cho một bề mặt trước và sau của đa giác. Biến bề mặt có thể là GL_FRONT_AND_BACK, GL_FRONT, hoặc GL_BACK; chế độ có thể là GL_POINT, GL_LINE, hay GL_FILL để chỉ ra các đa giác có thể được vẽ như là các điểm, khung hay được tô kín.

Ngầm định, cả hai bề mặt trước và sau được vẽ tô kín.

Ví dụ, bạn có thể có các bề mặt trước tô kín và bề mặt sau khung với hai lời gọi thủ tục sau:

```
glPolygonMode(GL_FRONT, GL_FILL);
```

```
glPolygonMode(GL_BACK, GL_LINE);
```

Các bề mặt đa giác chọn lọc và đảo chiều

Theo quy ước, các đa giác mà các đỉnh của nó xuất hiện theo hướng ngược chiều kim đồng hồ trên màn hình được gọi là bề mặt trước. Bạn có thể vẽ bề mặt đặc hợp lệ bất kì –toán học gọi đây là một bề mặt đa chiều định hướng được (hình cầu, bánh rán, và ấm trà là định hướng; chai Klein và dải Möbius thì không- từ các đa giác hướng nhất quán. Nói cách khác, bạn có thể sử dụng tất cả các đa giác theo chiều kim đồng hồ, hay tất cả các đa giác ngược chiều kim đồng hồ. (Đây là định nghĩa cơ sở toán học về *định hướng*)

Giả sử bạn có mô tả nhất quán một mô hình của một bề mặt định hướng nhưng bạn hướng theo chiều kim đồng hồ phía bên ngoài. Bạn cần đổi chỗ những gì OpenGL tính toán cho mặt sau bằng việc sử dụng hàm **glFrontFace()**, nó cung cấp hướng mong muốn cho các đa giác hướng bề mặt trước.

*void **glFrontFace**(GLenum mode);*

Điều khiển các đa giác bề mặt trước được xác định như thế nào. Ngầm định, chế độ là GL_CCW, chúng tương ứng với hướng ngược chiều kim đồng hồ của các đỉnh thứ tự của một đa giác được chiếu trong tọa độ cửa sổ. Nếu chế độ là GL_CW, các bề mặt theo chiều kim đồng hồ được xét bề mặt trước.

Trong bề mặt khép kín xây dựng hoàn chỉnh từ các đa giác đục với một hướng nhất quán, bề mặt sau của đa giác không thể thấy- chúng luôn luôn bị che bởi các bề mặt trước của đa giác. Nếu bạn ở bên ngoài bề mặt này, bạn có thể chọn lọc để loại bỏ các đa giác mà OpenGL xác định là bề mặt sau. Tương tự, nếu bạn ở bên trong đối tượng, chỉ đa giác bề mặt sau được nhìn thấy. Để chỉ dẫn OpenGL để loại bỏ đa giác bề mặt sau hay trước, sử dụng lệnh **glCullFace()** và có thể chọn lọc với **glEnable()**.

*void **glCullFace**(GLenum mode);*

*Chỉ ra những đa giác nào sẽ bị loại bỏ (theo chọn lọc) trước khi chúng được chuyển đổi sang tọa độ màn hình. Chế độ hoặc là GL_FRONT, GL_BACK, hoặc là GL_FRONT_AND_BACK để chỉ ra bề mặt trước, bề mặt sau, hay tất cả các đa giác. Để có hiệu lực, chọn lọc phải được bật với việc sử dụng **glEnable()** với GL_CULL_FACE; nó có thể bị hủy với **glDisable()** và đối số tương tự.*

Nâng cao

Trong điều kiện kĩ thuật, việc xác định mặt nào của một đa giác là mặt trước hay sau phụ thuộc vào dấu của diện tích đa giác tính toán trong tọa độ cửa sổ. Một cách để tính toán diện tích này là

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i$$

Với xi và yi là tọa độ cửa sổ x và y của đỉnh thứ i của đa giác n đỉnh và $i \oplus 1$ is $(i+1) \bmod n$.

Giả sử rằng GL_CCW đã được xác định, nếu a>0, đa giác tương ứng với đỉnh đó được xét là mặt trước; ngược lại, nó là mặt sau. Nếu GL_CW được xác định và nếu a<0, thì đa giác tương ứng là mặt trước; ngược lại, nó là mặt sau.

Thử điều này.

Sửa đổi Ví dụ 2-5 bằng việc thêm một số đa giác tô kín. Thử nghiệm với những màu khác. Thử với các chế độ đa giác khác. Cũng cho phép chọn lọc để thấy hiệu quả của nó.

Đa giác điểm

Ngầm định, các đa giác tô kín được vẽ với một mẫu liên tục. Chúng cũng có thể được tô kín với một mẫu điểm canh lề cửa sổ 32bit x 32 bít, chúng được bạn chỉ ra với **glPolygonStipple()**.

*void **glPolygonStipple**(const GLubyte *mask);*

*Định nghĩa mẫu điểm hiện tại cho đa giác tô kín. Đối số mask là một con trỏ đến ma trận bít 32x32 mà nó được giải thích như một mask của các bít 0 và 1. Ở vị trí bít là 1, điểm tương ứng trong đa giác được vẽ, và bít là 0, không có điểm nào được vẽ. Hình 2-10 chỉ ra một mẫu điểm được xây dựng từ kí tự trong mask. Đa giác điểm được bật và hủy bằng việc sử dụng **glEnable()** và*

glDisable() *với GL_POLYGON_STIPPLE là đối số. Giải thích của dữ liệu mask được tác động bằng các chế độ* ***glPixelStore*()*** *GL_UNPACK*. (Xem " điều khiển các chế độ lưu trữ điểm " ở Chương 8)*

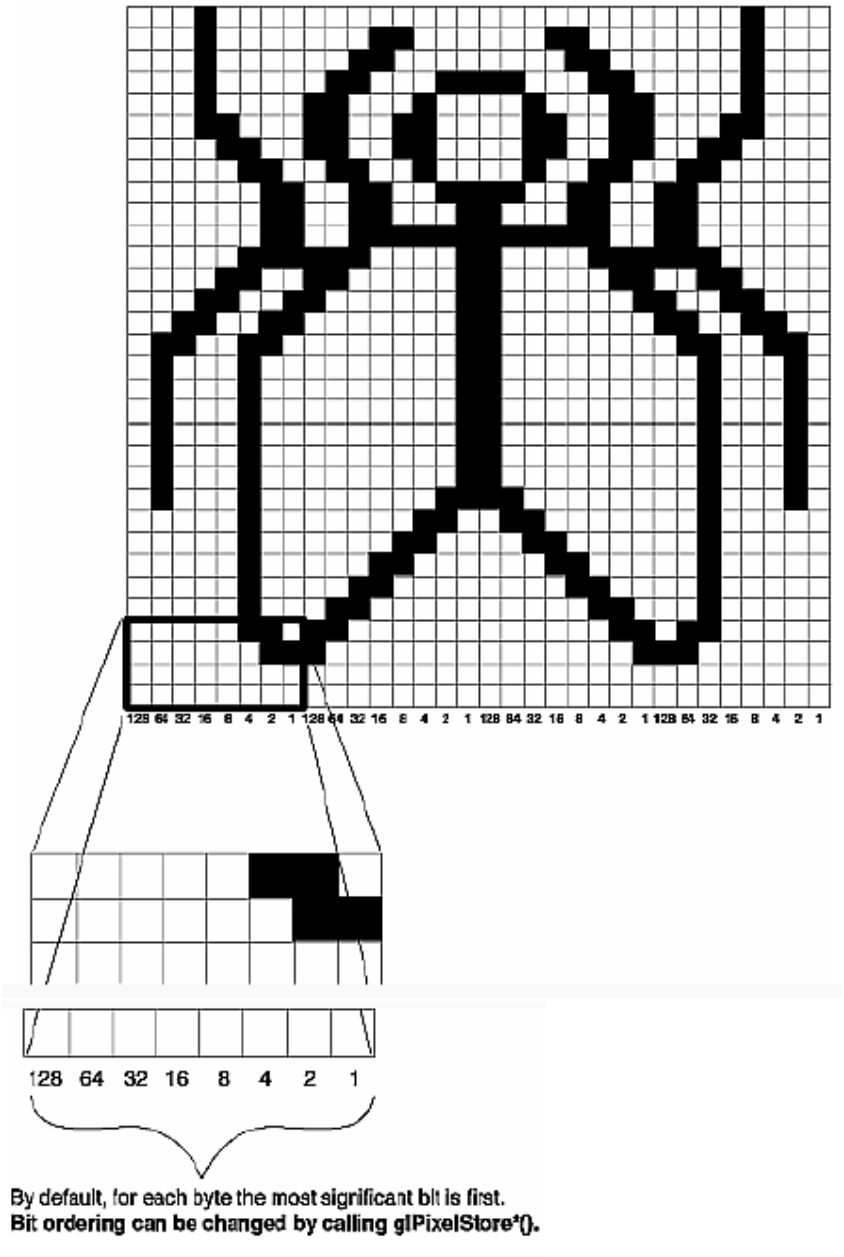
Ngoài việc định nghĩa mẫu điểm đa giác hiện tại, bạn phải bật mẫu điểm:

```
glEnable(GL_POLYGON_STIPPLE);
```

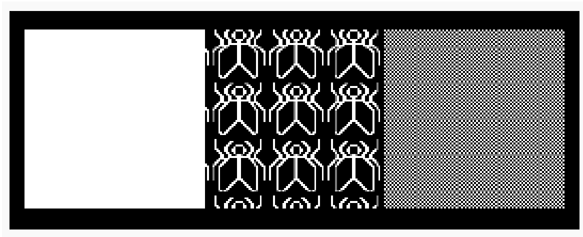
Sử dụng **glDisable()** với đối số tương tự để hủy đa giác điểm.

