# LỜI NÓI ĐẦU

*Đồ họa máy tính là một trong những lĩnh vực lí thú nhất và phát triển nhanh nhất của tin học. Ngay từ khi xuất hiện, đồ họa máy tính đã có sức lôi cuốn mãnh liệt, cuốn hút rất nhiều người và được sử dụng ở nhiều lĩnh vực khác nhau như : khoa học, nghệ thuật, kinh doanh, thương mại, công nghiệp, quản lí, giáo dục, giải trí, ... Số lượng các chương trình đồ họa ứng dụng thật khổng lồ và phát triển liên tục.*

*Cuốn sách này được biên soạn dựa trên đề cương môn Đồ họa máy tính thuộc chương trình đào tạo tin học bậc cử nhân và cao đẳng của Bộ Giáo dục và Đào tạo, tập trung vào các vấn đề của đồ họa hai chiều và ba chiều nhằm cung cấp một nền tảng kiến thức đầy đủ và chọn lọc bao gồm các khái niệm cơ bản nhất, các thuật toán cơ sở của đồ họa máy tính, ... giúp người đọc có thể tự tìm hiểu và xây dựng các chương trình ứng dụng đồ họa.*

*Cuốn sách được chia làm 10 chương, gồm hai phần chính : đồ họa hai chiều và đồ họa ba chiều. Cuối mỗi chương đều có phần tóm tắt và hệ thống bài tập để người đọc tự kiểm tra. Các thuật toán trình bày đều có lưu đồ và chương trình minh họa dưới dạng ngôn ngữ C.*

*Để các vấn đề trình bày được phong phú, đa dạng và cập nhật, chúng tôi đã rất nỗ lực trong việc tham khảo các tài liệu kinh điển, đặc biệt là các bài giảng về đồ họa của các trường đại học nổi tiếng trên thế giới ở Âu, Mỹ như Brown, Stanford, MIT, Waterloo, ... Tuy nhiên trong quá trình biên soạn chắc chắn không thể không tránh khỏi sơ sót, chúng tôi xin trân trọng tiếp thu tất cả những ý kiến đóng góp của bạn đọc cũng như các bạn đồng nghiệp để hoàn thiện cuốn sách ngày một tốt hơn.*

*Chúng tôi xin chân thành cảm ơn Ban chủ nhiệm Khoa Công nghệ Thông tin - Đại học Khoa học Tự nhiên, các anh chị trong Ban biên tập Nhà xuất bản Giáo dục đã hỗ trợ rất nhiệt tình để cuốn sách này sớm đến tay bạn đọc.*

**CÁC TÁC GIẢ**

***CHƯƠNG 1***

# GIỚI THIỆU VỀ ĐỒ HỌA MÁY TÍNH

Sự phát triển của khoa học, kĩ thuật, nghệ thuật, kinh doanh, và công nghệ luôn luôn phụ thuộc vào khả năng truyền đạt thông tin của chúng ta, hoặc thông qua các bit dữ liệu lưu trữ trong microchip hoặc thông qua giao tiếp bằng tiếng nói. Câu châm ngôn từ xa xưa "một hình ảnh có giá trị hơn cả vạn lời" hay "trăm nghe không bằng một thấy" cho thấy ý nghĩa rất lớn của hình ảnh trong việc chuyển tải thông tin. Hình ảnh bao giờ cũng được cảm nhận nhanh và dễ dàng hơn, đặc biệt là trong trường hợp bất đồng về ngôn ngữ. Do đó không có gì ngạc nhiên khi mà ngay từ khi xuất hiện máy tính, các nhà nghiên cứu đã cố gắng sử dụng nó để phát sinh các ảnh trên màn hình. Trong suốt gần 50 năm phát triển của máy tính, khả năng phát sinh hình ảnh bằng máy tính của chúng ta đã đạt tới mức mà bây giờ hầu như tất cả các máy tính đều có khả năng đồ họa.

Đồ họa máy tính là một trong những lĩnh vực lí thú nhất và phát triển nhanh nhất của tin học. Ngay từ khi xuất hiện, đồ họa máy tính đã có sức lôi cuốn mãnh liệt, cuốn hút rất nhiều người ở nhiều lĩnh vực khác nhau như khoa học, nghệ thuật, kinh doanh, quản lí, ... Tính hấp dẫn và đa dạng của đồ họa máy tính có thể được minh họa rất trực quan thông qua việc khảo sát các ứng dụng của nó.

## 1. MỘT SỐ ỨNG DỤNG CỦA ĐỒ HỌA MÁY TÍNH

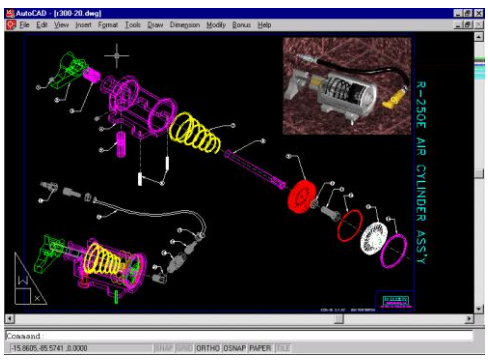
Ngày nay, đồ họa máy tính được sử dụng trong rất nhiều lĩnh vực khác nhau như công nghiệp, thương mại, quản lí, giáo dục, giải trí, … Số lượng các chương trình đồ họa ứng dụng thật khổng lồ và phát triển liên tục, sau đây là một số ứng dụng tiêu biểu :

### 1.1. Hỗ trợ thiết kế

Một trong những ứng dụng lớn nhất của đồ họa máy tính là hỗ trợ thiết kế (CAD – computer-aided design). Ngày nay CAD đã được sử dụng hầu hết trong việc thiết kế các cao ốc, ô tô, máy bay, tàu thủy, tàu vũ trụ, máy tính, trang trí mẫu vải, và rất nhiều sản phẩm khác.

Sử dụng các chương trình này, đầu tiên các đối tượng được hiển thị dưới dạng các phác thảo của phần khung (wireframe outline), mà từ đó có thể thấy được toàn bộ hình dạng và các thành phần bên trong của các đối tượng. Sử dụng kĩ thuật này, người thiết kế sẽ dễ dàng nhận thấy ngay các thay đổi của đối tượng khi tiến hành hiệu chỉnh các chi tiết hay thay đổi góc nhìn, ….

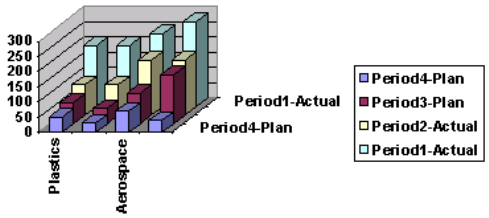
Một khi đã thiết kế xong phần khung của đối tượng, các mô hình chiếu sáng, tô màu và tạo bóng bề mặt sẽ được kết hợp để tạo ra kết quả cuối cùng rất gần với thế giới thực .




### 1.2. Biểu diễn thông tin

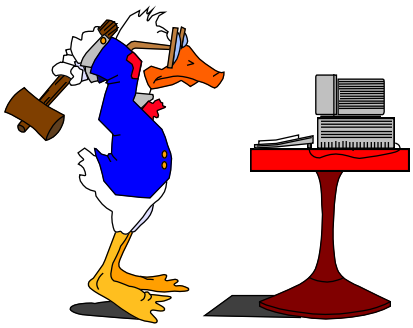
Đây là các ứng dụng sử dụng đồ họa máy tính để phát sinh các biểu đồ, đồ thị, … dùng minh họa mối quan hệ giữa nhiều đối tượng với nhau. Các ứng dụng này thường được dùng để tóm lược các dữ liệu về tài chính, thống kê, kinh tế, khoa học, toán học, … giúp cho việc nghiên cứu, quản lí, … một cách có hiệu quả.

**Hình 1.1** - Phác thảo phần khung và kết quả của thiết kế xy lanh




**Hình 1.2** – Thông tin tóm lược được biểu diễn qua các biểu đồ

### 1.3. Lĩnh vực giải trí, nghệ thuật



Trong lĩnh vực nghệ thuật, các chương trình máy tính như Paint Shop Pro, Adobe Photoshop, 3D Studio, … hỗ trợ rất đắc lực cho các họa sĩ, các nhà tạo mẫu trong việc thiết kế các hình ảnh sống động, và rất thực. Với các chương trình này, người họa sĩ được máy tính tạo cho cảm giác y như đang làm việc ngoài đời thực bằng cách cung cấp các công cụ như khung vẽ, giá vẽ, bảng pha màu, các hiệu ứng ba chiều, … làm cho họ cảm thấy rất thoải mái và tiện lợi.

Ngoài ra đồ họa máy tính còn giúp tạo ra các chương trình trò chơi, giải trí; hỗ trợ cho các kĩ xảo điện ảnh, cho các nhà làm phim. Có nhiều bộ phim rất nổi tiếng nhờ vào kĩ xảo điện ảnh như : Công viên Khủng long kỉ Jura (Jurassic Park), Titanic, Thế giới nước (Water World), …

**Hình 1.3** – Hình ảnh được tạo ra từ chương trình đồ họa

### 1.4. Giáo dục và đào tạo

Hiện nay các chương trình mô phỏng cấu trúc của các vật thể, tiến trình của các phản ứng hóa học, hoạt động của các gói tin trên mạng máy tính, … được dùng rất nhiều trong việc hỗ trợ giảng dạy.

Trong đào tạo, các ứng dụng mô phỏng được dùng để kiểm tra trình độ người lái, huấn luyện phi công, điều khiển giao thông, …

**Hình 1.4** – Chương trình học về máy tính

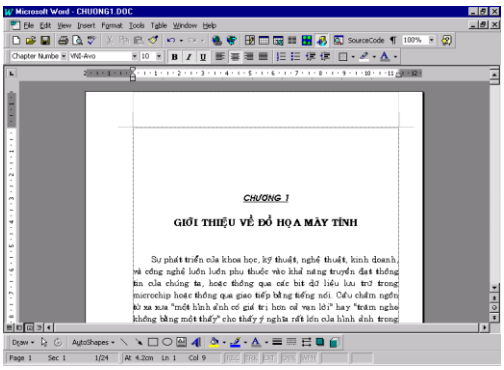
### 1.5. Giao tiếp giữa máy tính và người dùng

Mọi ứng dụng đều phải có giao diện giao tiếp với người dùng. Giao diện đồ họa thực sự là một cuộc cách mạng mang lại sự thuận tiện và thoải mái cho người dùng ứng dụng. Các ứng dụng dựa trên hệ điều hành MS Windows là một minh họa rất trực quan của giao diện đồ họa. Các chức năng của các ứng dụng này được thiết kế cho người dùng làm việc thông qua các biểu tượng mô tả chức năng đó. Ví dụ, chức năng lưu tập tin được hiểu thông qua biểu tượng đĩa mềm, chức năng in ấn được hiểu thông qua biểu tượng máy in, … Để chọn các chức năng, người dùng sử dụng chuột trỏ đến và nhấn vào các biểu tượng tương ứng. Điểm thuận lợi chính khi dùng biểu tượng là kích thước không gian mà nó chiếm ít hơn nhiều so với dùng văn bản để mô tả cho cùng một chức năng, ngoài ra việc nắm bắt các chức năng qua các biểu tượng sẽ dễ dàng hơn rất nhiều khi người dùng gặp trở ngại về mặt ngôn ngữ.

Các ứng dụng có giao diện đồ họa còn cho phép người dùng khả năng làm việc dễ dàng với nhiều cửa sổ với nhiều dạng tài liệu khác nhau cùng một lúc.

**Hình 1.5** – Giao diện của chương trình MS Word

## 2. KHÁI NIỆM VỀ ĐỒ HỌA MÁY TÍNH




Đồ họa máy tính là tất cả những gì liên quan đến việc sử dụng máy tính để phát sinh ra hình ảnh. Các vấn đề liên quan tới công việc này bao gồm : tạo, lưu trữ, thao tác trên các mô hình (các mô tả hình học của đối tượng) và các ảnh.

Theo định nghĩa này thì đồ họa máy tính bao gồm việc thiết kế phần cứng như thiết bị hiển thị, các thuật toán cần thiết để phát sinh các đường trên các thiết bị này, các phần mềm được sử dụng cho cả người lập trình hệ thống và người lập trình ứng dụng đồ họa, và các chương trình ứng dụng tạo ảnh bằng máy tính.

Đồ họa máy tính cung cấp một trong những phương cách tự nhiên nhất cho việc truyền đạt thông tin với máy tính. Ngày nay, trong nhiều quá trình

thiết kế, cài đặt và xây dựng, thông tin mà hình ảnh mang lại là hầu như không thể thiếu được. Kĩ thuật trực quan (scientific visualization) đã trở nên là một lĩnh vực rất quan trọng từ năm 1980, khi các nhà nghiên cứu khoa học và các kĩ sư nhận ra rằng họ không thể xử lí một lượng dữ liệu khổng lồ phát sinh từ các siêu máy tính mà dữ liệu không được tóm lược và làm nổi bật các xu hướng và hiện tượng qua nhiều loại biểu diễn đồ họa khác nhau.