Hình 2-11 chỉ ra kết quả của các đa giác vẽ không bằng điểm và sau đó với hai mẫu điểm khác nhau. Chương trình được chỉ ra trong Ví dụ 2-6.

Sự thay đổi trắng thành đen (từ Hình 2-10 đến Hình 2-11) xảy ra vì chương trình vẽ màu trắng trên một nền đen, bằng việc sử dụng mẫu trong Hình 2-10 như một bút chì.



Hình 2-10 : Xây dựng một mẫu điểm đa giác



Hình 2-11 : Đa giác được xây dựng bởi các điểm

Example 2-6 : Các mẫu điểm đa giác: polys.c

```
#include <GL/gl.h>
#include <GL/glut.h>
void display(void)
{
GLubyte fly[] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};
GLubyte halftone[] = {
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
/* draw one solid, unstippled rectangle, */
/* then two stippled rectangles */
glRectf (25.0, 25.0, 125.0, 125.0);
glEnable (GL_POLYGON_STIPPLE);
```

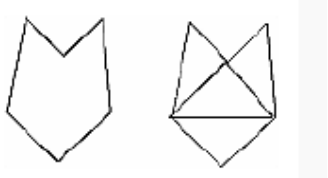
```
glPolygonStipple (fly);
glRectf (125.0, 25.0, 225.0, 125.0);
glPolygonStipple (halftone);
glRectf (225.0, 25.0, 325.0, 125.0);
glDisable (GL_POLYGON_STIPPLE);
glFlush ();
}
void init (void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (350, 150);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Bạn có thể muốn sử dụng danh sách hiển thị để chứa các mẫu điểm đa giác để có hiệu quả tối ưu. (Xem "triết lí thiết kế danh sách hiển thị" trong Chương 7).

Đánh dấu các cạnh biên đa giác

Nâng cao

OpenGL có thể chỉ tô vẽ một đa giác lồi, nhưng nhiều đa giác không lồi phát sinh trong thực tế. Để vẽ những đa giác không lồi này, bạn thường chia nhỏ chúng thành các đa giác lồi-thường là các tam giác, như chỉ ra trong Hình 2-12-và sau đó vẽ các tam giác. Thật không may, nếu bạn phân tách một đa giác tổng thành các tam giác và vẽ các tam giác, bạn không thể sử dụng **glPolygonMode()** để vẽ các khung đa giác, vì bạn đặt tất cả khung tam giác bên trong chúng. Để giải quyết vấn đề này, bạn có thể nói cho OpenGL một thứ tự đỉnh cụ thể một cạnh biên; OpenGL duy trì vết thông tin này bằng việc truyền dọc mỗi đỉnh một bít để chỉ ra đỉnh đó được theo sau bởi một cạnh biên hay không. Sau đó, khi một đa giác được vẽ trong chế độ GL_LINE, các cạnh không thuộc biên không được vẽ. Trong Hình 2-12, các đường gạch ngang biểu diễn các cạnh được thêm vào.



Hình 2-12 : Chia nhỏ một đa giác không lồi.

Ngầm định, tất cả các đỉnh được đánh dấu như một cạnh biên có trước, nhưng bạn có thể tự điều khiển việc thiết lập cờ hiệu của cạnh với lệnh **glEdgeFlag*()**. Lệnh này được sử dụng giữa các cặp

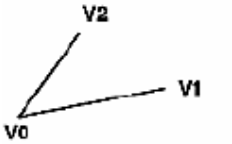
glBegin() and **glEnd(),** và nó tác động tất cả các đỉnh được chỉ ra sau khi lời gọi **glEdgeFlag()** tiếp theo vẫn được tạo. Nó áp dụng duy nhất đến các đỉnh chỉ ra cho các đa giác, tam giác, và tứ giác, không áp dụng đối với dải của tam giác hay tứ giác.

*void **glEdgeFlag**(GLboolean flag);*

*void **glEdgeFlagv**(const GLboolean *flag);*

Chỉ ra một đỉnh sẽ được xét hay không như khởi tạo một cạnh biên của một đa giác. Nếu cờ hiệu là GL_TRUE, cờ hiệu của cạnh được thiết lập là TRUE (ngầm định), và mọi đỉnh tạo được xét cho cạnh biên có trước cho đến khi các hàm này được gọi lại với cờ hiệu là GL_FALSR.

Theo một ví dụ, Ví dụ 2-7 vẽ khung được chỉ ra trong Hình 2-13



Hình 2-13 : Khung đa giác được vẽ với việc sử dụng cờ hiệu của cạnh

Ví dụ 2-7 : Đánh dấu các cạnh biên đa giác.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
glEdgeFlag(GL_TRUE);
glVertex3fv(V0);
glEdgeFlag(GL_FALSE);
glVertex3fv(V1);
glEdgeFlag(GL_TRUE);
glVertex3fv(V2);
glEnd();
```

Vectơ pháp tuyến

Một *vectơ pháp tuyến* (hay pháp tuyến, cho ngắn gọn) là một vectơ mà trỏ theo một hướng vuông góc với một bề mặt. Đối với bề mặt phẳng, một hướng vuông góc là như nhau cho mọi điểm trên bề mặt, nhưng đối với bề mặt cong tổng quát, vectơ pháp tuyến có thể khác tại mỗi điểm trên bề mặt. Với OpenGL, bạn có thể chỉ ra một pháp truyến cho mỗi đa giác hay cho mỗi đỉnh. Các đỉnh của cùng một đa giác có thể dùng chung pháp tuyến (đối với một bề mặt phẳng) hay có các pháp tuyến khác nhau (đối với bề mặt cong). Nhưng bạn không thể thiết kế các pháp tuyến nơi nào hơn tại các đỉnh.

Một vectơ pháp tuyến của một đối tượng định nghĩa hướng bề mặt của nó trong không gian- một cách cụ thể, hướng của nó liên quan đến các nguồn sáng. Các vectơ này được sử dụng bởi OpenGL để xác định có bao nhiêu nguồn sáng mà đối tượng nhận được tại đỉnh của nó. Ánh sáng - bản thân nó là một chủ đề lớn – là một đề tài của Chương 5, và bạn có thể muốn xem lại thông tin dưới đây sau khi bạn đã đọc chương đó. Vectơ pháp tuyến được trình bày ngắn gọn ở đây vì bạn định nghĩa vectơ pháp tuyến cho một đối tượng tại cùng thời điểm bạn định nghĩa hình học của đối tượng.

Bạn sử dụng **glNormal*()** để thiết lập pháp tuyến hiện tại đến giá trị của đối số được truyền trong đó. Lời gọi tiếp theo **glVertex*()** dẫn đến các đỉnh chỉ ra được gán pháp tuyến hiện tại. Thông thường, mỗi đỉnh có một pháp tuyến khác nhau, chúng đòi hỏi phải có một loạt lời gọi luân phiên, như trong Ví dụ 2-8.

Ví dụ 2-8 : Pháp tuyến bề mặt tại các đỉnh

```
glBegin (GL_POLYGON);
glNormal3fv(n0);
glVertex3fv(v0);
glNormal3fv(n1);
glVertex3fv(v1);
glNormal3fv(n2);
glVertex3fv(v2);
glNormal3fv(n3);
glVertex3fv(v3);
```

```
glEnd();
```

*void **glNormal3**{bsidf}(TYPEnx, TYPEny, TYPEnz);*
*void **glNormal3**{bsidf}**v**(const TYPE *v);*
*Thiết lập vectơ pháp tuyến hiện tại như được chỉ ra bởi các đối số. Phiên bản không có vectơ (không có **v**)có ba đối số, chúng chỉ ra một vectơ (nx, ny, nz) mà chúng được lấy làm pháp tuyến. Với một sự lựa chọn, bạn có thể sử dụng phiên bản vectơ của hàm này (với **v**) và cung cấp một mảng đơn với ba phần tử chỉ ra pháp tuyến mong muốn. Phiên bản **b, s** và **i** co các giá trị biến của chúng tuyến tính trong miền [-1.0, 1.0].*

Không có thủ thuật để tìm kiếm pháp tuyến cho một đối tượng- hầu như phổ biến, bạn phải thực hiện một số tính toán mà có thể bao gồm việc lấy đạo hàm – nhưng có một số kĩ thuật và mánh lới bạn có thể sử dụng để kích hoạt tác động nào đó.

Mục lục E giải thích làm thế nào để tìm vectơ pháp tuyến cho các bề mặt. nếu bạn đã biết thực hiện điều này như thế nào, nếu bạn có thể hy vọng luôn được cung cấp vectơ pháp tuyến, hay nếu bạn không muốn sử dụng điều kiện ánh sáng được cung cấp bởi điều kiện ánh sáng OpenGL, bạn không cần đọc mục lục này.