Đồ họa máy tính tương tác là một trong những phương tiện mang lại thêm nhiều sự thuận lợi cho người dùng trong việc phát sinh hình ảnh kể từ khi có phát minh của máy ảnh và truyền hình. Với máy tính, chúng ta có thể tạo các hình ảnh không chỉ của các đối tượng cụ thể, thực tế, mà còn của các đối tượng trừu tượng, nhân tạo; các biểu diễn của dữ liệu mà không có tính kế thừa về mặt hình học, như là kết quả điều tra, khảo sát. Hơn nữa, với đồ họa máy tính chúng ta không bị giới hạn trong các ảnh tĩnh. Các ảnh động thông thường mang lại nhiều hiệu quả hơn so với ảnh tĩnh, đặc biệt là với các hiện tượng biến đổi theo thời gian, cả thực tế (như sự đổi hướng của cánh máy bay siêu âm, hay sự phát triển của khuôn mặt người từ lúc trẻ thơ tới lúc già) và trừu tượng (như là xu hướng phát triển của việc sử dụng năng lượng, gia tăng dân số, …).

Có nhiều cách tiếp cận trong việc học môn đồ họa, trải rộng từ việc nghiên cứu phần cứng tới việc học để sử dụng đồ họa máy tính chỉ trong một lĩnh vực chuyên biệt nào đó như là thiết kế mạch tích hợp cao (VLSI – very large scale integrated circuit). Ở đây chúng ta tiếp cận từ góc độ của người lập trình ứng dụng, đó là người sử dụng tất cả các hỗ trợ của phần cứng, các công cụ phần mềm để xây dựng nên các ứng dụng.

Tuy nhiên để có thể thiết kế và cài đặt các chương trình ứng dụng đồ họa được tốt, ngoài việc tìm hiểu các khả năng của công cụ lập trình, chúng ta cũng cần phải nắm vững các khái niệm về phần cứng; các vấn đề, các nguyên lí liên quan đến cài đặt phần mềm, các thuật toán, các ứng dụng, …

## 3. TỔNG QUAN VỀ MỘT HỆ ĐỒ HỌA

Một hệ đồ họa bao giờ cũng có hai thành phần chính đó là phần cứng và phần mềm. Phần cứng bao gồm các thiết bị hiển thị và nhập dữ liệu, … Phần mềm bao gồm các công cụ lập trình và các trình ứng dụng đồ họa. Chúng ta sẽ lần lượt khảo sát các thành phần này.

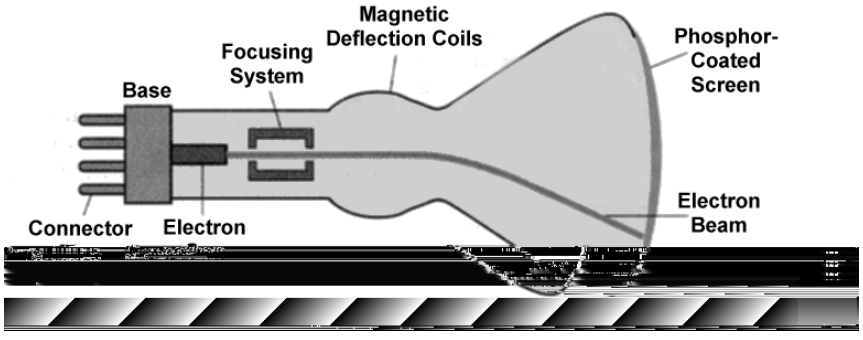
### 3.1. Phần cứng

#### 3.1.1. Thiết bị hiển thị

Màn hình là thiết bị hiển thị thông dụng nhất trong một hệ đồ họa. Các thao tác của hầu hết màn hình đều dựa trên thiết kế của ống tia âm cực (CRT – cathode ray tube).

***Cấu tạo của CRT***

Hình 1.6 minh họa thao tác cơ sở của một ống tia âm cực. Một chùm các tia điện tử (tia âm cực) phát ra từ một súng điện tử, vượt qua các hệ thống hội tụ (focusing) và dẫn hướng (deflection) sẽ hướng tới các vị trí xác định trên màn hình được phủ một lớp phosphor. Tại mỗi vị trí tương tác với tia điện tử, hạt phosphor sẽ phát ra một chấm sáng nhỏ. Vì ánh sáng phát ra bởi các hạt phosphor mờ dần rất nhanh nên cần phải có một cách nào đó để duy trì ảnh trên màn hình. Một trong các cách đó là lặp đi lặp lại nhiều lần việc vẽ lại ảnh thật nhanh bằng cách hướng các tia điện tử trở lại vị trí cũ. Kiểu hiển thị này gọi là refresh CRT.




**Hình 1.6** – Cấu tạo của CRT

Có nhiều loại phosphor được dùng trong một CRT. Ngoài màu sắc ra, điểm khác nhau chính giữa các loại phosphor là "độ bền" (persistent), đó là khoảng thời gian phát sáng sau khi tia CRT không còn tác động. Lớp phosphor có độ bền thấp cần tốc độ làm tươi cao hơn để giữ cho hình ảnh trên màn hình khỏi nhòe. Loại này thường rất tốt cho hoạt hình, rất cần thay đổi hình ảnh liên tục. Lớp phosphor có độ bền cao thường được dùng cho việc hiển thị các ảnh tĩnh, độ phức tạp cao. Mặc dù một số loại phosphor có độ bền lớn hơn 1 giây, tuy nhiên các màn hình đồ họa thường được xây dựng với độ bền dao động từ 10 đến 60 micro giây.

Số lượng tối đa các điểm có thể hiển thị trên một CRT được gọi là độ phân giải (resolution). Một định nghĩa chính xác hơn của độ phân giải là số lượng các điểm trên một centimet mà có thể được vẽ theo chiều ngang và chiều dọc, mặc dù nó thường được xem như là tổng số điểm theo mỗi hướng.

Kích thước vật lí của màn hình đồ họa được tính từ độ dài của đường chéo màn hình, thường dao động từ 12 đến 27 inch hoặc lớn hơn. Một màn hình CRT có thể được kết hợp với nhiều loại máy khác nhau, do đó số lượng các điểm trên màn hình có thể được vẽ thật sự còn tùy thuộc vào khả năng của hệ thống mà nó kết hợp vào.

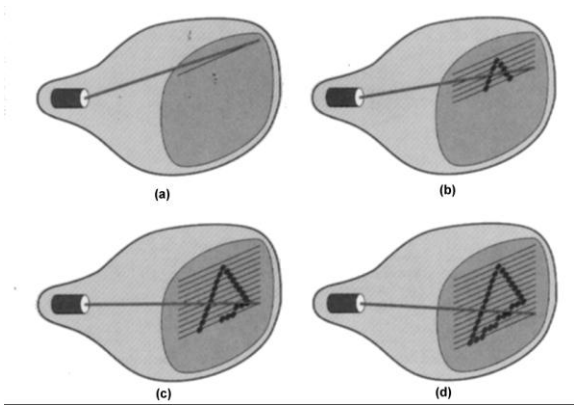
Một thuộc tính khác của màn hình nữa là tỉ số phương (aspect ratio). Tỉ số phương là tỉ lệ của các điểm dọc và các điểm ngang cần để phát sinh các đoạn thẳng có độ dài đơn vị theo cả hai hướng trên màn hình (trong một số trường hợp người ta thường dùng tỉ số phương như là tỉ số của các điểm theo chiều ngang so với các điểm theo chiều dọc). Với các màn hình có tỉ số phương khác 1, dễ dàng nhận thấy là các hình vuông hiển thị trên nó sẽ có dạng hình chữ nhật, các hình tròn sẽ có dạng hình ellipse. Thực ra khái niệm tỉ số phương xuất phát từ bản chất khoảng cách (nếu tính cùng một đơn vị độ dài) giữa các điểm dọc không bằng khoảng cách giữa các điểm ngang. Một tỉ số phương có giá trị ¾ có nghĩa là vẽ 3 điểm theo chiều dọc sẽ có cùng độ dài với việc vẽ 4 điểm theo chiều ngang.

***Màn hình dạng điểm (raster - scan display):***

Màn hình dạng điểm là dạng thường gặp nhất trong số các dạng màn hình sử dụng CRT dựa trên công nghệ truyền hình.

Trong hệ thống này, chùm tia điện tử sẽ được quét ngang qua màn hình, mỗi lần một dòng và quét tuần tự từ trên xuống dưới. Sự bật tắt của các điểm sáng trên màn hình phụ thuộc vào cường độ của tia điện tử và đây chính là cơ sở của việc tạo ra hình ảnh trên màn hình.

Mỗi điểm trên màn hình được gọi là một pixel hay là pel (viết tắt của picture element). Các thông tin về hình ảnh hiển thị trên màn hình được lưu trữ trong một vùng bộ nhớ gọi là vùng đệm làm tươi (refresh buffer) hay là vùng đệm khung (frame buffer). Vùng bộ nhớ này lưu trữ tập các giá trị cường độ sáng của toàn bộ các điểm trên màn hình và luôn luôn tồn tại một song ánh giữa mỗi điểm trên màn hình và mỗi phần tử trong vùng này.

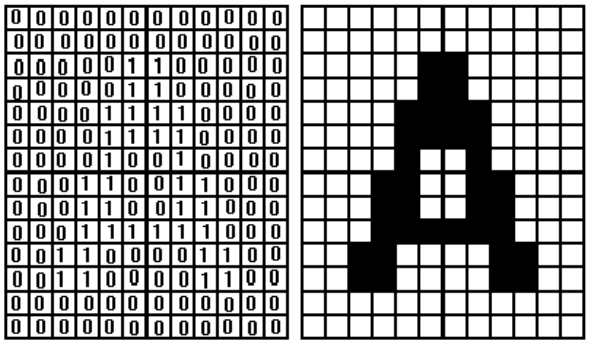



**Hình 1.7** – Quá trình tạo hình ảnh của các tia quét

Để thay đổi các hình ảnh cần hiển thị, các giá trị tương ứng với vị trí và độ sáng phải được đặt vào vùng đệm khung. Hình 1.8 minh họa các giá trị tương ứng trong vùng đệm khung để hiển thị hình ảnh của chữ A trên màn hình.

Đối với màn hình đen trắng, vùng đệm khung còn được gọi là bitmap, với các màn hình khác vùng đệm khung thường được gọi là pixmap.

Để tạo ra các ảnh đen trắng, đơn giản chỉ cần lưu thông tin của mỗi pixel bằng 1 bit (các giá trị 0, 1 sẽ tượng trưng cho việc tắt (tối), bật (sáng) pixel trên màn hình). Trong trường hợp ảnh nhiều màu, người ta cần nhiều bit hơn, nếu thông tin của mỗi pixel được lưu bằng b bit, thì ta có thể có $2^b$ giá trị màu phân biệt cho pixel đó.



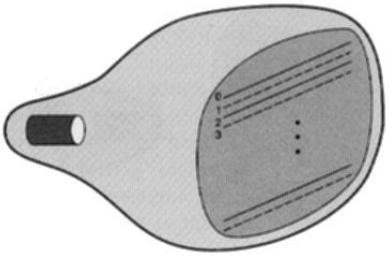
**Hình 1.8** – Song ánh giữa vùng đệm khung và màn hình

Trong các màn hình màu, người ta định nghĩa tập các màu làm việc trong một bảng tra (LookUp Table - LUT). Mỗi phần tử của LUT định nghĩa một bộ ba giá trị R (Red), G (Green), B (Blue) mô tả một màu nào đó. Khi cần sử dụng một màu, ta chỉ cần chỉ định số thứ tự (index) tương ứng của màu đó trong LUT. Bảng LUT có thể được thay đổi bởi các ứng dụng và người lập trình có thể can thiệp điều khiển. Với cách làm này chúng ta có thể tiết kiệm không gian lưu trữ cho mỗi phần tử trong vùng đệm khung.

Số phần tử của LUT được xác định từ số lượng các bits/pixel. Nếu mỗi phần tử của vùng đệm khung dùng b bits để lưu thông tin của một pixel, thì bảng LUT có $2^b$ phần tử. Nếu b=8, LUT sẽ có $2^8$=256 phần tử, đó chính là số màu có thể được hiển thị cùng một lúc trên màn hình.

Việc làm tươi trên màn hình dạng này được thực hiện ở tốc độ 60 đến 80 frame/giây. Đôi khi tốc độ làm tươi còn được biểu diễn bằng đơn vị Hertz (Hz – số chu kì/ giây), trong đó một chu kì tương ứng với một frame. Sử dụng đơn vị này, chúng ta có thể mô tả tốc độ làm tươi 60 frame/giây đơn giản là 60Hz. Khi đạt đến cuối mỗi dòng quét, tia điện tử quay trở lại bên trái của màn hình để bắt đầu dòng quét kế tiếp. Việc quay trở lại phía trái màn hình sau khi làm tươi mỗi dòng quét được gọi là tia hồi ngang (horizontal retrace). Và tới cuối mỗi frame, tia điện tử (tia hồi dọc – vertical retrace) quay trở lại góc trên bên trái của màn hình để chuẩn bị bắt đầu frame kế tiếp.

Trong một số màn hình, mỗi frame được hiển thị thành hai giai đoạn sử dụng kĩ thuật làm tươi đan xen nhau (interlaced refesh). Ở giai đoạn đầu tiên, tia quét sẽ quét một số dòng từ trên xuống dưới, sau tia hồi dọc, các dòng còn lại sẽ được quét. Việc đan xen các dòng quét này cho phép chúng ta thấy được toàn màn hình hiển thị chỉ trong một nửa thời gian so với dùng để quét tất cả các dòng một lần từ trên xuống dưới. Kĩ thuật này thường được dùng cho loại màn hình có tốc độ làm tươi thấp.



**Hình 1.9** – Hoạt động của màn hình interlaced

***Các hệ màu***

Việc nghiên cứu màu sắc bao gồm nhiều lĩnh vực như : quang học, sinh lí học, tâm lí học và các nhân tố khác thuộc về con người. Vì thế, có rất nhiều quan niệm cũng như các thành ngữ về khoa học các màu sắc. Đối với những người làm tin học, vấn đề mà họ quan tâm là mối tương tác qua lại giữa sự cảm nhận màu sắc của con người với các bộ phận phần cứng hiển thị màu sắc của màn hình máy tính, và với các phần mềm thiết kế trên nó. Bảng dưới đây sẽ trình bày mối quan hệ này :

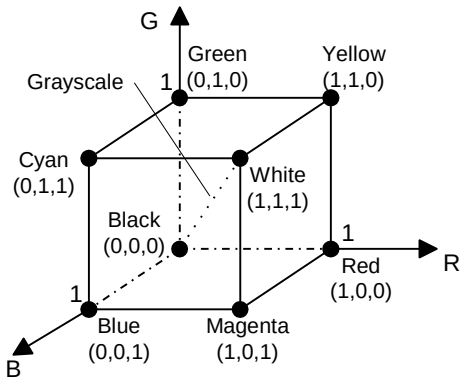
| **Sự cảm nhận của con người** | **Đặc điểm phần cứng** | **Đặc điểm phần mềm** |
|---|---|---|
| Màu sắc | Các màu hiển thị gốc | Thuật toán trên không gian màu |
| Sắc độ màu (Hue) | Bước sóng (WaveLength) | |
| Độ bão hòa (Saturation) | Sự thuần nhất của màu | |
| Độ sáng hay độ chói | Cường độ sáng | Hiệu chỉnh gamma |
| Sự "rung" của màn hình | Tốc độ làm tươi (refresh) | |

Không gian màu (color space) do đó được đưa ra để định các màu hiển thị trên máy tính bởi vì chúng làm đơn giản hóa các thao tác tính toán cần thiết cho việc chuyển đổi màu sắc (color transformation). Không gian màu có thể được thiết kế hoặc là dựa trên cơ sở của bộ phát sinh màu của phần cứng (hardware color generation) (ví dụ như không gian RGB) hoặc là dựa trên sự cảm nhận màu sắc của mắt (như không gian HSL). Với một ứng dụng, việc chọn không gian màu nào để sử dụng tùy thuộc vào một số nhân tố sau : độ chính xác mà các nhà thiết kế cần kiểm soát màu sắc (color control); yêu cầu về sự tương tác giữa các màu sắc và tốc độ các tính toán cho ứng dụng đó.

***Không gian RGB (RGB space)***

Không gian RGB mô tả màu sắc bằng ba thành phần **R**ed, **G**reen, **B**lue. Không gian này được minh họa bằng một khối lập phương với các trục chính R, G, B.

Mỗi màu trong không gian RGB đều được biểu diễn như là một vector thông qua ba vector cơ sở là Red, Green, Blue. Do đó, ứng với các tổ hợp khác nhau của ba màu này sẽ cho ta một màu mới.




**Hình 1.10** - Mô hình không gian RGB

Trong hình lập phương mỗi màu gốc (Red, Green, Blue) được đặt vào góc đối diện với các màu bù nó. (Hai màu bù nhau là hai màu mà khi kết hợp tạo thành màu trắng hay xám (grey)). Như vậy Red đối diện với Cyan, Green đối diện với Magenta, Blue đối diện với Yellow. Giá trị xám nằm trên đường chéo nối các đỉnh $(0,0,0),(1,1,1)$ của hình lập phương. Thường thường các trục R, G, B được chuẩn hóa. Khi kết hợp hai màu lại với nhau thì màu sinh ra có vector bằng tổng các vector thành phần.

*Một số thuận lợi khi dùng không gian RGB :*

- Không gian RGB là chuẩn công nghiệp cho các thao tác đồ họa máy tính. Các thao tác màu sắc có thể được tính toán trên các không gian màu khác nhưng cuối cùng cần phải chuyển về không gian RGB để có thể hiển thị trên màn hình (do thiết kế của phần cứng dựa trên mô hình RGB).
- Có thể chuyển đổi qua lại giữa không gian RGB với các không gian màu khác như CIE, CMY, HSL, HSV, ...
- Các thao tác tính toán trên không gian RGB thường đơn giản hơn.

*Một số bất lợi :*

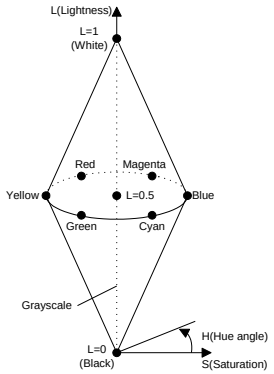
- Các giá trị RGB của một màu là khác nhau đối với các màn hình khác nhau : Nghĩa là các giá trị RGB của màu tim trên màn hình màu này sẽ không sinh ra đúng màu đó trên một màn hình khác.
- Sự mô tả các màu trong thế giới thực đối với không gian RGB còn nhiều hạn chế bởi vì không gian RGB không hoàn toàn phù hợp với sự cảm nhận màu sắc của con người. Hai điểm phân biệt trong không gian RGB, với mắt người có thể hoặc không thể là thể hiện của hai màu khác nhau. Chính vì điều này mà không gian RGB không thể ánh xạ trực tiếp đến bất cứ chiều cảm nhận nào khác (như hue, saturation, lightness) ngoài hue (sắc độ).

***Không gian HSL***

Không gian này có chú trọng hơn không gian RGB đến các thành phần của sự cảm nhận màu sắc của mắt (**H**ue, **S**aturation, **L**ightness). Tuy nhiên, không gian HSL thực ra cũng chỉ là một phép biến đổi gần đúng của không gian RGB mà thôi. Không giống như các không gian màu khác xây dựng trên sự cảm nhận màu sắc của mắt, không gian HSL vẫn còn bị lệ

thuộc vào phần cứng của CRT.

Không gian HSL được biểu diễn trong hệ tọa độ trụ, hình minh họa là hai hình nón úp vào nhau. H (Hue) là toạ độ ứng với góc quay, S (Saturation) là tọa độ gốc, L là trục thẳng đứng. Hầu hết các màu đạt bão hòa khi S = 1 và L = 0.5.




**Hình 1.11** - Mô hình không gian HSL

*Một số thuận lợi của không gian HSL :*

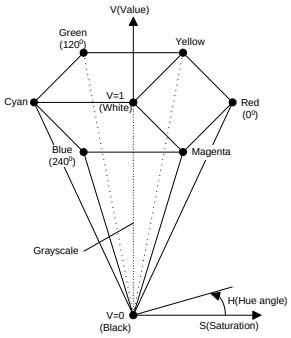
- Không gian HSL gần với sự cảm nhận các thuộc tính màu sắc của con người hơn không gian RGB (tuy cách tiếp cận đã đơn giản hóa đi nhiều). Các màu được xác định dễ dàng hơn chẳng hạn do H quay quanh trục đứng nên các màu bù được xác định một cách dễ dàng, đối với các giá trị lightness cũng vậy.
- Việc kiểm soát các màu cơ sở HSL dễ hơn cho những người mới làm quen với các chương trình đồ họa.

*Một số bất lợi :*

- Việc thêm vào một vector không thể thực hiện đơn giản như không gian RGB (chỉ thêm vào các thành phần màu). Các thao tác lượng giác khi biến đổi sẽ ảnh hưởng đáng kể đến tốc độ của chương trình.
- Cần phải qua hiệu chỉnh gamma trước khi hiển thị (giống như các không gian khác).

***Không gian HSV***

Không gian HSV thực chất cũng chỉ là một sự biến đổi khác của không gian RGB. Không gian HSV được mô hình bằng hình lập phương RGB quay trên đỉnh Black của nó. **H** (Hue) là góc quay quanh trục Values, **S** (Saturation) đi từ 0 đến 1, trục **V** (Values) do vậy tương ứng với đường chéo nối đỉnh White và Black.




**Hình 1.12** - Mô hình không gian HSV

Theo cách này, các màu đạt bão hòa khi S=1 và V=1. Trong không gian HSV các màu được chuẩn hóa về số các gam (gamut) màu của thiết bị hiển thị.

*Một số thuận lợi của không gian HSV :*

- Không gian HSV dễ dàng đáp ứng các màu sắc của các chương trình đồ họa do được xây dựng dựa trên sự bắt chước luật trộn màu của người họa sĩ. Ví dụ : Khi cần thêm màu trắng vào, phải đặt V=S=1 sau đó giảm S từ từ cho tới khi đạt được màu vừa ý; hay khi cần thêm màu đen vào, điều đó có nghĩa là giảm V (cường độ sáng) và cố định S,...
- Do không cần sử dụng các phép biến đổi lượng giác khi muốn chuyển sang không gian RGB nên không gian HSV có nhiều thuận lợi về mặt tính toán hơn so với không gian HSL.

*Một số bất lợi :*

- Cần có các phép hiệu chỉnh gamma.

***Bảng so sánh giữa các không gian màu***

| RGB | HSL | HSV |
|---|---|---|
| Chuẩn công nghiệp cho các thao tác đồ họa máy tính | Hình thức biến đổi khác của không gian RGB | Hình thức biến đổi khác của không gian RGB |
| Liên hệ trực tiếp với phần cứng | Liên hệ gần hơn với sự cảm nhận màu sắc của con người | Liên hệ gần hơn với sự cảm nhận màu sắc của con người |
| Là chuyển đổi cuối cùng cho tất cả các nhu cầu hiển thị | Đòi hỏi các phép biến đổi phức tạp | Đã đơn giản hóa các thao tác tính toán. |
| Không thể chuyển sang màn hình khác (phụ thuộc thiết bị) | Độc lập thiết bị | Độc lập thiết bị |
| Không có sự tương ứng 1-1 với cách cảm nhận màu của con người | Có | Có |
| Mô hình là hình lập phương | Mô hình là hai hình nón úp vào nhau | Mô hình là hình nón đơn |
| Được chuẩn hóa về 1 | Được chuẩn hóa về 1 | Được chuẩn hóa về 1 |
| Độ bão hòa đạt max khi S =1 | Độ bão hòa đạt max khi S =1, L =0.5 | Độ bão hòa đạt max khi S =1, V =1 |
| Trộn màu không rõ ràng | Rõ ràng | Rõ ràng |

### 3.1.2. Các thiết bị nhập

***Bàn phím :*** Xuất hiện trong hầu hết các máy tính, nó là thiết bị để nhập dữ liệu dạng văn bản và số. Đây là loại thiết bị quen thuộc nhất với người sử dụng tuy có hạn chế là tương tác không cao.

***Chuột :*** Cùng với sự xuất hiện của các ứng dụng đồ họa tương tác cao, chuột là thiết bị nhập ngày càng quen thuộc với người sử dụng. Người ta dùng chuột để trỏ và chọn (point-click) các chức năng phù hợp với yêu cầu của mình. Bằng cách này, giao tiếp giữa người dùng và máy tính càng ngày càng thân thiện và dễ dàng hơn. Ngoài ra chúng ta cũng có một số thiết bị nhập khác cùng họ với chuột như track ball, …

## 3.2. Phần mềm

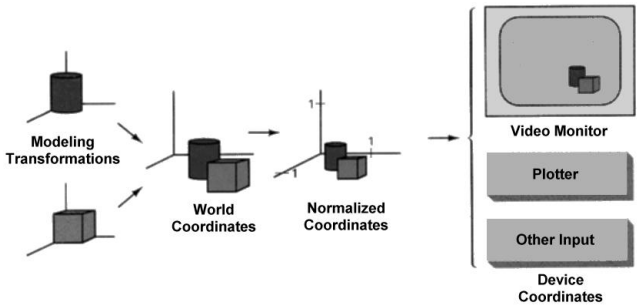
Phần mềm đồ họa có thể phân thành 2 loại : các công cụ lập trình và các trình ứng dụng đồ họa phục vụ cho một mục đích nào đó. Các công cụ lập trình cung cấp một tập các hàm đồ họa có thể được dùng trong các ngôn ngữ lập trình cấp cao như C, Pascal, .. Ví dụ như các thư viện đồ họa của các ngôn ngữ như C, Pascal hay GL (Graphics Library) của Silicon Graphics. Các hàm cơ sở của nó bao gồm việc tạo các đối tượng cơ sở của hình ảnh như đoạn thẳng, đa giác, đường tròn, …, thay đổi màu sắc, chọn khung nhìn, áp dụng các phép biến đổi, …. Trong khi đó, các ứng dụng đồ họa được thiết kế cho những người dùng không phải là lập trình viên, cho phép người dùng tạo các đối tượng, hình ảnh, … mà không cần quan tâm tới việc chúng được tạo ra như thế nào. Ví dụ như là Photoshop, AutoCAD, …

***Biểu diễn tọa độ***

Thông thường các hệ đồ họa sử dụng hệ tọa độ Descartes để mô tả đối tượng. Nếu các tọa độ của đối tượng được mô tả trong các hệ tọa độ khác như tọa độ cầu, …, chúng phải được chuyển về tọa độ Descartes trước khi dùng.