Chú ý rằng tại một điểm đã cho trên một bề mặt, hai vectơ là vuông góc đến bề mặt, và chúng trỏ đến hai hướng đối diện nhau. Theo quy ước, pháp tuyến là vectơ trỏ đến phía ngoài của bề mặt đang làm mô hình. (Nếu bạn đảo mặt trong và ngoài trong mô hình của bạn, chỉ thay đổi mọi vectơ pháp tuyến từ (x, y, z) thành (- &xgr; , -y, -z))

Ngoài ra, luôn nhớ rằng do vectơ pháp tuyến chỉ ra hướng duy nhất, độ dài của chúng thường không được quan tâm. Bạn có thể chỉ ra pháp tuyến với độ dài bất kì, nhưng cuối cùng chúng phải được chuyển thành độ dài là 1 trước khi việc tính toán ánh sáng được thực hiện. (Một vectơ mà có độ dài là 1 được gọi là độ dài đơn vị, hay pháp tuyến hóa) . Nói chung, bạn sẽ cung cấp pháp tuyến hóa vectơ pháp tuyến.Để tạo một vectơ pháp tuyến có độ dài đơn vị, chia mỗi thành phần x, y, z của nó cho độ dài của pháp tuyến;

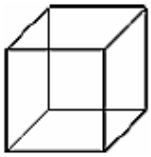
$$\text{Length} = \sqrt{x^2 + y^2 + z^2}$$

Vectơ pháp tuyến duy trì pháp tuyến hóa miễn là biến đổi mô hình của bạn bao gồm phép quay và tịnh tiến duy nhất. (Xem Chương 3 về trình bày của các phép biến đổi). Nếu bạn thực hiện biến đổi không đều (Như là co dãn hay nhân với một ma trận biến dạng), hay nếu bạn chỉ ra pháp tuyến với độ dài khác một, thì bạn nền có pháp tuyến hóa tự động OpenGL vectơ pháp tuyến của bạn sau các phép biến đổi. Để thực hiện điều này, gọi **glEnable()** với GL_NORMALIZE là đối số của nó. Ngầm định, pháp tuyến hóa tự động là không thực hiện. Chú ý rằng pháp tuyến hóa tự động điển hình yêu cầu cộng thêm tính toán mà có thể giảm hiệu suất ứng dụng của bạn

1. Các mảng đỉnh (Vertex Arrays)

Bạn có thể đã được thông báo rằng OpenGL yêu cầu nhiều lời gọi hàm để tô vẽ hình học nguyên thủy. Vẽ một đa giác 20 cạnh yêu cầu 22 lời gọi hàm: một lời gọi đến **glBegin()**, mỗi lời gọi hàm cho một đỉnh, và lời gọi cuối cùng đến **glEnd()**. Trong hai ví dụ về đoạn mã trước, thông tin thêm vào (các cờ hiệu của cạnh biên đa giác hay pháp tuyến bề mặt) bổ sung thêm các lời gọi hàm cho mỗi đỉnh. Điều này có thể tăng gấp đôi hay ba lần số lượng lời gọi hàm yêu cầu cho một đối tượng hình học. Đối với một số hệ thống, các lời gọi hàm tốn rất nhiều chi phí và có thể giảm hiệu suất

Thêm một vấn đề nữa là thừa việc xử lí các đỉnh mà được dùng chung giữa các đa giác liền kề. Ví dụ, hình lập phương trong Hình 2-14 có 6 mặt và 8 đỉnh dùng chung. Thật không may, việc sử dụng phương pháp chuẩn của việc mô tả đối tượng này, mỗi đỉnh sẽ được chỉ rõ ba lần: mỗi lần cho mọi mặt mà sử dụng nó. Do đó, 24 đỉnh sẽ được xử lí, mặc dù chỉ cần 8 là đủ.



Hình 2-14 : Sáu mặt; tám đỉnh dùng chung

OpenGL có các thủ tục mảng đỉnh mà cho phép bạn chỉ ra rất nhiều dữ liệu liên quan đỉnh chỉ với một số mảng và để truy cập dữ liệu với một số lời gọi hàm tương tự. Việc sử dụng các thủ tục mảng đỉnh, tất cả 20 đỉnh trong một đa giác 20 cạnh có thể đặt trong một mảng và gọi với một hàm. Nếu mỗi đỉnh cũng có một pháp tuyến bề mặt, tất cả 20 pháp tuyến bề mặt sẽ được đặt trong một mảng khác và cũng được gọi với một hàm.

Miền dữ liệu trong mảng đỉnh có thể tăng hiệu suất ứng dụng của bạn. Việc sử dụng các mảng đỉnh giảm số lượng lời gọi hàm, chúng cải tiến hiệu suất. Ngoài ra, việc sử dụng các mảng đỉnh có thể cho phép xử lí không thừa của các đỉnh dùng chúng. (Các đỉnh dùng chúng không được hỗ trợ trên tất cả các cài đặt của OpenGL)

Chú ý: Các mảng đỉnh là tiêu chuẩn trong phiên bản 1.1 của OpenGL nhưng không không phải là phần đặc tả của OpenGL 1.0. Với OpenGL 1.0, một số nhà cung cấp đã cài đặt mảng đỉnh là một mở rộng.

Có ba bước để sử dụng mảng đỉnh tô vẽ hình học:

1. Kích hoạt (có thể thực hiện) lên sáu mảng, mỗi mảng chứa một kiểu dữ liệu khác nhau: tọa độ đỉnh, màu RGBA, màu chỉ số, pháp tuyến bề mặt, tọa độ kết cấu, hay các cờ hiệu của cạnh đa giác.
2. Đặt dữ liệu thành mảng hay các mảng. Các mảng được truy cập theo địa chỉ của (tức là, trỏ đến) vị trí bộ nhớ của chúng. Trong mô hình khách- chủ, dữ liệu này được lưu trữ trong không gian địa chỉ máy khách.
3. Vẽ hình học với dữ liệu. OpenGL thu được dữ liệu từ mọi mảng kích hoạt bằng việc tham chiếu lại con trỏ. Trong mô hình khách chủ, dữ liệu được chuyển giao đến nơi địa chỉ máy chủ, dữ liệu được chuyển thành đỏ đến không gian địa chỉ của máy chủ. Có ba cách để thực hiện điều này:
   1.Truy cập các phần tử mảng riêng lẻ (dao động xung quanh một cách ngẫu nhiên)
   2. Tạo một danh sách các phần tử mảng riêng lẻ (dao động xung quanh có phương pháp)
   3. Xử lí các phần tử mảng theo trình tự
   Tham chiếu lại phương pháp bạn chọn có thể phụ thuộc vào kiểu bài toán bạn gặp phải.

Chèn dữ liệu mảng đỉnh là phương pháp phổ biến khác của tổ chức. Thay vì gọi sáu mảng khác nhau, mỗi lần gọi duy trì một kiểu dữ liệu khác nhau (màu sắc, pháp tuyến bề mặt, tọa độ, và tiếp tục như thế), bạn có thể có trộn các kiểu dữ liệu khác nhau thành một mảng đơn lẻ. (Xem "Chèn mảng" với hai phương pháp để giải quyết vấn đề này)

Bước 1: Kích hoạt mảng

Bước đầu tiên là gọi **glEnableClientState()** với một biến đếm, chúng kích hoạt mảng lựa chọn. Theo lý thuyết, bạn có thể cần gọi điều này sáu lần để kích hoạt sáu mảng có sẵn. Trong thực tế, bạn sẽ chỉ cần kích hoạt từ một đến bốn mảng. Ví dụ, không chắc rằng bạn sẽ kích hoạt cả hai GL_COLOR_ARRAY và GL_INDEX_ARRAY, do chế độ hiển thị trong chương trình của bạn hỗ trợ hoặc chế độ RGBA hoặc chế độ chỉ số màu, nhưng chắc chắn không đồng thời cả hai.

*void **glEnableClientState**(GLenum array)*

Chỉ ra mảng được phép. Các hằng số tượng trưng GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, và GL_EDGE_FLAG_ARRAY là các biến được chấp nhận.

Nếu bạn sử dụng ánh sáng, bạn có thể muốn định nghĩa một pháp tuyến bề mặt cho mọi đỉnh (Xem "vectơ pháp tuyến"). Để sử dụng mảng đỉnh cho trường hợp đó, bạn kích hoạt cả hai pháp tuyến bề mặt và mảng tọa độ đỉnh:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Giả sử rằng bạn muốn tắt nguồn sáng ở một điểm và chỉ vẽ hình học với việc sử dụng màu đơn lẻ. Bạn muốn gọi **glDisable()** để tắt trạng thái ánh sáng (xem Chương 5). Bây giờ nguồn sáng đã ngừng hoạt động, bạn cũng muốn dừng việc thay đổi các giá trị của trạng thái pháp tuyến bề mặt, điều này lãng phí công sức. Để làm điều này, bạn gọi:

```
glDisableClientState(GL_NORMAL_ARRAY);
```

*void **glDisableClientState**(GLenum array);*
*Chỉ ra những mảng không thể. Sử dụng hằng số tương tự như **glEnableClientState()**.*
Bạn có thể tự hỏi tại sao các nhà xây dựng OpenGL tạo ra các tên lệnh mới (và dài), **gl*ClientState()**. Tại sao bạn không thể chỉ cần gọi **glEnable()** và **glDisable()**? Một lí do đó là **glEnable()** và **glDisable()** có thể được lưu trữ trong một danh sách hiển thị, nhưng đặc tả của các mảng đỉnh thì không thể, do dữ liệu còn lại trên phần của máy khách.