***Quy trình hiển thị đối tượng***

Trước tiên chúng ta mô tả các đối tượng thành phần của một ảnh phức tạp trong các hệ tọa độ riêng để thuận tiện cho việc biểu diễn tọa độ của chúng. Các hệ tọa độ này được gọi là hệ tọa độ mô hình (modeling coordinates) hay còn gọi là hệ tọa độ cục bộ (local coordinates). Một khi các đối tượng thành phần được biểu diễn xong, chúng ta sẽ đặt chúng vào các vị trí tương ứng trong ảnh sử dụng hệ tọa độ thế giới thực (world coordinates). Sau cùng, các mô tả của ảnh trong hệ tọa độ thế giới thực sẽ được chuyển đến một hoặc nhiều hệ tọa độ khác nhau của thiết bị hiển thị, tùy vào chúng ta muốn hiển thị trên thiết bị nào. Các hệ tọa độ này còn được gọi là hệ tọa độ thiết bị (device coordinates). Các mô tả trong các hệ tọa độ cục bộ và hệ tọa độ thế giới thực cho phép chúng ta sử dụng thứ nguyên thích hợp cho các đơn vị đo mà không phải bị ràng buộc gì của từng thiết bị hiển thị cụ thể.




**Hình 1.13** – Quy trình hiển thị đối tượng

Thông thường, các hệ đồ họa chuyển các mô tả trong hệ tọa độ thế giới thực tới hệ tọa độ thiết bị chuẩn (normalized device coordinates) có các chiều là đơn vị trước khi chuyển tới hệ tọa độ thiết bị. Điều này làm cho hệ thống độc lập với nhiều loại thiết bị khác nhau.

***Các hàm đồ họa***

Các hàm đồ họa cung cấp khả năng tạo và thao tác hình ảnh. Các hàm này được phân loại như sau :

- Tập các công cụ tạo ra các đối tượng đồ họa cơ sở như điểm, đoạn thẳng, đường cong, vùng tô, kí tự, …
- Tập các công cụ thay đổi thuộc tính dùng để thay đổi thuộc tính của các đối tượng đồ họa cơ sở như màu sắc, kiểu đường, kiểu chữ, mẫu tô, …
- Tập các công cụ thực hiện các phép biến đổi hình học dùng để thay đổi kích thước vị trí, hướng của các đối tượng, …
- Tập các công cụ biến đổi hệ quan sát dùng để xác định vị trí quan sát đối tượng và vị trí trên thiết bị hiển thị được dùng để hiển thị đối tượng.
- Tập các công cụ nhập liệu : Các ứng dụng đồ họa có thể sử dụng nhiều loại thiết bị nhập khác nhau như bút vẽ, bảng, chuột, bàn phím,… để điều khiển và xử lí dòng dữ liệu nhập.
- Cuối cùng là tập các công cụ chứa các thao tác dùng cho việc quản lí và điều khiển ví dụ như xóa toàn bộ màn hình, thiết lập chế độ đồ họa, …

***Các chuẩn phần mềm***

Mục tiêu căn bản của các phần mềm đồ họa được chuẩn là tính tương thích. Khi các công cụ được thiết kế với các hàm đồ họa chuẩn, phần mềm có thể được di chuyển một cách dễ dàng từ hệ phần cứng này sang hệ phần cứng khác và được dùng trong nhiều cài đặt và ứng dụng khác nhau.

Sau những nỗ lực không nhỏ của các tổ chức chuẩn hóa của các quốc gia và quốc tế, một chuẩn cho việc phát triển các phần mềm đồ họa đã ra đời đó là GKS (Graphics Kernel System – Hệ đồ họa cơ sở). Hệ thống này ban đầu được thiết kế cho tập các công cụ đồ họa hai chiều, sau đó được phát triển và mở rộng cho đồ họa ba chiều.

Các hàm của GKS thực sự chỉ là các mô tả trừu tượng, độc lập với bất kì ngôn ngữ lập trình nào. Để cài đặt một chuẩn đồ họa cho ngôn ngữ cụ thể nào, các cú pháp tương ứng sẽ được xác định và cụ thể hóa.

Mặc dù GKS xác lập được các ý tưởng ban đầu cho các hàm đồ họa cơ sở, tuy nhiên nó không cung cấp một cách thức chuẩn cho việc giao tiếp đồ họa với các thiết bị xuất. Nó cũng không xác định các cách thức cho các mô hình thời gian thực cũng như các cách thức lưu trữ và chuyển đổi hình ảnh. Các chuẩn cho các cách thức này được xây dựng riêng, cụ thể là : Các chuẩn cho các cách thức giao tiếp thiết bị được cho bởi hệ CGI (Computer Graphics Interface System), hệ CGM (Computer Graphics Metafile) xác định các chuẩn cho việc lưu trữ và chuyển đổi hình ảnh, và hệ PHIGS (Programmer's Hierarchical Interactive Graphics Standard) xác định các cách thức chuẩn cho các mô hình thời gian thực và các khả năng lập trình ở mức độ cao hơn mà chưa được quan tâm tới trong GKS.

## TÓM TẮT

Sự ra đời của đồ họa máy tính thực sự là cuộc cách mạng trong giao tiếp giữa người dùng và máy tính. Với lượng thông tin trực quan, đa dạng và phong phú được chuyển tải qua hình ảnh, các ứng dụng đồ họa máy tính đã lôi cuốn nhiều người nhờ tính thân thiện, dễ dùng, kích thích khả năng sáng tạo và tăng đáng kể hiệu suất làm việc.

Đồ họa máy tính ngày nay được ứng dụng rất rộng rãi trong nhiều lĩnh vực khoa học, kĩ thuật, nghệ thuật, kinh doanh, quản lí, … Các ứng dụng đồ họa rất đa dạng, phong phú và phát triển liên tục không ngừng. Ngày nay, hầu như không có chương trình ứng dụng nào mà không sử dụng kĩ thuật đồ họa để làm tăng tính hấp dẫn của mình.

Một hệ đồ họa bao giờ cũng có hai thành phần chính đó là phần cứng và phần mềm..

Thành phần phần cứng bao gồm các thiết bị hiển thị (hay là thiết bị xuất) và các thiết bị nhập. Tiêu biểu nhất trong các thiết bị hiển thị là màn hình mà cơ chế hoạt động dựa trên cấu tạo của ống tia âm cực CRT. Các thiết bị nhập dữ liệu thường gặp bao gồm bàn phím, chuột.

Phần mềm đồ họa có thể chia làm hai loại đó là các công cụ lập trình như các hàm thư viện của C, Pascal, GL, … và các ứng dụng phục vụ cho một mục đích nào đó như AutoCAD, Photoshop, … Hướng tiếp cận của chúng ta trong tài liệu này ở mức độ của người lập trình, nghĩa là chúng ta sẽ tìm hiểu các thuật toán, các nguyên lí để xây dựng nên các ứng dụng đồ họa chứ không phải là học cách sử dụng các phần mềm như AutoCAD, Photoshop, …

## BÀI TẬP

**1.** Cấu tạo và nguyên lí hoạt động của màn hình dạng điểm. Các khái niệm như vùng đệm khung, độ phân giải, tỉ số phương, … của màn hình dạng này.

**2.** Ý nghĩa và hoạt động của bảng tra LUT.

**3.** Ba màn hình có độ phân giải lần lượt là 640x480, 1024x768, 1280x1024. Hãy cho biết kích thước của vùng đệm khung (tính bằng byte) nếu mỗi pixel được mô tả bằng 8 bit, 12 bit, 24 bit.

**4.** Hai màn hình có độ phân giải là 640x480 và 1024x768. Cho biết số pixel được truy cập trong một giây của mỗi màn hình nếu tốc độ làm tươi của CRT là 60Hz.

**5.** Một màn hình có kích thước theo chiều ngang là 12 inche, chiều dọc là 9.6 inch. Hãy cho biết đường kính của mỗi điểm trên màn hình nếu độ phân giải là 1280x1024 và tỉ số phương là 1.

**6.** Hãy cho biết thông tin trong vùng đệm khung của các hình vẽ các kí tự B, G, H, …

**7.** Các hệ màu. Mối liên hệ giữa chúng.

**8.** Quy trình hiển thị đối tượng. Ý nghĩa của các hệ tọa độ.

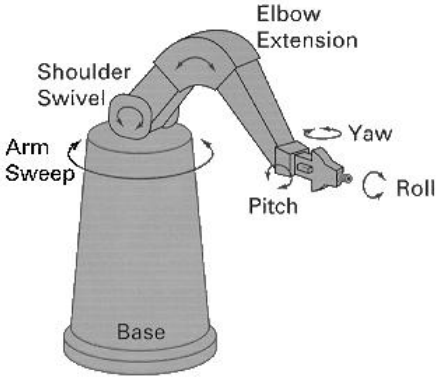
**9.** Tập các hàm đồ họa của một công cụ lập trình. Liên hệ tới các thư viện đồ họa của các ngôn ngữ đã học như C, Pascal, …

**10.** Tại sao cần phải chuẩn hóa các phần mềm ? Tìm hiểu các chuẩn GKS, PHIGS.

*CHƯƠNG 2*

# CÁC ĐỐI TƯỢNG ĐỒ HỌA CƠ SỞ

Bất kì một ảnh mô tả thế giới thực nào bao giờ cũng được cấu trúc từ tập các đối tượng đơn giản hơn. Ví dụ một ảnh thể hiện bài trí của một căn phòng sẽ được cấu trúc từ các đối tượng như cây cảnh, tủ kính, bàn ghế, tường, ánh sáng đèn, … Với các ảnh đồ họa phát sinh bằng máy tính, hình dạng và màu sắc của mỗi đối tượng có thể được mô tả riêng biệt bằng hai cách : hoặc là bằng dãy các pixel tương ứng hoặc là bằng tập các đối tượng hình học cơ sở như đoạn thẳng hay vùng tô đa giác, … Sau đó, các ảnh sẽ được hiển thị bằng cách nạp các pixel vào vùng đệm khung.



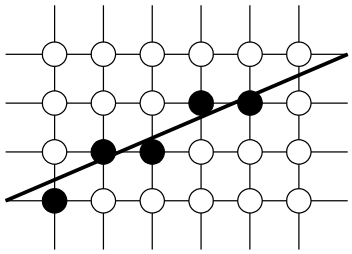

**Hình 2.1** – Ảnh cánh tay robot được cấu tạo từ các đối tượng đồ họa cơ sở

Với các ảnh được mô tả bằng các đối tượng hình học cơ sở, cần phải có một quá trình chuyển các đối tượng này về dạng ma trận các pixel trước. Quá trình này còn được gọi là quá trình chuyển đổi bằng dòng quét (scan-converting). Bất kì công cụ lập trình đồ họa nào cũng phải cung cấp các hàm để mô tả một ảnh dưới dạng các đối tượng hình học cơ sở hay còn gọi là các đối tượng đồ họa cơ sở (output primitives) và các hàm cho phép kết hợp tập các đối tượng cơ sở để tạo thành đối tượng có cấu trúc phức tạp hơn.

Mỗi đối tượng đồ họa cơ sở được mô tả thông qua dữ liệu về tọa độ và các thuộc tính của nó, đây chính là thông tin cho biết kiểu cách mà đối tượng được hiển thị. Đối tượng đồ họa cơ sở đơn giản nhất là điểm và đoạn thẳng, ngoài ra còn có đường tròn, và các đường conics, mặt bậc hai, các mặt và đường splines, các vùng tô đa giác, chuỗi kí tự, … cũng được xem là các đối tượng đồ họa cơ sở để giúp xây dựng các ảnh phức tạp. Chương này sẽ khảo sát các thuật toán hiển thị các đối tượng đồ họa cơ sở cho các thiết bị hiển thị dạng điểm.

Xét về mặt bản chất, các thuật toán này thực hiện quá trình chuyển đổi các đối tượng đồ họa cơ sở được mô tả trong hệ tọa độ thực về dãy các pixel có tọa độ nguyên của thiết bị hiển thị. Có hai yêu cầu đặt ra cho các thuật toán này đó là :

- Đối tượng được mô tả trong hệ tọa độ thực là đối tượng liên tục, còn đối tượng trong hệ tọa độ thiết bị là đối tượng rời rạc, do đó bản chất của quá trình chuyển đổi này chính là sự rời rạc hóa và nguyên hóa các đối tượng sao cho có thể xác định các điểm nguyên xấp xỉ đối tượng một cách tốt nhất, thực nhất. Nghĩa là đối tượng hiển thị bằng lưới nguyên trên thiết bị hiển thị phải có hình dạng tương tự như đối tượng trong lưới tọa độ thực và "có vẻ" liên tục, liền nét. Sự liên tục trên lưới nguyên của thiết bị hiển thị có được do mắt người không thể phân biệt được hai điểm quá gần nhau.
- Do các đối tượng đồ họa cơ sở là thành phần chính cấu trúc các đối tượng phức tạp nên các thuật toán hiển thị chúng cần phải được tối ưu hóa về mặt tốc độ, đây chính là điểm mấu chốt cho việc ra đời các thuật toán khác nhau.



**Hình 2.2** – Quá trình chuyển đổi một đoạn thẳng về dãy các pixel tương ứng

## 4. CÁC ĐỐI TƯỢNG ĐỒ HỌA CƠ SỞ

### 4.1. Hệ tọa độ thế giới thực và hệ tọa độ thiết bị

#### 4.1.1. Hệ tọa độ thế giới thực

Hệ tọa độ thế giới thực (hay hệ tọa độ thực) là hệ tọa độ được dùng mô tả các đối tượng thế giới thực. Một trong các hệ tọa độ thực thường được dùng nhất đó là hệ tọa độ Descartes. Với hệ tọa độ này, bất kì một điểm nào trong mặt phẳng cũng được mô tả bằng một cặp tọa độ (x, y) trong đó x, y $\in$ **R**. Gốc tọa độ là điểm O có tọa độ (0, 0). Các trục tọa độ có chiều dương được quy ước như hình 2.3; Ox, Oy lần lượt được gọi là trục hoành, trục tung; x là khoảng cách từ điểm đến trục hoành hay còn được gọi là hoành độ, y là khoảng cách từ điểm đến trục tung hay còn được gọi là tung độ.

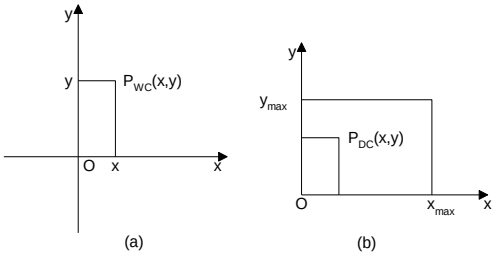
Các tọa độ thế giới thực cho phép người dùng sử dụng bất kì một thứ nguyên (dimension) quy ước như foot, cm, mm, km, inch, ... nào và có thể lớn nhỏ tùy ý.

#### 4.1.2. Hệ tọa độ thiết bị

Hệ tọa độ thiết bị là hệ tọa độ được dùng bởi một thiết bị xuất cụ thể nào đó như máy in, màn hình, ... Đặc điểm chung của các hệ tọa độ thiết bị đó là :

- Các điểm trong hệ tọa độ thiết bị cũng được mô tả bởi một cặp tọa độ (x, y), tuy nhiên điểm khác với hệ tọa độ thực là x, y $\in$ **N**. Điều này cho thấy các điểm trong hệ tọa độ thực được định nghĩa liên tục, còn các điểm trong các hệ tọa độ thiết bị là rời rạc do tính chất của tập các số tự nhiên.

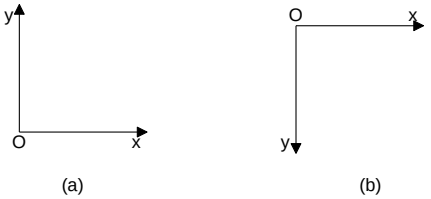
- Các tọa độ x, y của hệ tọa độ thiết bị không thể lớn tùy ý mà đều bị giới hạn trong một khoảng nào đó. Một số thiết bị chỉ cho x chạy trong đoạn[0,639], y chạy trong đoạn [0,479]. Khoảng giới hạn các tọa độ x, y là khác nhau đối với từng loại thiết bị khác nhau.




**Hình 2.3** – Hệ tọa độ thực (a) và hệ tọa độ thiết bị (b)

Hệ tọa độ với các hướng của các trục tọa độ như trên còn được gọi là hệ tọa độ theo quy ước bàn tay phải.

Ngoài ra do cách tổ chức bộ nhớ nên thông thường các hệ tọa độ thiết bị thường dựa trên hệ tọa độ theo quy ước bàn tay trái.




**Hình 2.4** - Hệ tọa độ theo quy ước bàn tay phải (a) và quy ước bàn tay trái (b)

### 4.2. Điểm

Điểm là thành phần cơ sở được định nghĩa trong một hệ tọa độ. Đối với hệ tọa độ hai chiều mỗi điểm được xác định bởi cặp tọa độ (x, y).

Ngoài thông tin về tọa độ, điểm còn có thuộc tính là màu sắc.

### 4.3. Đoạn thẳng, đường gấp khúc

Một đường thẳng có thể xác định nếu biết hai điểm thuộc nó. Phương trình đường thẳng đi qua hai điểm ($x_1$, $y_1$) và ($x_2$, $y_2$) có dạng sau :

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1}$$

hay ở dạng tương đương : $(x - x_1)(y_2 - y_1) = (y - y_1)(x_2 - x_1)$

Khai triển ta có dạng : $y = mx + b$, trong đó :

$$m = \frac{Dy}{Dx}, Dy = y_2 - y_1, Dx = x_2 - x_1$$
$$b = y_1 - mx_1$$

Đây còn được gọi là phương trình đoạn chắn của đường thẳng.

Nếu khai triển dưới dạng :

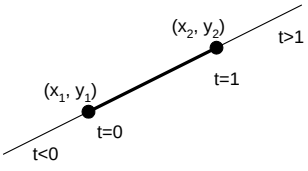
$$(y_2 - y_1)x - (x_2 - x_1)y - x_1y_2 + x_2y_1 = 0$$

và đặt $A = y_2 - y_1, B = -(x_2 - x_1), C = x_2y_1 - x_1y_2$ thì phương trình đường thẳng sẽ có dạng $Ax + By + C = 0$, dạng này được gọi là phương trình tổng quát của đường thẳng.

Phương trình tham số của đường thẳng có dạng các tọa độ x, y được mô tả qua một thành phần thứ ba là t. Dạng này rất thuận tiện khi khảo sát các đoạn thẳng.

$$\begin{cases} x = (1 - t)x_1 + tx_2 \\ y = (1 - t)y_1 + ty_2 \end{cases}$$

Nếu $t \in [0,1]$, ta có các điểm (x,y) thuộc về đoạn thẳng giới hạn bởi hai điểm ($x_1$, $y_1$) và ($x_2$, $y_2$), nếu $t \in [-\infty,+\infty]$, ta sẽ có toàn bộ đường thẳng.



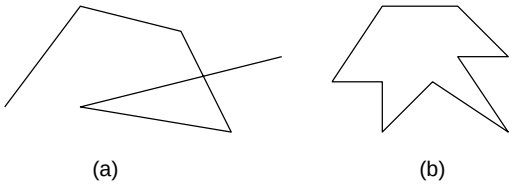

Một đoạn thẳng là một đường thẳng bị giới hạn bởi hai điểm đầu, cuối.

**Hình 2.5** – Dạng tham số của phương trình đường thẳng

Đường gấp khúc là tập các đoạn thẳng nối với nhau một cách tuần tự. Các đoạn thẳng này không nhất thiết phải tạo thành

một hình khép kín và các đoạn có thể cắt lẫn nhau. Điểm giao của hai đoạn thẳng được gọi là đỉnh. Các đường gấp khúc được xác định qua danh sách các đỉnh, mỗi đỉnh được cho bởi các cặp tọa độ $(x_i, y_i)$.

Một đa giác là một đường gấp khúc có điểm đầu và điểm cuối trùng nhau.



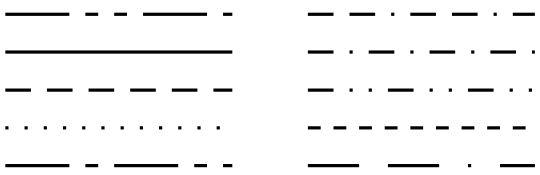

**Hình 2.6** – Đường gấp khúc (a) và đa giác (b)

*Các thuộc tính của đoạn thẳng bao gồm :*

- Màu sắc
- Độ rộng của nét vẽ.
- Kiểu nét vẽ của đoạn thẳng : có thể là một trong các dạng như hình 2.7. Hầu hết các công cụ đồ họa đều định nghĩa tập các kiểu nét vẽ đoạn thẳng có thể dùng và cho phép người dùng định nghĩa kiểu đoạn thẳng của mình thông qua một mẫu (pattern) gồm các số 0, 1.

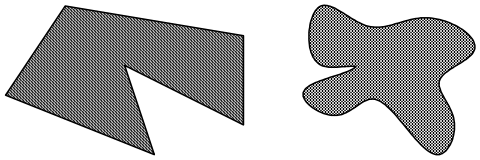
Đối với đường gấp khúc, các đoạn thẳng trong cùng một đường gấp khúc thì có cùng một thuộc tính.

**Hình 2.7** – Một số kiểu nét vẽ của đoạn thẳng



### 4.4. Vùng tô

Một vùng tô bao gồm đường biên và vùng bên trong. Đường biên là một đường khép kín ví dụ như đa giác.



*Các thuộc tính của vùng tô bao gồm:*

- Thuộc tính của đường biên : chính là các thuộc tính như thuộc tính của đoạn thẳng.
- Thuộc tính của vùng bên trong : bao gồm màu tô và mẫu tô.

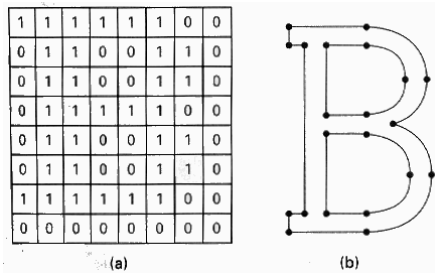
**Hình 2.8** – Vùng tô với các dạng đường biên và mẫu tô khác nhau

### 4.5. Kí tự, chuỗi kí tự

Các chuỗi kí tự giúp hiển thị nội dung các thông điệp theo một ngôn ngữ nào đó.

*Các thuộc tính của kí tự bao gồm :*

- Màu sắc của các kí tự.
- Font chữ : bộ kí tự dùng hiển thị; Nó định nghĩa kiểu, kích thước của kí tự hiển thị. Hình dạng của mỗi kí tự có thể được xác định bởi một tập các đường gấp khúc (trường hợp font vector) hay là mẫu các pixel (font bitmap). Có nhiều loại font khác nhau như font bitmap, font truetype, font CHR, ...
- Kích thước : chiều cao và chiều rộng của kí tự. Các kí tự định nghĩa bằng đường gấp khúc có thể dễ dàng thay đổi kích thước hơn là các kí tự định nghĩa bằng mẫu các pixel.
- Khoảng cách giữa các kí tự.
- Sự canh chỉnh (gióng lề) : canh trái (left text), canh phải (right text), canh giữa (center text), canh đều nhau (justify text).
- Cách hiển thị tuần tự của các kí tự : có thể là phải sang trái, từ trên xuống dưới, từ trái sang phải, từ dưới lên trên.
- Hướng của kí tự.




**Hình 2.9** – Dạng bitmap và vector của font kí tự B

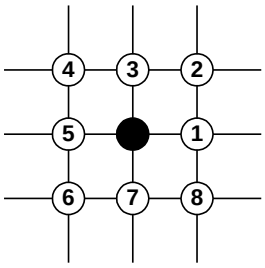
## 5. CÁC THUẬT TOÁN VẼ ĐƯỜNG

Giả sử tọa độ các điểm nguyên sau khi xấp xỉ đối tượng thực lần lượt là $(x_i, y_i), i = 0,\ldots$. Đây là các điểm nguyên sẽ được hiển thị trên màn hình.

Bài toán đặt ra là nếu biết được $(x_i, y_i)$ là tọa độ nguyên xác định ở bước thứ i, điểm nguyên tiếp theo $(x_{i+1}, y_{i+1})$ sẽ

được xác định như thế nào.

Nhận xét rằng để đối tượng hiển thị trên lưới nguyên được liền nét, các điểm mà $(x_{i+1}, y_{i+1})$ có thể chọn chỉ là một trong tám điểm được đánh số từ 1 đến 8 trong hình 2.10 (điểm đen chính là $(x_i, y_i)$).Hay nói cách khác : $(x_{i+1}, y_{i+1}) = (x_i \pm 1, y_i \pm 1)$.

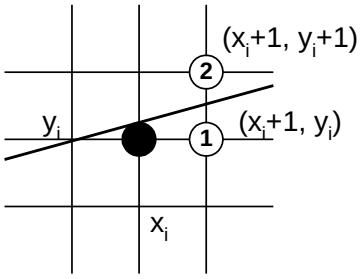



Dáng điệu của đường sẽ cho ta gợi ý khi chọn một trong tám điểm trên. Cách chọn các điểm như thế nào sẽ tùy thuộc vào từng thuật toán trên cơ sở xem xét tới vấn đề tối ưu tốc độ.

**Hình 2.10** – Các điểm $(x_{i+1}, y_{i+1})$ có thể chọn ở bước (i+1)

### 5.1. Thuật toán vẽ đoạn thẳng

Xét đoạn thẳng có hệ số góc $0 < m < 1$ và $Dx > 0$.