Bước 2: Xác định dữ liệu cho các mảng

Có một cách dễ hiểu bằng một lệnh đơn chỉ ra một mảng đơn ở máy khách. Có sáu thủ tục khác nhau để chỉ ra các mảng- một thủ tục cho mỗi loại mảng. Cũng có một lệnh mà có thể chỉ ra một số mảng phía máy khách một lần, tất cả bắt nguồn từ một mảng chèn đơn lẻ.
*void **glVertexPointer**(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);*
Chỉ ra vị trí tọa độ không gian dữ liệu có thể được truy cập. pointer là địa chỉ bộ nhớ của tọa độ đầu tiên của đỉnh đầu tiên trong mảng. type chỉ ra kiểu dữ liệu GL_SHORT, GL_INT, GL_FLOAT, hay GL_DOUBLE) của mỗi tọa độ trong mảng. size là số lượng tọa độ trên một đỉnh, chúng phải là 2, 3, hay 4 stride là khối các byte giữa các đỉnh liên tiếp. Nếu stride là 0, các đỉnh được hiểu là gói trong mảng.
Để truy cập đến năm mảng khác, có năm thủ tục tương tự:
*void **glColorPointer**(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);*
*void **glIndexPointer**(GLenum type, GLsizei stride, const GLvoid *pointer);*
*void **glNormalPointer**(GLenum type, GLsizei stride, const GLvoid *pointer);*
*void **glTexCoordPointer**(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);*
*void **glEdgeFlagPointer**(GLsizei stride, const GLvoid *pointer);*
Sự khác biệt chính giữa các thủ tục cho dù là kích cỡ và kiểu là duy nhất hay phải được chỉ ra. Ví dụ, một pháp tuyến bề mặt luôn có ba thành phần, do đó nó là thừa để chỉ ra kích cỡ của nó. Một cờ hiệu của ạnh luôn là Boolean đơn lẻ, do đó không cần đề cập kích cỡ hay kiểu. Bảng 2-4 hiển thị các giá trị hợp lệ cho kích cỡ và kiểu dữ liệu.

Bảng 2-4 : Các kích cỡ mảng đỉnh (các giá trị trên đỉnh) và kiểu dữ liệu (liên tục)

Command	Sizes	Values for *type* Argument
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glIndexPointer	1	GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	no type argument (type of data must be GLboolean)

Ví dụ 2-9 sử dụng các mảng đỉnh cho cả màu RGBA và tọa độ đỉnh. các giá trị dấu chấm động RGB và tọa độ nguyên (x, y) tương ứng của chúng được nạp thành GL_COLOR_ARRAY và GL_VERTEX_ARRAY.
Ví dụ 2-9 : Kích hoạt và nạp các mảng đỉnh: varray.c

```
static GLint vertices[] = {25, 25,
100, 325,
175, 25,
175, 325,
250, 25,
325, 325};
static GLfloat colors[] = {1.0, 0.2, 0.2,
0.2, 0.2, 1.0,
0.8, 1.0, 0.2,
```

```
0.75, 0.75, 0.75,
0.35, 0.35, 0.35,
0.5, 0.5, 0.5};
glEnableClientState (GL_COLOR_ARRAY);
glEnableClientState (GL_VERTEX_ARRAY);
glColorPointer (3, GL_FLOAT, 0, colors);
glVertexPointer (2, GL_INT, 0, vertices);
```

Bước nhảy

Với một bước nhảy là không, mỗi kiểu của mảng đỉnh (màu RGB, chỉ số màu, tọa độ đỉnh và tiếp tục như thế) phải được đóng gói chặt. Dữ liệu trong mảng phải là đồng nhất; đó là dữ liệu phải là tất cả các giá trị màu RGB, tất cả các tọa độ đỉnh, hay tất cả những dữ liệu tương tự khác theo khuôn.

Việc sử dụng một bước nhảy lớn hơn không có thể có ích, đặc biệt khi liên quan đến các mảng chèn. Trong mảng sau đây của Glfloats, có sáu đỉnh. Với mỗi đỉnh, có ba giá trị màu RGB, chúng lần lượt là các tọa độ đỉnh (x, y, z)

```
static GLfloat intertwined[] =
{1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Bước nhảy cho phép một mảng đỉnh để truy cập dữ liệu mong muốn của nó định kỳ trong mảng. Ví dụ, để tham chiếu duy nhất các giá trị màu trong mảng intertwined, lời gọi sau đây bắt đầu từ đầu của mảng (Chúng cũng có thể được truyền là &intertwined[0])) và bước nhảy 6 * **sizeof**(GLfloat) bytes, chúng là kích cỡ của cả hai màu và các giá trị tọa độ đỉnh. Bước nhảy này là đủ để có được sự khởi đầu của dữ liệu cho đỉnh tiếp theo.

```
glColorPointer (3, GL_FLOAT, 6 * sizeof(GLfloat), intertwined);
```

Đối với con trỏ tọa độ đỉnh, bạn cần bắt đầu từ xa hơn nữa trong mảng, tại phần tử thứ tư của intertwined (nhớ rằng lập trình viên C bắt đầu chỉ số là 0).

```
glVertexPointer(3, GL_FLOAT,6*sizeof(GLfloat), &intertwined[3]);
```

Bước 3: Tham chiếu lại và tô vẽ

Cho đến khi các nội dung của các mảng đỉnh được tham chiếu lại, các mảng duy trì phía máy khách, và nội dung của chúng được thay đổi dễ dàng. Trong Bước 3, nội dung của các mảng được thu, gửi cho máy chủ, và sau đó gửi xuống quy trình xử lí đồ họa để tô vẽ.

Có ba cách để thu được dữ liệu: từ một phần tử mảng đơn (vị trí chỉ số), từ một chuỗi các phần tử mảng, và từ một danh sách có thứ tự các phần tử mảng.

Tham chiếu lại một phần tử mảng đơn.

*`void **glArrayElement**(GLint ith)*

*Thu được dữ liệu của một (thứ i)đỉnh với tất các mảng có thể hiện tại. Đối với mảng tọa độ đỉnh, lệnh tương ứng sẽ là **glVertex**[size][type]**v()**,với size là một giá trị trong [2, 3,4], và kiểu là một trong những kiểu [s, i, f, d] tương ứng với GLshort, GLint, GLfloat, và Gldouble.*

*Cả size và type đã được định nghĩa bởi **glVertexPointer().** Với các mảng kích hoạt khác, **glArrayElement()** gọi **glEdgeFlagv()**, **glTexCoord**[size][type]**v()**, **glColor**[size][type]**v()**, **glIndex**[type]**v()**, và **glNormal**[type]**v()**. Nếu mảng tọa độ đỉnh được kích hoạt, thủ tục **glVertex*v()** được chạy sau cùng, sau khi chạy (nếu được kích hoạt) thành năm giá trị mảng tương ứng.*

glArrayElement() luôn được gọi giữa **glBegin()** và **glEnd().** (Nếu lời gọi bên ngoài, **glArrayElement()** thiết lập trạng thái hiện tại cho tất cả các mảng kích hoạt, ngoại trừ với đỉnh, chúng không có trạng thái hiện tại). Trong Ví dụ 2-10, một tam giác được vẽ với việc sử dụng đỉnh thứ ba, thứ tư và thứ 6 từ các mảng đỉnh được kích hoạt (đôi khi, nhớ rằng lập trình C có vị trí mảng bắt đầu là không)

Ví dụ 2-10 : sử dụng glArrayElement() để định nghĩa các màu và các đỉnh

```
glEnableClientState (GL_COLOR_ARRAY);
```

```
glEnableClientState (GL_VERTEX_ARRAY);
glColorPointer (3, GL_FLOAT, 0, colors);
glVertexPointer (2, GL_INT, 0, vertices);
glBegin(GL_TRIANGLES);
glArrayElement (2);
glArrayElement (3);
glArrayElement (5);
glEnd();
```

Khi chạy, năm dòng lệnh cuối có tác dụng như

```
glBegin(GL_TRIANGLES);
glColor3fv(colors+(2*3*sizeof(GLfloat));
glVertex3fv(vertices+(2*2*sizeof(GLint));
glColor3fv(colors+(3*3*sizeof(GLfloat));
glVertex3fv(vertices+(3*2*sizeof(GLint));
glColor3fv(colors+(5*3*sizeof(GLfloat));
glVertex3fv(vertices+(5*2*sizeof(GLint));
glEnd();
```

Do **glArrayElement()** chỉ là một hàm đơn trên đỉnh, nó có thể giảm số lời gọi hàm, chúng tăng hiệu suất nói chung.

Cảnh báo rằng nếu nội dung của mảng được thay đổi giữa **glBegin()** và **glEnd()**, không có đảm bảo rằng bạn sẽ nhận dữ liệu ban đầu hay dữ liệu thay đổi với phần tử yêu cầu của bạn. Để an toàn, đừng thay đổi nội dung của phần tử mảng bất kì mà có thể truy cập cho đến khi hình học nguyên thủy được hoàn thiện.

Tham chiếu lại một danh sách của các phần tử mảng

glArrayElement() là phù hợp với mảng dữ liệu "dao động xung quanh" một cách ngẫu nhiên. Một thủ tục tương tự, **glDrawElements()** là thích hợp cho dao động xung quanh mảng dữ liệu của bạn trong một cách thức thứ tự hơn.

*void **glDrawElements**(GLenum mode, GLsizei count, GLenum type, void *indices);*

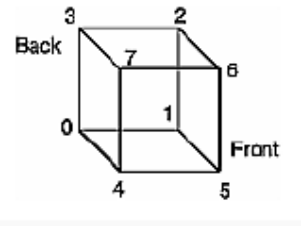
*Định nghĩa một chuỗi hình học nguyên thủy với việc sử dụng count phần tử, chỉ số của chúng được lưu trữ trong mảng indices. Type bắt buộc phải là một trong GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, hay GL_UNSIGNED_INT, việc chỉ ra kiểu dữ liệu của mảng indices. mode chỉ ra những kiểu nguyên thủy được xây dựng và là một trong các giá trị tương tự mà được chấp nhận bởi **glBegin()**; ví dụ, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, và cứ tiếp tục như thế.*

Tác dụng của **glDrawElements()** gần như tương tự trình tự lệnh này:

```
int i;
glBegin (mode);
for (i = 0; i < count; i++)
glArrayElement(indices[i]);
glEnd();
```

glDrawElements() bổ sung kiểm tra để chắc chắn mode, count, và type là hợp lệ. Ngoài ra, khác với chuỗi trước đó, việc chạy **glDrawElements()** tạo ra một số trạng thái không xác định. Sau khi chạy **glDrawElements()**, màu RGB hiện tại, chỉ số màu, tọa độ pháp tuyến, tọa độ kết cấu và cờ hiệu của cạnh là không xác định nếu mảng tương ứng đã được kích hoạt.

Với **glDrawElements(),** các đỉnh cho mỗi mặt của hình hộp lập phương có thể được đặc trong một mảng chỉ số. Ví dụ 2-11 chỉ ra hai cách để sử dụng **glDrawElements()** để tô vẽ hình hộp lập phương. Hình 2-15 chỉ ra số các đỉnh sử dụng trong Ví dụ 2-11.



Hình 2-15 : Hình hộp lập phương với các đỉnh được đánh số.

Ví dụ 2-11 : Hai cách để sử dụng glDrawElements()

```
static GLubyte frontIndices = {4, 5, 6, 7};
static GLubyte rightIndices = {1, 2, 6, 5};
static GLubyte bottomIndices = {0, 1, 5, 4};
static GLubyte backIndices = {0, 3, 2, 1};
static GLubyte leftIndices = {0, 4, 7, 3};
static GLubyte topIndices = {2, 3, 7, 6};
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE,frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE,rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

Hay tốt hơn, gộp tất cả các chỉ số lại với nhau:

```
static GLubyte allIndices = {4, 5, 6, 7, 1, 2, 6, 5,
0, 1, 5, 4, 0, 3, 2, 1,
0, 4, OpenGL   , kj, 3, 2, 3, 7, 6};
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

Chú ý: việc tóm gọm **glDrawElements()** giữa cặp **glBegin()**/**glEnd()** là một lỗi. Với cả hai **glArrayElement()** và **glDrawElements()**, cài đặt OpenGL của bạn nó cũng có thể lưu trữ các đỉnh đượ xử lí gần đây nhất, cho phép ứng dụng của bạn dùng chung hay dùng lại các đỉnh. Lấy hình hộp lập phương đề cập lúc trước, ví dụ, chúng có sáu mặt (đa giác) nhưng tám đỉnh. Mỗi đỉnh được sử dụng bởi đúng ba mặt. Bỏ qua **glArrayElement()** hay **glDrawElements()**, tô vẽ tất cả sáu mặt sẽ yêu cầu việc xử lí 24 đỉnh, mặc dù mười sáu đỉnh sẽ là thừa. Cài đặt OpenGL của bạn có thể giảm thiểu dư thừa và xử lí một số trong tám đỉnh. (Sử dụng lại các đỉnh có thể bị giới hạn cho tất cả các đỉnh trong một lời gọi đơn **glDrawElements()** hay với **glArrayElement()**, trong phạm vi một cặp **glBegin()**/**glEnd()**.)

Tham chiếu lại một chuỗi các phần tử mảng

Trong khi **glArrayElement()** và **glDrawElements()** "dao động xung quanh" mảng dữ liệu của bạn, **glDrawArrays()** đi thẳng qua chúng.

*void **glDrawArrays**(GLenum mode, GLint first, GLsizei count);*

*Xây dựng một chuỗi hình học nguyên thủy với việc sử dụng các phần tử mảng bắt đầu ở first và kết thúc tại first+count-1 của mỗi mảng kích hoạt. mode chỉ ra những kiểu nguyên thủy được xây dựng và là một trong những giá trị giống nhau được chấp nhận bằng **glBegin()**; ví dụ, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, và tiếp tục như thế.*

Tác động của **glDrawArrays()** hầu hết giống như chuỗi lệnh này

```
int i;
glBegin (mode);
for (i = 0; i < count; i++)
glArrayElement(first + i);
glEnd();
```

Như trường hợp với **glDrawElements()**, **glDrawArrays()** cũng thực hiện kiểm tra lỗi trên các giá trị biến của nó và để lại màu RGB hiện tại, chỉ số màu, tọa độ pháp tuyến, tọa độ kết cấu và cờ hiệu của cạnh với các giá trị không xác định nếu mảng tương ứng đã được kích hoạt.

Thử điều này

Thay đổi thủ tục vẽ khối mười hai mặt trong Ví dụ 2-13 để sử dụng các mảng đỉnh.

Các mảng chèn

Nâng cao

Phần trước trong chương này (trong "Bước nhảy"), trường hợp đặc biệt của các mảng chèn đã được khảo sát. Trong phần đó, mảng trộn, chúng chèn màu RGB và các tọa độ đỉnh 3D, đã được truy cập bằng lời gọi **glColorPointer()** và **glVertexPointer().** Sử dụng một cách cẩn thận trợ giúp bước nhảy chỉ ra chính xác các mảng.

```
static GLfloat intertwined[] =
{1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Cũng có một thủ tục lớn, **glInterleavedArrays(),** có thể chỉ ra một số mảng đỉnh mỗi lần. **glInterleavedArrays()** cũng kích hoạt và hủy các mảng tương ứng (do đó nó kết hợp cả Bước 1 và Bước 2). Mảng trộn điều chỉnh chính xác một trong mười bốn cấu hình dữ liệu chèn được hỗ trợ bởi **glInterleavedArrays().** Do đó để chỉ ra nội dung của mảng trộn thành màu RGB và các mảng đỉnh và kích hoạt cả hai mảng, gọi

```
glInterleavedArrays (GL_C3F_V3F, 0, intertwined);
```

Hàm này gọi tới **glInterleavedArrays()** để kích hoạt mảng GL_COLOR_ARRAY và GL_VERTEX_ARRAY. Nó tắt GL_INDEX_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_NORMAL_ARRAY, và GL_EDGE_FLAG_ARRAY.

Điều này cũng có tác động tượng tự như việc gọi **glColorPointer()** và **glVertexPointer()** để chỉ ra các giá trị 6 đỉnh thành mỗi mảng. Bây giờ bạn sẵn sàng cho Bước 3: việc gọi **glArrayElement()**, **glDrawElements()**, hay **glDrawArrays()** để tham chiếu lại mỗi phần tử mảng.

*void **glInterleavedArrays**(GLenum format, GLsizei stride, void *pointer)*

Khởi tạo tất cả sáu mảng, hủy đi những mảng mà không được xác định cú pháp, và kích hoạt những mảng mà được xác định. format là một trong 14 hằng số tượng trưng, chúng biểu diễn 14 cấu hình dữ liệu.; Bảng 2-5 hiển thị giá trị format, stride chỉ ra khối byte giữa các đỉnh liên tiếp nhau. Nếu stride là 0, các đỉnh được hiểu là các gói cô đọng trong mảng. pointer là địa chỉ bộ nhớ của tọa độ đầu tiên trong mảng.

Chú ý rằng **glInterleavedArrays()** không hỗ trợ các cờ hiệu.

Cơ chế của **glInterleavedArrays()** là khó hiểu và yêu cầu tham khảo đến Ví dụ 2-12 và Bảng 2-5. Trong ví dụ và bảng đó, bạn sẽ hiểu et, ec, và ec chúng là các giá trị logic cho tọa độ kết cấu, màu và mảng pháp tuyến được kích hoạt và hủy, và bạn sẽ thấy sắc thái, sc và sv, chúng là các kích cỡ (số các thành phần) cho tọa độ kết cấu, màu sắc và mảng đỉnh, toàn cục là kiểu dữ liệu cho màu RGBA, chúng là mảng duy nhất có thể có các giá trị chèn không có giá trị thực. pc,pn và pv được tính toán các bước nhảy để nhảy qua màu riêng lẻ, pháp tuyến và các giá trị đỉnh, và s là bước nhảy (nếu chúng không được chỉ ra bởi người dùng) để nhảy từ một phần tử mảng đến phần tử tiếp theo.

Tác dụng của **glInterleavedArrays()** tương tự như việc gọi chuỗi lệnh trong Ví dụ 2-12 với nhiều giá trị đã định nghĩa trong Bảng 2-5. Tấc cả trỏ số học được thực hiện trong đơn vị của **sizeof**(GL_UNSIGNED_BYTE).

Ví dụ 2-12. Tác động của glInterleavedArrays(format, stride, pointer)

```
int str;
/* thiết lập et, ec, en, st, sc, sv, tc, pc, pn, pv, và s
* là một hàm của Bảng 2-5 và giá trị của định dạng*/
str = stride;
if (str == 0)
```

```
str = s;
glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);
if (et) {
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(st, GL_FLOAT, str, pointer);
}
else
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (ec) {
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(sc, tc, str, pointer+pc);
}
else
glDisableClientState(GL_COLOR_ARRAY);
if (en) {
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, str, pointer+pn);
}
else
glDisableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT, str, pointer+pv);
```

Trong Bảng 2-5, T và F là True và False. f là **sizeof**(GL_FLOAT). c gấp bốn lần **sizeof**(GL_UNSIGNED_BYTE), làm tròn lên đến bội số sát nhất của f.