Với các đoạn thẳng dạng này, nếu $(x_i, y_i)$ là điểm đã xác định được ở bước thứ i (điểm màu đen) thì điểm cần chọn $(x_{i+1}, y_{i+1})$ ở bước thứ (i+1) sẽ là một trong hai trường hợp như hình vẽ sau :

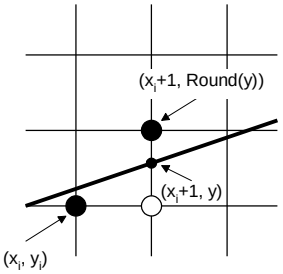
**Hình 2.11** – Các điểm $(x_{i+1}, y_{i+1})$ chọn ở bước (i+1) cho trường hợp đoạn thẳng có hệ số góc 0<m<1

Như vậy : $\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} \in \{y_i, y_i + 1\} \end{cases}$

Vấn đề còn lại là cách chọn một trong hai điểm trên như thế nào để có thể tối ưu về mặt tốc độ.

#### 5.1.1. Thuật toán DDA (Digital Differential Analyzer)

Với thuật toán DDA, việc quyết định chọn $y_{i+1}$ là $y_i$ hay $y_i + 1$, dựa vào phương trình của đoạn thẳng $y = mx + b$. Nghĩa là, ta sẽ tính tọa độ của điểm $(x_i + 1, y)$ thuộc về đoạn thẳng thực. Tiếp đó, $y_{i+1}$ sẽ là giá trị sau khi làm tròn giá trị tung độ *y*.




Như vậy : $\begin{aligned} y &= m(x_i + 1) + b \\ y_{i+1} &= Round(y) \end{aligned}$

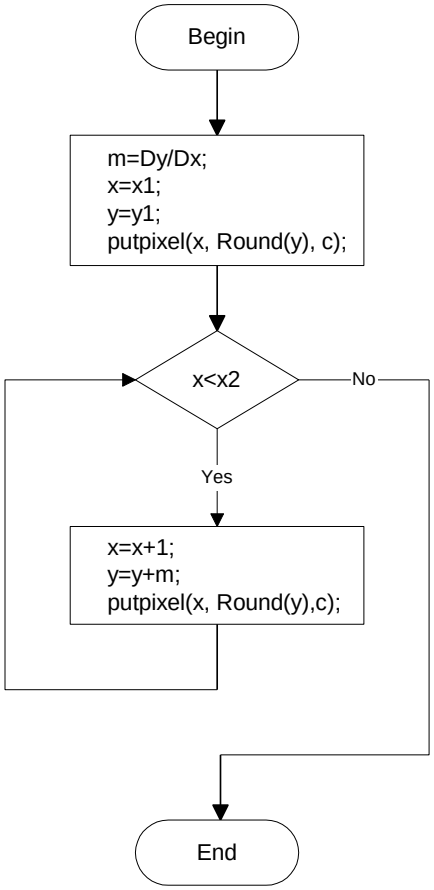
**Hình 2.12** – Minh họa thuật toán DDA

Nếu tính trực tiếp giá trị thực y ở mỗi bước từ phương trình $y = mx + b$ thì phải cần một phép toán nhân và một phép toán cộng số thực. Để cải thiện tốc độ, người ta tính giá trị thực của y ở mỗi bước theo cách sau để khử phép tính nhân trên số thực :

Nhận xét rằng : $y_{sau} = mx_{i+1} + b = m(x_i + 1) + b$

$y_{trước} = mx_i + b$

$\Rightarrow y_{sau} = y_{trước} + m$



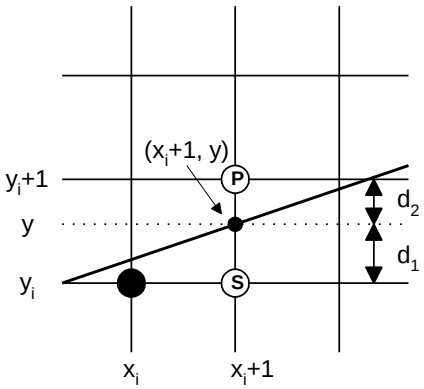

***Lưu đồ thuật toán DDA vẽ đoạn thẳng qua hai điểm $(x_1, y_1)$ và $(x_2, y_2)$***

***Cài đặt minh họa thuật toán DDA***

```
#define Round(a)  int(a+0.5)
int Color = GREEN;

void LineDDA (int x1, int y1, int x2, int y2)
{
    int x = x1;
    float y = y1;
    float m = float(y2-y1)/(x2-x1);

    putpixel(x, Round(y), Color);
    for(int i=x1; i<x2; i++)
    {
        x++;
        y +=m;
        putpixel(x, Round(y), Color);
    }

} // LineDDA
```

***Nhận xét***

- Việc sử dụng công thức $y_{sau} = y_{trước} + m$ để tính giá trị y tại mỗi bước đã giúp cho thuật toán DDA nhanh hơn hẳn so với cách tính y từ phương trình $y = mx + b$ do khử được phép nhân trên số thực. Tuy nhiên, việc cộng dồn giá trị thực m vào y có thể sẽ tích lũy sai số làm cho hàm làm tròn có kết quả sai dẫn tới việc xác định vị trí của điểm vẽ ra bị chệch hướng so với đường thẳng thực. Điều này chỉ xảy ra khi vẽ đoạn thẳng khá dài.
- Tuy đã khử được phép nhân số thực nhưng thuật toán DDA vẫn còn bị hạn chế về mặt tốc độ do vẫn còn phép toán cộng số thực và làm tròn. Có thể khắc phục thao tác cộng số thực m và làm tròn trong thuật toán bằng cách nhận xét $m = \frac{Dy}{Dx}$ với *Dy, Dx* là các số nguyên.




**Hình 2.13** – Minh họa thuật toán Bresenham

### 5.1.2. Thuật toán Bresenham

Thuật toán Bresenham đưa ra cách chọn $y_{i+1}$ là $y_i$ hay $y_i + 1$ theo một hướng khác sao cho có thể tối ưu hóa về mặt tốc độ so với thuật toán DDA. Vấn đề mấu chốt ở đây là làm thế nào để hạn chế tối đa các phép toán trên số thực trong thuật toán.

Gọi $(x_i + 1, y)$ là điểm thuộc đoạn thẳng. Ta có: $y = m(x_i + 1) + b$.

Đặt $\begin{aligned} d_1 &= y - y_i \\ d_2 &= (y_i + 1) - y \end{aligned}$

Xét tất cả các vị trí tương đối của *y* so với $y_i$ và $y_i + 1$, việc chọn điểm $(x_{i+1}, y_{i+1})$ là S hay P phụ thuộc vào việc so sánh $d_1$ và $d_2$ hay dấu của $d_1 - d_2$ :

- Nếu $d_1 - d_2 < 0$, ta sẽ chọn điểm S, tức là $y_{i+1} = y_i$.
- Ngược lại, nếu $d_1 - d_2 \geq 0$, ta sẽ chọn điểm P, tức là $y_{i+1} = y_i + 1$.

Xét $p_i = Dx(d_1 - d_2) = Dx(2y - 2y_i - 1)$

$\Rightarrow p_i = Dx[2(m(x_i + 1) + b) - 2y_i - 1]$

Thay $m = \frac{Dy}{Dx}$ vào phương trình trên ta được : $p_i = 2Dyx_i - 2Dxy_i + c$, với $c = 2Dy + (2b-1)Dx$.

Nhận xét rằng do $Dx > 0$ nên dấu của biểu thức $d_1 - d_2$ cũng chính là dấu của $p_i$. Hay nói một cách khác, nếu tại bước thứ i ta xác định được dấu của $p_i$ thì xem như ta xác định được điểm cần chọn ở bước (i+1). Vấn đề còn lại là làm thế nào để tính được $p_i$ tại mỗi bước thật nhanh.

Ta có :

$$p_{i+1} - p_i = (2Dyx_{i+1} - 2Dxy_{i+1} + c) - (2Dyx_i - 2Dxy_i + c)$$

$$\Leftrightarrow p_{i+1} - p_i = 2Dy(x_{i+1} - x_i) - 2Dx(y_{i+1} - y_i)$$

$$\Leftrightarrow p_{i+1} - p_i = 2Dy - 2Dx(y_{i+1} - y_i), \text{ do } x_{i+1} = x_i + 1$$

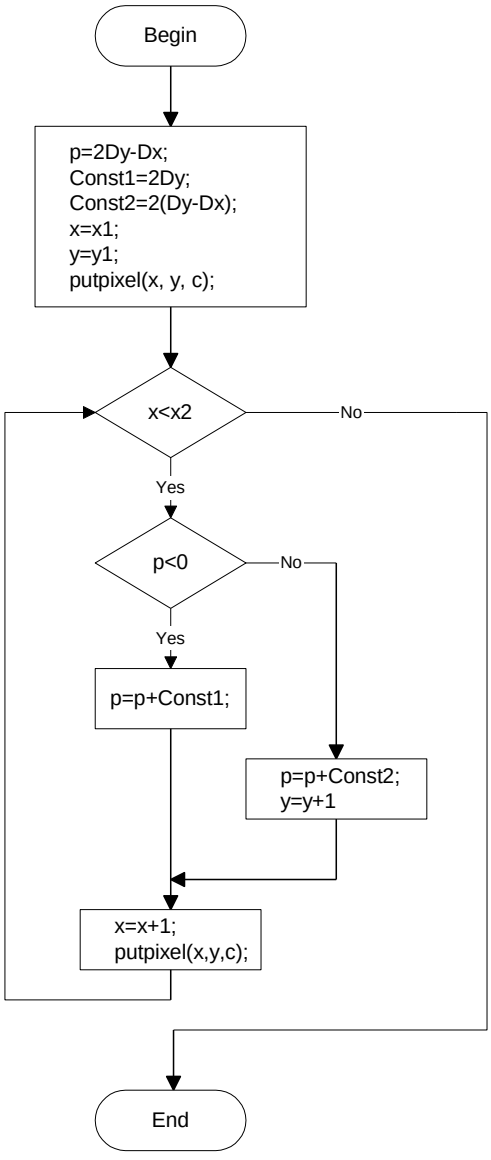
Từ đây ta có thể suy ra cách tính $p_{i+1}$ từ $p_i$ như sau :

- Nếu $p_i < 0$ thì $p_{i+1} = p_i + 2Dy$ do ta chọn $y_{i+1} = y_i$.
- Ngược lại, nếu $p_i \geq 0$, thì $p_{i+1} = p_i + 2Dy - 2Dx$, do ta chọn $y_{i+1} = y_i + 1$.

Giá trị $p_0$ được tính từ điểm vẽ đầu tiên $(x_0, y_0)$ theo công thức : $p_0 = 2Dyx_0 - 2Dxy_0 + c = 2Dyx_0 - 2Dxy_0 + 2Dy - (2b-1)Dx$

Do $(x_0, y_0)$ là điểm nguyên thuộc về đoạn thẳng nên ta có $y_0 = mx_0 + b = \frac{Dy}{Dx}x_0 + b$. Thế vào phương trình trên ta suy ra : $p_0 = 2Dy - Dx$.

***Lưu đồ thuật toán Bresenham***



***Cài đặt minh họa thuật toán Bresenham***

```
void LineBres (int x1, int y1, int x2, int y2)
{
    int Dx, Dy, p, Const1, Const2;
    int x, y;

    Dx     = x2 - x1;
    Dy     = y2 - y1;
    p      = 2*Dy - Dx; // Dy <<1 - Dx
    Const1 = 2*Dy; // Dy <<1
    Const2 = 2*(Dy-Dx); // (Dy-Dx) <<1
    x = x1;
    y = y1;
    putpixel(x, y, Color);
    for(i=x1; i<x2; i++)
    {
        if (p<0)
            p += Const1;
        else
        {
            p += Const2;
            y++;
        }
        x++;
        putpixel(x, y, Color);
    }
} // LineBres
```

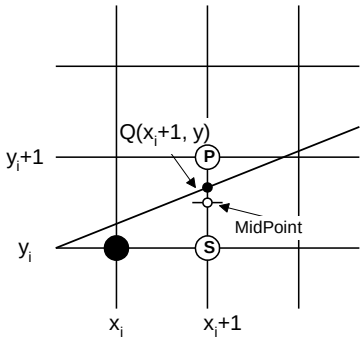
***Nhận xét***

- Thuật toán Bresenham chỉ làm việc trên số nguyên và các thao tác trên số nguyên chỉ là phép cộng và phép dịch bit (phép nhân 2) điều này là một cải tiến làm tăng tốc độ đáng kể so với thuật toán DDA. Ý tưởng chính của thuật toán nằm ở chỗ xét dấu $p_i$ để quyết định điểm kế tiếp, và sử dụng công thức truy hồi $p_{i+1} - p_i$ để tính $p_i$ bằng các phép toán đơn giản trên số nguyên.
- Thuật toán này cho kết quả tương tự như thuật toán DDA.

### 5.1.3. Thuật toán MidPoint

Thuật toán MidPoint đưa ra cách chọn $y_{i+1}$ là $y_i$ hay $y_i + 1$ bằng cách so sánh điểm thực $Q(x_i + 1, y)$ với điểm MidPoint là trung điểm của S và P. Ta có :

- Nếu điểm Q nằm dưới điểm MidPoint, ta chọn S.
- Ngược lại nếu điểm Q nằm trên điểm MidPoint ta chọn P.