Bảng 2-5 : (tiếp tục) Các biến trực tiếp glInterleavedArrays()

format	et	ec	en	st	sc	sv	tc	pc	pn	pv	s
GL_V2F	F	F	F			2				0	2f
GL_V3F	F	F	F			3				0	3f
GL_C4UB_V2F	F	T	F		4	2	GL_UNSIGNED_BYTE	0		c	c+2f
GL_C4UB_V3F	F	T	F		4	3	GL_UNSIGNED_BYTE	0		c	c+3f
GL_C3F_V3F	F	T	F		3	3	GL_FLOAT	0		3f	6f
GL_N3F_V3F	F	F	T			3			0	3f	6f
GL_C4F_N3F_V3F	F	T	T		4	3	GL_FLOAT	0	4f	7f	10f
GL_T2F_V3F	T	F	F	2		3				2f	5f
GL_T4F_V4F	T	F	F	4		4				4f	8f

GL_T2F_C4UB_V3F	T	T	F	2	4	3	GL_UNSIGNED_BYTE	2f		c+2f	c+5f
GL_T2F_C3F_V3F	T	T	F	2	3	3	GL_FLOAT	2f		5f	8f
GL_T2F_N3F_V3F	T	F	T	2		3			2f	5f	8f
GL_T2F_C4F_N3F_V3F	T	T	T	2	4	3	GL_FLOAT	2f	6f	9f	12f
GL_T4F_C4F_N3F_V4F	T	T	T	4	4	4	GL_FLOAT	4f	8f	11f	15f

Bắt đầu với việc học định dạng đơn giản, GL_V2F, GL_V3F, và GL_C3F_V3F. Nếu bạn sử dụng định dạng bất kì với C4UB, bạn có thể phải sử dụng kiểu dữ liệu cấu trúc hay thực hiện một số kiểu thú vị và gợi ý toán học để đóng gói bốn unsigned byte thành một word 32 bit đơn.

Đối với một số cài đặt OpenGL, sử dụng mảng chèn có thể tăng hiệu suất ứng dụng. Với một mảng chèn, cần phải xác định chính xác sắp đặt dữ liệu của bạn. Bạn biết dữ liệu của bạn được đóng gói cô đặc và có thể được truy cập trong một vùng. Nếu các mảng chèn không được sử dụng, thông tin bước nhảy và kích cỡ phải được khảo sát để dò xem dữ liệu có được đóng gói cô đặc hay không.

Chú ý: glInterleavedArrays() chỉ kích hoạt và hủy các mange đỉnh và xác định các giá trị đối với dữ liệu mảng đỉnh. Nó không tô vẽ bất cứ thứ gì. Bạn vẫn phải hoàn thiện Bước 3 và gọi **glArrayElement()**, **glDrawElements()**, hay **glDrawArrays()** để tham chiếu lại các con trỏ và tô vẽ đồ họa.

Các nhóm thuộc tính

Trong "Quản lí trạng thái cơ bản" bạn đã hiểu làm thế nào để thiết lập hay truy vấn một trạng thái riêng lẻ hay biến trạng thái. Thế đấy, bạn cũng có thể lưu trữ và phục hồi các giá trị của một tập hợp hay biến trạng thái liên quan với một lệnh đơn lẻ.

OpenGL nhóm các biến trạng thái liên quan thành một nhóm thuộc tính. Ví dụ, thuộc tính GL_LINE_BIT bao gồm năm biến trạng thái: độ rộng đường thẳng, GL_LINE_STIPPLE kích hoạt trạng thái, mẫu nét đứt đường thẳng, đếm số lặp lại các đường thẳng nét đứt và GL_LINE_SMOOTH kích hoạt trạng thái. (Xem "khử răng cưa" trong Chương 6) Với các lệnh **glPushAttrib()** và **glPopAttrib(),** bạn có thể lưu trữ và khôi phục tất cả năm biến trạng thái, tất cả trong cùng một lần.

Một số biến trạng thái có hơn một nhóm thuộc tính. Ví dụ, biến trạng thái, GL_CULL_FACE, là một phần của cả hai đa giác và nhóm thuộc tính được kích hoạt.

Trong phiên bản OpenGL 1.1, hiện nay có hai ngăn xếp thuộc tính khác nhau. Ngoài ra ngăn xếp thuộc tính gốc (chúng lưu các giá trị của các biến trạng thái máy chủ), cũng có một ngăn xếp thuộc tính máy khách, có thể truy cập bằng các lệnh **glPushClientAttrib()** và **glPopClientAttrib()**.

Nói chung, việc sử dụng các lệnh này là nhanh hơn tự bạn tìm, lưu trữ và khôi phục các giá trị. Một số giá trị có thể được duy trì trong phần cứng, và việc tìm chúng có thể tốn kém. Ngoài ra, nếu bạn đang thao tác trên máy khách từ xa, tất cả thuộc tính dữ liệu phải được truyền đi trên mạng kết nối và truyền về khi nó đã được thu, đã lưu trữ và đã khôi phục. Tuy nhiên, cài đặt OpenGL của bạn duy trì ngăn xếp thuộc tính trên máy chủ, tránh việc trễ mạng không cần thiết.

Có 20 nhóm thuộc tính khác nhau, chúng có thể được lưu trữ và khôi phục bằng **glPushAttrib()** và **glPopAttrib()**.. Có hai nhóm thuộc tính máy khách, chúng có thể được lưu trữ và khôi phục bằng **glPushClientAttrib()** và **glPopClientAttrib()**. Đối với cả hai máy chủ và khách, các thuộc tính được lưu trữ trên một ngăn xếp, chúng lưu trữ ít nhất 16 nhóm thuộc tính. (Thực tế độ sâu ngăn xếp cho cài đặt của bạn có thể thu được bằng việc sử dụng GL_MAX_ATTRIB_STACK_DEPTH và GL_MAX_CLIENT_ATTRIB_STACK_DEPTH với **glGetIntegerv()**.) Việc đẩy vào một ngăn xếp đầy hay lấy ra trong một ngăn xếp rỗng tạo nên một lỗi. (Xem các bảng trong Mục lục B để tìm ra chính xác những thuộc tính được lưu cho các giá trị mặt nạ cụ thể; đó là, những thuộc tính ở trong một nhóm thuộc tính cụ thể)

*void **glPushAttrib**(GLbitfield mask);*
*void **glPopAttrib**(void);*
***glPushAttrib()** lưu tất cả các thuộc tính đã xác định bởi các bít trong mặt nạ bằng việc đẩy chúng vào ngăn xếp thuộc tính. **glPopAttrib()** phục hồi các giá trị của trạng thái đó mà đã được lưu với **glPushAttrib()** cuối cùng. Bảng 2-7 liệt kê mặt nạ bít có thể mà có thể là OR logic với nhau để lưu mọi kết hợp của các thuộc tính. Mỗi bit tương ứng một tập hợp của các biến trạng thái riêng lẻ. Ví dụ, GL_LIGHTING_BIT có nghĩa là tất cả các biến trạng thái liên quan đến ánh sáng, chúng bao gồm màu, ánh sáng bao quanh, khuếch tán, phản chiếu và phát ra của chất liệu hiện tại. một danh sách ánh sáng được kích hoạt, và hướng về một nơi. Khi **glPopAttrib()** được gọi, tất cả các biến được khôi phục.*
Mặt nạ đặc biệt, GL_ALL_ATTRIB_BITS, được sử dụng để lưu và khôi phục tất cả các biến trạng thái trong mọi nhóm thuộc tính.

Mask Bit	Attribute Group
GL_ACCUM_BUFFER_BIT	accum-buffer
GL_ALL_ATTRIB_BITS	--
GL_COLOR_BUFFER_BIT	color-buffer
GL_CURRENT_BIT	current
GL_DEPTH_BUFFER_BIT	depth-buffer
GL_ENABLE_BIT	enable
GL_EVAL_BIT	eval
GL_FOG_BIT	fog
GL_HINT_BIT	hint
GL_LIGHTING_BIT	lighting
GL_LINE_BIT	line
GL_LIST_BIT	list
GL_PIXEL_MODE_BIT	pixel
GL_POINT_BIT	point
GL_POLYGON_BIT	polygon
GL_POLYGON_STIPPLE_BIT	polygon-stipple
GL_SCISSOR_BIT	scissor
GL_STENCIL_BUFFER_BIT	stencil-buffer
GL_TEXTURE_BIT	texture
GL_TRANSFORM_BIT	transform
GL_VIEWPORT_BIT	viewport

*void **glPushClientAttrib**(GLbitfield mask);*
*void **glPopClientAttrib**(void);*
***glPushClientAttrib()** lưu tất cả các thuộc tính chỉ ra bởi các bit trong mặt nạ bằng việc đẩy chúng lên trên ngăn xếp thuộc tính máy khách. **glPopClientAttrib()** khôi phục các giá trị các biến trạng thái của chúng mà đã được lưu với **glPushClientAttrib()** cuối cùng. Bảng 2-7 liệt kê mặt nạ bít hợp lí mà có thể là phép OR logic với nhau để lưu mọi kết hợp của các thuộc tính máy khách.*
Có hai nhóm thuộc tính máy khách, phản hồi và truyền ngược, chúng không thể được lưu hay khôi phục với cơ chế ngăn xếp.
Bảng2-7 : Nhóm thuộc tính máy khách

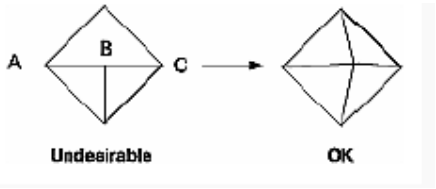
Mask Bit	Attribute Group
GL_CLIENT_PIXEL_STORE_BIT	pixel-store
GL_CLIENT_VERTEX_ARRAY_BIT	vertex-array
GL_ALL_CLIENT_ATTRIB_BITS	--
can't be pushed or popped	feedback
can't be pushed or popped	select

Một số gợi ý cho việc xây dựng mô hình đa giác của bề mặt.