**Hình 2.14** – Minh họa thuật toán MidPoint

Ta có dạng tổng quát của phương trình đường thẳng :

$$Ax + By + C = 0$$

với $A = y_2 - y_1, B = -(x_2 - x_1), C = x_2 y_1 - x_1 y_2$

Đặt $F(x, y) = Ax + By + C$, ta có nhận xét :

$$F(x,y)\begin{cases} < 0, \text{nếu } (x,y) \text{ nằm phía trên đường thẳng} \\ = 0, \text{nếu } (x,y) \text{ thuộc về đường thẳng} \\ > 0, \text{nếu } (x,y) \text{ nằm phía dưới đường thẳng.} \end{cases}$$

Lúc này việc chọn các điểm S, P ở trên được đưa về việc xét dấu của $p_i = 2F(\text{MidPoint}) = 2F\left(x_i + 1, y_i + \frac{1}{2}\right)$.

- Nếu $p_i < 0$, điểm MidPoint nằm phía trên đoạn thẳng. Lúc này điểm thực Q nằm dưới điểm MidPoint nên ta chọn S, tức là $y_{i+1} = y_i$.
- Ngược lại, nếu $p_i \geq 0$, điểm MidPoint nằm phía dưới đoạn thẳng. Lúc này điểm thực Q nằm trên điểm MidPoint nên ta chọn P, tức là $y_{i+1} = y_i + 1$.

Mặt khác :

$$p_{i+1} - p_i = 2F\left(x_{i+1} + 1, y_{i+1} + \frac{1}{2}\right) - 2F\left(x_i + 1, y_i + \frac{1}{2}\right)$$

$$\Leftrightarrow p_{i+1} - p_i = 2\left[A(x_{i+1} + 1) + B\left(y_{i+1} + \frac{1}{2}\right) + C\right] - 2\left[A(x_i + 1) + B\left(y_i + \frac{1}{2}\right) + C\right]$$

$$\Leftrightarrow p_{i+1} - p_i = 2A + 2B(y_{i+1} - y_i) = 2Dy - 2Dx(y_{i+1} - y_i)$$

Vậy :

- $p_{i+1} = p_i + 2Dy$, nếu $p_i < 0$ do ta chọn $y_{i+1} = y_i$.
- $p_{i+1} = p_i + 2Dy - 2Dx$, nếu $p_i \geq 0$ do ta chọn $y_{i+1} = y_i + 1$.

Ta tính giá trị $p_0$ ứng với điểm ban đầu $(x_0, y_0)$, với nhận xét rằng $(x_0, y_0)$ là điểm thuộc về đoạn thẳng, tức là có : $Ax_0 + By_0 + C = 0$

$$p_0 = 2F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = 2\left[A(x_0 + 1) + B\left(y_0 + \frac{1}{2}\right) + C\right]$$

$$\Rightarrow p_0 = 2(Ax_0 + By_0 + C) + 2A + B = 2A + B = 2Dy - Dx$$

Nhận xét rằng thuật toán MidPoint cho kết quả tương tự như thuật toán Bresenham.

### 5.2. Thuật toán vẽ đường tròn

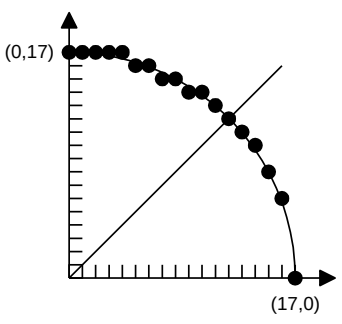
Phương trình đường tròn có tâm là gốc tọa độ, bán kính R là : $x^2 + y^2 = R^2$. Từ phương trình này ta có thể đưa về dạng $y = \pm\sqrt{R^2 - x^2}$. Để vẽ các đường tròn có tâm $(x_C, y_C)$ bất kì, đơn giản chỉ cần tịnh tiến các điểm sau khi vẽ xong đường tròn có tâm là gốc tọa độ theo vector tịnh tiến $(x_C, y_C)$.

#### 5.2.1. Một số cách tiếp cận vẽ đường tròn

Do tính đối xứng nên để vẽ toàn bộ đường tròn, ta chỉ cần vẽ cung ¼ đường tròn sau đó lấy đối xứng để xác định các điểm còn lại.

Một trong những cách đơn giản nhất là cho x chạy từ 0 đến R, sau đó tính y từ công thức trên (chỉ lấy giá trị dương) rồi làm tròn để xác định giá trị nguyên tương ứng. Cách làm này không hiệu quả do gặp phải các phép toán nhân và lấy căn làm hạn chế tốc độ, ngoài ra đường tròn vẽ ra theo cách này có thể không liền nét (trừ trường hợp R lớn) khi x gần R (do chỉ có một giá trị y duy nhất cho một giá trị x). Chúng ta có thể khắc phục điều này bằng cách điều chỉnh đối tượng thay đổi là x (rồi tính y theo x) hay y (rồi tính x theo y) tùy vào giá trị tuyệt đối của hệ số góc đường tròn là lớn hơn hay nhỏ hơn 1, nhưng cách làm này đòi hỏi thêm các phép tính toán và kiểm tra nên làm cho thuật toán phức tạp thêm. (Xem hình 2.15)

Một cách tiếp cận khác là vẽ các điểm $(R\cos(\theta), R\sin(\theta))$, với $\theta$ chạy từ $0^0$ đến $90^0$. Cách này sẽ khắc phục hạn chế đường không liền nét của thuật toán trên, tuy nhiên điểm hạn chế chính của thuật toán này đó là chọn bước nhảy cho $\theta$ như thế nào cho phù hợp khi bán kính thay đổi.



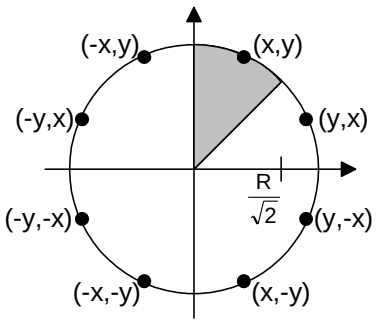

**Hình 2.15** – Đường tròn vẽ ra không liền nét theo cách vẽ trên

#### 5.2.2. Thuật toán MidPoint

Do tính đối xứng của đường tròn (C) nên ta chỉ cần vẽ cung ($C^{1/8}$) là cung 1/8 đường

tròn, sau đó lấy đối xứng. Cung ($C^{1/8}$) được mô tả như sau (cung của phần tô xám trong hình vẽ) :

$$\begin{cases} 0 \le x \le R\dfrac{\sqrt{2}}{2} \\ R\dfrac{\sqrt{2}}{2} \le y \le R \end{cases}$$

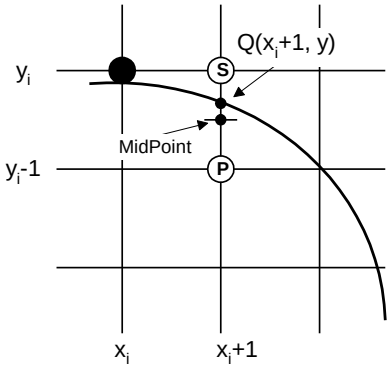



**Hình 2.16** – Các vị trí đối xứng trên đường tròn (C) tương ứng với (x,y)

Như vậy nếu có (x, y) ∈ ($C^{1/8}$) thì các điểm : (y, x), (y,-x), (x,-y), (-x,-y), (-y,-x), (-y,x), (-x,y) sẽ thuộc (C).

Chọn điểm bắt đầu để vẽ là điểm (0,R). Dựa vào hình vẽ, nếu $(x_i, y_i)$ là điểm nguyên đã tìm được ở bước thứ i, thì điểm $(x_{i+1}, y_{i+1})$ ở bước thứ (i+1) là sự lựa chọn giữa S và P.

Như vậy : $\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} \in \{y_i, y_i - 1\} \end{cases}$




Tương tự như thuật toán MidPoint vẽ đoạn thẳng, việc quyết định chọn một trong hai điểm S và P sẽ được thực hiện thông qua việc xét dấu của một hàm nào đó tại điểm MidPoint là điểm nằm giữa chúng.

**Hình 2.17** – Thuật toán MidPoint vẽ đường tròn

Đặt $F(x,y) = x^2 + y^2 - R^2$, ta có :

$$F(x,y)\begin{cases} < 0, \text{nếu } (x,y) \text{ nằm trong đường tròn} \\ = 0, \text{nếu } (x,y) \text{ nằm trên đường tròn} \\ > 0, \text{nếu } (x,y) \text{ nằm ngoài đường tròn.} \end{cases}$$

Xét $p_i = F(\text{MidPoint}) = F\left(x_i + 1, y_i - \dfrac{1}{2}\right)$. Ta có :

- Nếu $p_i < 0$, điểm MidPoint nằm trong đường tròn. Lúc này điểm thực Q gần S hơn nên ta chọn S, tức là $y_{i+1} = y_i$.
- Ngược lại, nếu $p_i \ge 0$, điểm MidPoint nằm ngoài đường tròn. Lúc này điểm thực Q gần P hơn nên ta chọn P, tức là $y_{i+1} = y_i - 1$.

Mặt khác :

$$p_{i+1} - p_i = F\left(x_{i+1} + 1, y_{i+1} - \frac{1}{2}\right) - F\left(x_i + 1, y_i - \frac{1}{2}\right)$$

$$\Leftrightarrow p_{i+1} - p_i = \left[(x_{i+1}+1)^2 + \left(y_{i+1} - \frac{1}{2}\right)^2 - R^2\right] - \left[(x_i+1)^2 + \left(y_i - \frac{1}{2}\right)^2 - R^2\right]$$

$$\Leftrightarrow p_{i+1} - p_i = \left[(x_i+2)^2 + \left(y_{i+1} - \frac{1}{2}\right)^2 - R^2\right] - \left[(x_i+1)^2 + \left(y_i - \frac{1}{2}\right)^2 - R^2\right]$$

$$\Leftrightarrow p_{i+1} - p_i = 2x_i + 3 + \left(y_{i+1}^2 - y_i^2\right) - \left(y_{i+1} - y_i\right)$$

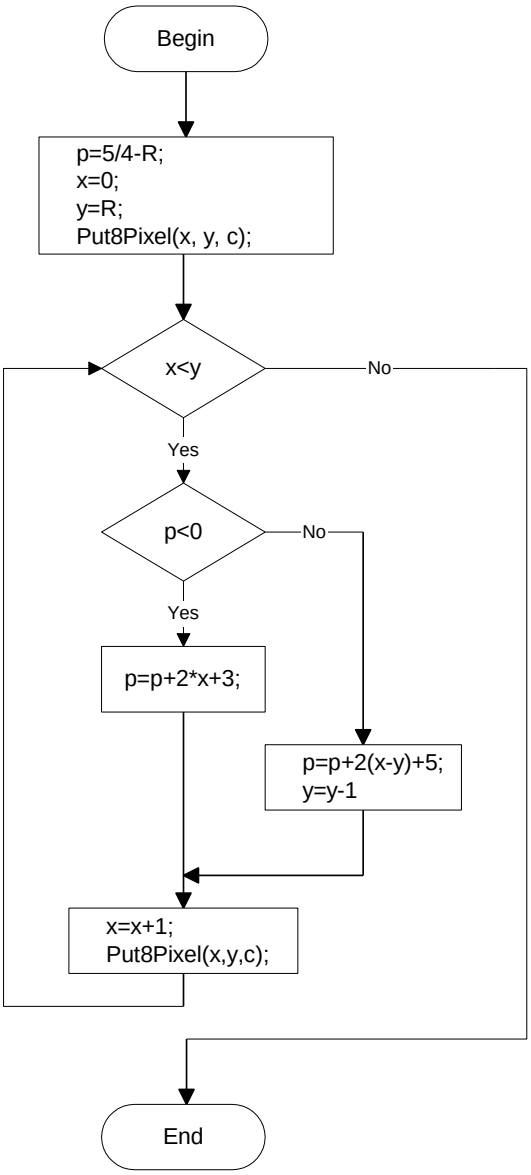
Vậy :

- $p_{i+1} = p_i + 2x_i + 3$, nếu $p_i < 0$ do ta chọn $y_{i+1} = y_i$.
- $p_{i+1} = p_i + 2x_i - 2y_i + 5$, nếu $p_i \ge 0$ do ta chọn $y_{i+1} = y_i - 1$.