Theo một số kĩ thuật mà bạn có thể muốn sử dụng như xây dựng đa giác xấp xỉ bề mặt. Bạn có thể muốn xem lại phần này sau khi đọc Chương 5 về ánh sáng và Chương 7 về danh sách hiển thị. Điều kiện ánh sáng ảnh hưởng mô hình như thế nào mỗi khi chúng được vẽ, và một số kĩ thuật sau đây sẽ hiệu quả hơn nhiều khi sử dụng cùng với danh sách hiển thị. Khi bạn đọc các kĩ thuật này, nhớ rằng khi tính toán ánh sáng được kích hoạt, vectơ pháp tuyến phải được chỉ ra để thu được kết quả tốt hơn.

Việc xây dựng đa giác xấp xỉ bề mặt là một nghệ thuật, và chủ yếu dựa trên kinh nghiệm. Phần này, tuy nhiên, liệt kê một số điểm mà có thể làm cho chúng dễ dàng hơn một chút khi bắt đầu.

- Giữ hướng đa giác nhất quán. Chắc chắn rằng khi được quan sát từ bên ngoài, tất cả các đa giác trên bề mặt có cùng hướng (tất cả theo chiều kim đồng hồ hoặc tất cả ngược chiều kim đồng hồ). Hướng nhất quán quan trọng để lọc đa giác và hai mặt ánh sáng. Cố gắng làm đúng điều này ngay lúc đầu, vì nó sẽ rất mất công sức để chỉnh sửa nếu sau đó có vấn đề. (Nếu bạn sử dụng **glScale*()** để phản xạ hình học quanh một số trục đối xứng, bạn có thể thay đổi hướng với **glFrontFace()** để duy trì hướng nhất quán.)
- Khi bạn chia nhỏ một bề mặt, chú ý với mọi đa giác không có dạng tam giác. Ba đỉnh của một tam giác sẽ đảm bảo việc nằm trên một mặt phẳng; mọi đa giác với bốn đỉnh hay hơn có thể không đảm bảo điều này. Đa giác không phẳng có thể được quan sát từ một số hướng sao cho các cạnh giao nhau và OpenGL có thể không tô vẽ những đa giác đó một cách chính xác.
- Luôn có một thỏa hiệp giữa hiển thị tốc độ và chất lượng ảnh. Nếu bạn chia nhỏ một bề mặt thành số lượng nhỏ các đa giác nó tô vẽ nhanh hơn nhưng có thể có bề mặt răng cưa; nếu bạn chia nó thành hàng triệu đa giác nhỏ, nó trông đẹp hơn nhưng có thể mất nhiều thời gian để tô vẽ. Để lý tưởng, bạn có thể cung cấp một biến đến thủ tục chia nhỏ mà xác định bạn muốn chia nhỏ với độ mịn như thế nào, và nếu đối tượng là ở xa so với mắt, bạn có thể sử dụng tỉ lệ chia thô hơn. Ngoài ra, khi bạn chia nhỏ, sử dụng những đa giác lớn ở vị trí bề mặt là khá bằng phẳng, và đa giác nhỏ ở vùng độ cong lớn.
- Với ảnh chất lượng cao, một ý kiến hay là chia nhỏ trên các cạnh tô bóng hơn là phía bên trong. Nếu bề mặt được quay liên quan đến mắt, điều này không dễ dàng thực hiện, do các cạnh tô bóng đang di chuyển. Các cạnh tô bóng xuất hiện ở vị trí vectơ pháp tuyến vuông góc với veco từ bề mặt đến điểm quan sát- tức là, khi tích vô hướng vectơ của chúng là 0. Thuật toán chia nhỏ của bạn có thể lựa chọn để chia nhỏ hơn nếu tích vô hướng này xấp xỉ 0.
- Cố gắng tránh giao điểm hình chữ T trong mô hình của bạn (xem Hình 2-16). Như đã thấy, không có gì đảm bảo rằng đoạn thẳng AB và BC nằm chính xác trên đoạn thẳng AC. Đôi khi chúng như thế, và đôi khi thì không, phụ thuộc vào phép biến đổi và hướng.
  Điều này có thể dẫn đến phá vỡ sự xuất hiện không liên tục trên bề mặt.



Hình 2-16 : Việc sửa đổi một giao điểm hình chữ T không mong muốn.

Nếu bạn đang xây dựng một bề mặt khép kín, hay chắc chắn sử dụng số giống nhau hoàn toàn trong tọa độ đầu và cuối của một vòng lặp khép kín, hay bạn có thể lấy các khoảng trống và phá vỡ do việc làm tròn số. Đây là một ví dụ hai chiều của một đoạn mã tồi:

```
/* không sử dụng mã này */
#define PI 3.14159265
#define EDGES 30
/* vẽ một đường tròn */
glBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
glEnd();
```

Các cạnh nối với nhau chính xác chỉ khi máy của bạn quản lí để tính toán sin và cos 0 và của (2*PI*EDGES/EDGES) và lấy các giá trị giống nhau hoàn toàn. Nếu bạn tin tưởng đơn vị dấu chấm động trên máy của bạn làm đúng điều này, có nghĩa bạn đã bị lừa… Để chỉnh đúng đoạn mã, chắc chắn rằng khi i == EDGES, bạn sử dụng 0 cho sin và cos, không phải là 2*PI*EDGES/EDGES. (Hay vẫn đơn giản, sử dụng GL_LINE_LOOP thay cho GL_LINE_STRIP, va thay đổi điều kiện kết thúc vòng lặp i < EDGES.)

Ví dụ: Xây dựng một khối 20 mặt

Để minh họa một số điều kiện mà tăng xấp xỉ bề mặt, hay quan sát một số ví dụ chuỗi lệnh. Đoạn mã này quan tâm các đỉnh của khối 20 mặt thông thường (chúng là một Plato đặc bao gồm 20 mặt mà nối 20 đỉnh, mỗi mặt của chúng là một tam giác đều). Một khối 20 mặt có thể được xem là một xấp xỉ thô của một hình cầu.

Ví dụ 2-13 định nghĩa các đỉnh và tam giác tạo thành một khối 20 mặt và sau đó vẽ khối.

Ví dụ 2-13 : Vẽ một khối 20 mặt

```
#define X .525731112119133606
#define Z .850650808352039932
static GLfloat vdata[12][3] = {
{-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
{0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
{Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
static GLuint tindices[20][3] = {
{0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
{8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
{7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
{6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
int i;
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
/* color information here */
glVertex3fv(&vdata[tindices[i][0]][0]);
glVertex3fv(&vdata[tindices[i][1]][0]);
glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

Các số chưa biết X và Z được lựa chọn sao cho khoảng cách từ gốc đến mọi đỉnh của khối là 1.0. Tọa độ của 20 đỉnh được gửi vào trong mảng vdata[][], nơi mà đỉnh thứ 0 là {- &Xgr; , 0.0, &Zgr; }, thứ nhất là {X, 0.0, Z}, và tiếp tục như thế. Mảng tindices[][] chỉ ra các đỉnh liên kết như thế nào để tạo thành các tam giác. Ví dụ, tam giác đầu tiên được tạo từ đỉnh thứ 0, thứ tư và thứ nhất. Nếu bạn lấy các đỉnh cho tam giác theo thứ tự đã cho, tất cả các tam giác có cùng hướng.

Dòng lệnh mà đề cập thông tin màu sẽ được thay thế bằng một lệnh thiết lập màu của bề mặt thứ i. Nếu không có mã nào viết ở đây, tất cả các bề mặt được vẽ cùng một màu, và nó sẽ không thể thấy không gian 3 chiều của đối tượng. Một lựa chọn để xác định chính xác màu là định nghĩa pháp tuyến bề mặt và sử dụng ánh sáng, như mô tả trong phần tiếp theo.

Chú ý: trong mọi ví dụ mô tả ở phần này, trừ khi bề mặt được vẽ duy nhất một lần, bạn chắc chắn lưu các đỉnh tính toán và tọa độ pháp tuyến sao cho tính toán không phải lặp lại mỗi khi bề mặt được vẽ. Điều này có thể được thực hiện với việc sử dụng cấu trúc dữ liệu riêng của bạnvhay bằng việc xây dựng danh sách hiển thị. (Xem Chương 7)

Việc tính toán vectơ pháp tuyến cho một bề mặt

Nếu một bề mặt được chiếu sáng, bạn cần cung cấp vectơ pháp tuyến đến bề mặt. Việc tính toán tích có hướng của hai vectơ trên bề mặt đó cung cấp vectơ pháp tuyến. Với những bề mặt phẳng của một khối 20 mặt, mọi định nghĩa ba đỉnh của một bề mặt có cùng vectơ pháp tuyến. Trong trường hợp này, pháp tuyến cần được xác định duy nhất một lần cho mỗi bộ ba đỉnh. Mã trog Ví dụ 2-14 có thể thay thế "thông tin màu ở đây" nằm trong Ví dụ 2-13 cho việc vẽ khối 20 mặt.

Ví dụ 2-14 : Tạo các vectơ pháp tuyến cho một một bề mặt

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

Hàm **normcrossprod()** tính tích có hướng của hai vectơ, như chỉ trong Ví dụ 2-15.

Ví dụ 2-15 : Tính toán tích có hướng của hai vectơ

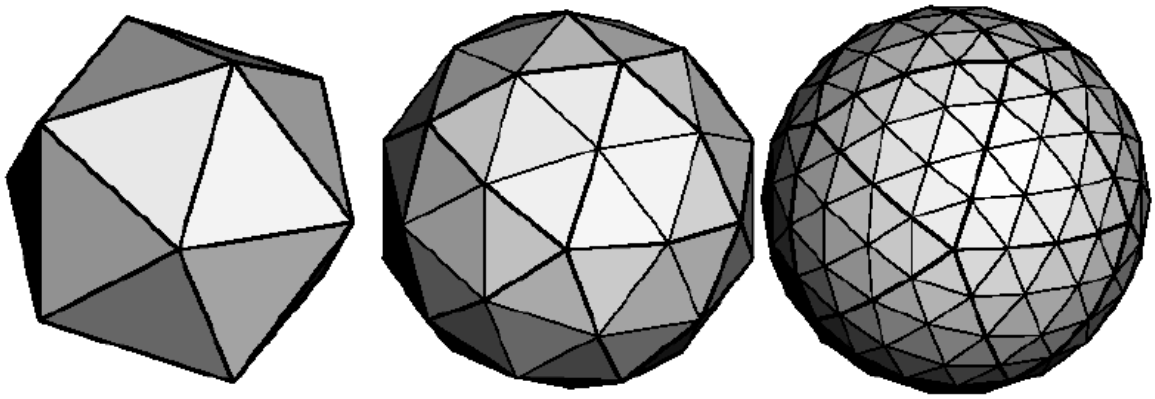
```
void normalize(float v[3]) {
GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
if (d == 0.0) {
error("zero length vector");
return;
}
v[0] /= d; v[1] /= d; v[2] /= d;
}
void normcrossprod(float v1[3], float v2[3], float out[3])
{
GLint i, j;
GLfloat length;
out[0] = v1[1]*v2[2] - v1[2]*v2[1];
out[1] = v1[2]*v2[0] - v1[0]*v2[2];
out[2] = v1[0]*v2[1] - v1[1]*v2[0];
normalize(out);
}
```

Nếu bạn đang sử dụng một khối 20 mặt như một xấp xỉ cho một hình cầu chiếu sáng, bạn sẽ muốn sử dụng các vectơ pháp tuyến mà vuông góc với bề mặt chính xác của hình cầu, hơn là vuống góc với các bề mặt. Đối với một hình cầu, các vectơ pháp tuyến là đơn giản; mỗi điểm ở cùng hướng là vectơ từ gốc đến điểm tương ứng. Do dữ liệu đỉnh khối 20 mặt cho một khối 20 mặt bán kính là 1, dữ liệu pháp tuyến và đỉnh là như nhau. Đây là mã vẽ một khối 20 mặt xấp xỉ của hình cầu được chiếu sáng mịn. (giả sử ánh sáng được kích hoạt, như được mô tả trong Chương 5)

```
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
glNormal3fv(&vdata[tindices[i][0]][0]);
glVertex3fv(&vdata[tindices[i][0]][0]);
glNormal3fv(&vdata[tindices[i][1]][0]);
glVertex3fv(&vdata[tindices[i][1]][0]);
glNormal3fv(&vdata[tindices[i][2]][0]);
glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

Hoàn thiện mô hình

Một hình cầu với 20 bề mặt xấp xỉ không được đẹp trừ khi ảnh của hình cầu trên màn hình khá nhỏ, nhưng có một cách dễ dàng để tăng độ chính xác của xấp xỉ. Tưởng tượng khối 20 mặt vẽ nối tiếp trong một hình cầu, và chia nhỏ các tam giác được chỉ ra trong Hình 2-17. Các đỉnh được đưa vào sau nằm co bên trong hình cầu, nên đẩy chúng đến bề mặt bằng chuẩn hóa chúng (việc chia chúng cho một thừa số để độ dài của chúng là 1). Xử lí phép chia này có thể được lặp lại với độ chính xác tùy ý. Ba đối tượng được chỉ ra trong Hình 2-17 sử dụng 20, 80, và 320 các tam giác xấp xỉ, một cách tương ứng.



Hình 2-17 : Việc chia để hoàn thiện một đa giác xấp xỉ thành một bề mặt.

Ví dụ 2-16 thực hiện một phép chia đơn, việc tạo một xấp xỉ hình cầu 80 cạnh.

Ví dụ 2-16 : Phép chia đơn

```
void drawtriangle(float *v1, float *v2, float *v3)
{
glBegin(GL_TRIANGLES);
glNormal3fv(v1); vlVertex3fv(v1);
glNormal3fv(v2); vlVertex3fv(v2);
glNormal3fv(v3); vlVertex3fv(v3);
glEnd();
}
void subdivide(float *v1, float *v2, float *v3)
{
GLfloat v12[3], v23[3], v31[3];
GLint i;
for (i = 0; i < 3; i++) {
v12[i] = v1[i]+v2[i];
v23[i] = v2[i]+v3[i];
v31[i] = v3[i]+v1[i];
}
normalize(v12);
normalize(v23);
normalize(v31);
```

```
drawtriangle(v1, v12, v31);
drawtriangle(v2, v23, v12);
drawtriangle(v3, v31, v23);
drawtriangle(v12, v23, v31);
}
for (i = 0; i < 20; i++) {
subdivide(&vdata[tindices[i][0]][0],
&vdata[tindices[i][1]][0],
&vdata[tindices[i][2]][0]);
}
```

Ví dụ 2-17 là một sửa đổi nhỏ của Ví dụ 2-16, chúng chia đệ quy các tam giác cho độ sâu thích hợp. Nếu giá trị độ sâu là 0, không có phép chia nào được thực hiện, và tam giác được vẽ là hiện tại. nếu độ sâu là 1, một phép chia đơn được thực hiện, và cứ tiếp tục như thế.

Ví dụ 2-17 : Phép chia đệ quy

```
void subdivide(float *v1, float *v2, float *v3, long depth)
{
GLfloat v12[3], v23[3], v31[3];
GLint i;
if (depth == 0) {
drawtriangle(v1, v2, v3);
return;
}
for (i = 0; i < 3; i++) {
v12[i] = v1[i]+v2[i];
v23[i] = v2[i]+v3[i];
v31[i] = v3[i]+v1[i];
}
normalize(v12);
normalize(v23);
normalize(v31);
subdivide(v1, v12, v31, depth-1);
subdivide(v2, v23, v12, depth-1);
subdivide(v3, v31, v23, depth-1);
subdivide(v12, v23, v31, depth-1);
}
```

Phép chia tổng quát

Một kĩ thuật phép chia đệ quy như được mô tả trong Ví dụ 2-17 có thể được sử dụng cho những kiểu khác của bề mặt. Thông thường, đệ quy kết thúc khi hoặc chắc chắn đạt tới một độ sâu hoặc một số điều kiện của độ cong được thỏa mãn (phần độ cong lớn của bề mặt trông đẹp hơn khi được chia nhỏ hơn).

Để thấy giải pháp tổng quát cho bài toán chia nhỏ, xét bề mặt tham số tùy ý với hai biến u[0] và u[1].

Giả sử rằng có hai thủ tục được cấp:

```
void surf(GLfloat u[2], GLfloat vertex[3], Glfloat normal[3]);
float curv(GLfloat u[2]);
```

Nếu **surf()** được truyền u[], đỉnh ba chiều tương ứng và các vectơ pháp tuyến (với độ dài là 1) được trả về. Nếu u[] được truyền tới **curv()**, độ cong của bề mặt tại điểm đó được tính toán và trả về. (Xem giới thiệu sách giáo khoa về hình học vi phân để có thêm thông tin về đánh giá độ cong bề mặt)

Ví dụ 2-18 chỉ ra thủ tục đệ quy mà chia nhỏ một tam giác cho đến khi hoặc đạt được độ sâu lớn nhất hoặc độ cong cực đại tại ba đỉnh nhỏ hơn một vài ngưỡng.

Ví dụ 2-18 : Phép chia tổng quát

```
void subdivide(float u1[2], float u2[2], float u3[2],
float cutoff, long depth)
{
GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
GLfloat u12[2], u23[2], u32[2];
GLint i;
if (depth == maxdepth || (curv(u1) < cutoff &&
curv(u2) < cutoff && curv(u3) < cutoff)) {
surf(u1, v1, n1); surf(u2, v2, n2); surf(u3, v3, n3);
glBegin(GL_POLYGON);
glNormal3fv(n1); glVertex3fv(v1);
glNormal3fv(n2); glVertex3fv(v2);
glNormal3fv(n3); glVertex3fv(v3);
glEnd();
return;
}
for (i = 0; i < 2; i++) {
u12[i] = (u1[i] + u2[i])/2.0;
u23[i] = (u2[i] + u3[i])/2.0;
u31[i] = (u3[i] + u1[i])/2.0;
}
subdivide(u1, u12, u31, cutoff, depth+1);
subdivide(u2, u23, u12, cutoff, depth+1);
subdivide(u3, u31, u23, cutoff, depth+1);
subdivide(u12, u23, u31, cutoff, depth+1);
}
```