Ta tính giá trị $p_0$ ứng với điểm ban đầu $(x_0, y_0) = (0, R)$.

$$p_0 = F\left(x_0 + 1, y_0 - \frac{1}{2}\right) = F\left(1, R - \frac{1}{2}\right) = \frac{5}{4} - R$$

***Lưu đồ thuật toán MidPoint vẽ đường tròn***




***Cài đặt minh họa thuật toán MidPoint vẽ đường tròn***

```
// Ve 8 diem doi xung
void Put8Pixel(int x, int y)
{

    putpixel(x, y, Color);
    putpixel(y, x, Color);
    putpixel(y, -x, Color);
    putpixel(x, -y, Color);
    putpixel(-x, -y, Color);
    putpixel(-y, -x, Color);
    putpixel(-y, x, Color);
    putpixel(-x, y, Color);

} // Put8Pixel

void CircleMidPoint (int R)
```

```
{
    int x, y;

    x = 0;
    y = R;
    Put8Pixel(x, y);
    p = 1 - R; // 5/4-R
    while (x < y)
    {
        if (p < 0)
            p += 2*x + 3;
        else
        {
            p += 2*(x -y) + 5;
            y--;
        }

        x++;
        Put8Pixel(x, y);
    }

} // CircleMidPoint
```

### 5.3. Thuật toán vẽ các đường conics và một số đường cong khác

Phương trình tổng quát của các đường conics có dạng : $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ . Giá trị của các hằng số A, B, C, D, E, F sẽ quyết định dạng của đường conics, cụ thể là nếu:

$$B^2 - 4AC \begin{cases} < 0, \text{dạng đường tròn (nếu A = C và B = 0 ) hay ellipse} \\ = 0, \text{dạng parabol} \\ > 0, \text{dạng hyperbol.} \end{cases}$$

Ta sẽ áp dụng ý tưởng của thuật toán MidPoint để vẽ các đường conics và một số đường cong khác, theo các bước tuần tự sau:

***Bước 1 :*** Dựa vào dáng điệu và phương trình đường cong, để xem thử có thể rút gọn phần đường cong cần vẽ hay không. Điều này sẽ làm tăng tốc độ vẽ so với việc phải vẽ toàn bộ đường cong. Một trong những cách đơn giản nhất là dựa vào tính đối xứng, tính chất của hàm chẵn, hàm lẻ, …

***Bước 2 :*** Tính đạo hàm để từ đó phân thành các vùng vẽ :

- Nếu $0 \le f'(x) \le 1$ thì $\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} \in \{y_i, y_i + 1\} \end{cases}$ (*)
- Nếu $-1 \le f'(x) \le 0$ thì $\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} \in \{y_i, y_i - 1\} \end{cases}$ (*)
- Nếu $f'(x) > 1$ thì $\begin{cases} y_{i+1} = y_i + 1 \\ x_{i+1} \in \{x_i, x_i + 1\} \end{cases}$ (*)
- Nếu $f'(x) < -1$ thì $\begin{cases} y_{i+1} = y_i + 1 \\ x_{i+1} \in \{x_i, x_i - 1\} \end{cases}$ (*)

Đây là bước quan trọng vì với việc xác định đối tượng x hay y biến thiên theo dáng điệu của đường cong sẽ đảm bảo đường sau khi được vẽ ra sẽ liền nét, không bị hở.

***Bước 3 :*** Xác định công thức của $p_i$ cho từng trường hợp để quyết định (*) dựa trên dấu của $p_i$. $p_i$ thường là hàm được xây dựng từ phương trình đường cong để cho $p_i = 0$ nếu $(x_i, y_i)$ thuộc về đường cong. Việc chọn $p_i$ cần phải chú ý sao cho thao tác tính $p_i$ sau này hạn chế phép toán trên số thực.

***Bước 4 :*** Tìm mối liên quan của $p_{i+1}$ và $p_i$ bằng cách xét hiệu $p_{i+1} - p_i$.

***Bước 5 :*** Tính $p_0$ và hoàn chỉnh thuật toán.

## 6. CÁC THUẬT TOÁN TÔ MÀU

Các vùng tô là một trong những đối tượng đồ họa cơ sở được hầu hết các công cụ lập trình đồ họa hỗ trợ. Có hai dạng vùng tô thường gặp đó là : tô bằng một màu thuần nhất (solid fill) hay tô theo một mẫu tô (fill-pattern) nào đó.

Một vùng tô thường được xác định bởi một đường khép kín nào đó gọi là đường biên. Một trong những dạng đường biên đơn giản nhất đó là đa giác.

Để tô màu một vùng tô, người ta thường chia làm hai công đoạn : công đoạn thứ nhất là xác định các điểm nào để tô và công đoạn còn lại đơn giản hơn đó là quyết định tô các điểm đó bằng giá trị màu nào. Công đoạn thứ hai chỉ thực sự phức tạp nếu ta tô theo một mẫu tô nào đó không phải là tô thuần một màu.

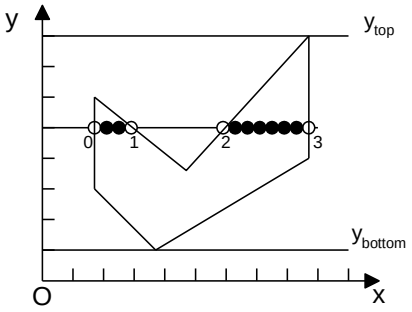
Có hai cách tiếp cận chính để tô màu một vùng tô đối với thiết bị hiển thị dạng điểm đó là : tô theo dòng quét (scan-line fill) và tô dựa theo đường biên (boundary fill).

Phương pháp tô theo dòng quét sẽ xác định các phần giao của các dòng quét kế tiếp nhau với đường biên của vùng tô, sau đó sẽ tô màu các điểm thuộc về phần giao này. Cách tiếp cận này thường được dùng để tô màu các đa giác, đường tròn, ellipse, và một số đường cong đơn giản khác. Phương pháp tô dựa theo đường biên sẽ bắt đầu từ một điểm ở bên trong vùng tô và từ đó loang dần ra cho tới khi ta gặp các điểm biên. Cách tiếp cận này thường được dùng cho các vùng tô có dạng đường biên phức tạp hơn.

### 6.1. Thuật toán tô màu dựa theo dòng quét

Giả sử vùng tô được cho bởi một đa giác N đỉnh : $P_i(x_i, y_i), i = 0,...N-1$. Đa giác này có thể là đa giác lồi, đa giác lõm, và cả đa giác tự cắt, …

Hình 2.18 sau minh họa ý tưởng chính của thuật toán. Với mỗi dòng quét, ta sẽ xác định phần giao của đa giác và dòng quét rồi tô màu các pixel thuộc đoạn giao đó. Để xác định các đoạn giao ta tiến hành việc tìm giao điểm của dòng quét với các cạnh của đa giác, sau đó các giao điểm này sẽ được sắp theo thứ tự tăng dần của hoành độ giao điểm. Các đoạn giao chính là các đoạn thẳng được giới hạn bởi từng cặp giao điểm một, ví dụ như (0,1), (2,3), ….




**Hình 2.18** – Thuật toán scan-line với một dòng quét nào đó

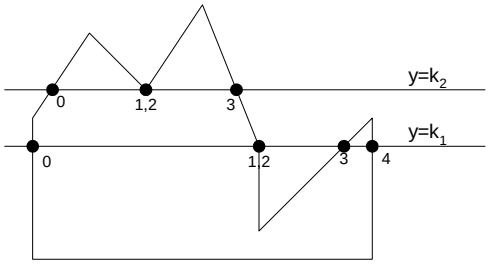
Ta có thể tóm bắt các bước chính của thuật toán :

- Tìm $y_{top}$, $y_{bottom}$ lần lượt là giá trị lớn nhất, nhỏ nhất của tập các tung độ của các đỉnh của đa giác đã cho $y_{top} = \max\{y_i, (x_i, y_i) \in P\}$, $y_{bottom} = \min\{y_i, (x_i, y_i) \in P\}$.
- Ứng với mỗi dòng quét $y = k$, với k thay đổi từ $y_{bottom}$ đến $y_{top}$, lặp :
  - ♦ Tìm tất cả các hoành độ giao điểm của dòng quét $y = k$ với các cạnh của đa giác.
  - ♦ Sắp xếp các hoành độ giao điểm theo thứ tự tăng dần : $x_0$, $x_1$, …
  - ♦ Tô màu các đoạn thẳng trên đường thẳng $y = k$ lần lượt được giới hạn bởi các cặp $(x_0, x_1), (x_2, x_3), ..., (x_{2k}, x_{2k+1})$.

Nếu chỉ dừng ở mức này và chuyển sang cài đặt, chúng ta sẽ gặp một số vấn đề sau :

- Nhận xét rằng, ứng với mỗi dòng quét, không phải lúc nào tất cả các cạnh của đa giác cũng tham gia cắt dòng quét. Do đó để cải thiện tốc độ cần phải có một cách nào đó để hạn chế được số cạnh cần tìm giao điểm ứng với mỗi dòng quét.
- Việc tìm giao điểm của cạnh đa giác với mỗi dòng quét sẽ gặp các phép toán phức tạp như nhân, chia, … trên số thực nếu ta dùng cách giải hệ phương trình tìm giao điểm. Điều này sẽ làm giảm tốc độ thuật toán khi phải lặp đi lặp lại nhiều lần thao tác này khi dòng quét quét qua đa giác.
- Nếu số giao điểm tìm được giữa các cạnh đa giác và dòng quét là lẻ thì việc nhóm từng cặp giao điểm kế tiếp nhau để hình thành các đoạn tô có thể sẽ không chính xác. Điều này chỉ xảy ra khi dòng quét đi ngang qua các đỉnh của đa giác. Nếu tính số giao điểm tại đỉnh dòng quét đi ngang qua là hai thì có thể sẽ cho kết quả tô không chính xác như trong trường hợp của hình 2.19. Ngoài ra, việc tìm giao điểm của dòng quét với các cạnh nằm ngang là một trường hợp đặc biệt cần phải có cách xử lí thích hợp.

Để giải quyết các vấn đề trên, cần phải xây dựng một cấu trúc dữ liệu và thuật toán thích hợp đối với chúng.




**Hình 2.19** – Dòng quét $y=k_2$ đi ngang qua đỉnh có thể sẽ cho kết quả tô không chính xác so với dòng quét $y=k_1$

### 6.1.1. Danh sách các cạnh kích hoạt AET (Active Edge Table)

Để hạn chế số cạnh cần tìm giao điểm ứng với mỗi dòng quét, ta xây dựng một số cấu trúc dữ liệu như sau :

***Cạnh đa giác (EDGE)***

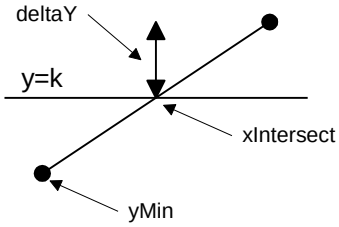
Mỗi cạnh của đa giác được xây dựng từ hai đỉnh kề nhau $P_i(x_i, y_i)$ và $P_{i+1}(x_{i+1}, y_{i+1})$ gồm các thông tin sau :

- $y_{Min}$ : giá trị tung độ nhỏ nhất trong 2 đỉnh của cạnh.
- xIntersect : hoành độ giao điểm của cạnh với ***dòng quét hiện đang xét.***
- DxPerScan : giá trị 1/m (m là hệ số góc của cạnh).
- deltaY : khoảng cách từ dòng quét hiện hành tới đỉnh $y_{Max}$.

***Danh sách các cạnh kích hoạt AET***

Danh sách này dùng để lưu các tập cạnh của đa giác có thể cắt ứng với dòng quét hiện hành và tập các điểm giao tương ứng. Nó có một số đặc điểm :

- Các cạnh trong danh sách được sắp theo thứ tự tăng dần của các hoành độ giao điểm để có thể tô màu các đoạn giao một cách dễ dàng.
- Thay đổi ứng với mỗi dòng quét đang xét, do đó danh sách này sẽ được cập nhật liên tục trong quá trình thực hiện thuật toán. Để hỗ trợ cho thao tác này, đầu tiên người ta sẽ tổ chức một danh sách chứa toàn bộ các cạnh của đa giác gọi là ET (Edge Table) được sắp theo thứ tự tăng dần của $y_{Min}$, rồi sau mỗi lần dòng quét thay đổi sẽ di chuyển các cạnh trong ET thỏa điều kiện sang AET.
- Một dòng quét $y = k$ chỉ cắt một cạnh của đa giác khi và chỉ khi $\begin{cases} k \geq y_{Min} \\ deltaY > 0 \end{cases}$. Chính vì vậy mà với cách tổ chức của ET (sắp theo thứ tự tăng dần của $y_{Min}$) điều kiện để chuyển các cạnh từ ET sang AET sẽ là $k \geq y_{Min}$; và điều kiện để loại một cạnh ra khỏi AET là $deltaY \leq 0$.



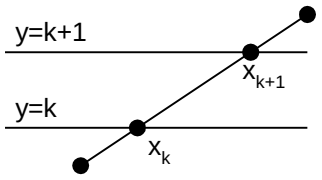

**Hình 2.20** – Thông tin của một cạnh

### 6.1.2. Công thức tìm giao điểm nhanh

Nếu gọi $x_k, x_{k+1}$ lần lượt là các hoành độ giao điểm của một cạnh nào đó với các dòng quét $y = k$ và $y = k + 1$, ta có :

$$x_{k+1} - x_k = \frac{1}{m}\big((k+1) - k\big) = \frac{1}{m} \text{ hay } x_{k+1} = x_k + \frac{1}{m}.$$

Như vậy nếu lưu hoành độ giao điểm ứng với dòng quét trước lại, cùng với hệ số góc của cạnh, ta có thể dễ dàng xác định hoành độ giao điểm ứng với dòng quét kế tiếp một cách đơn giản theo công thức trên. Điều này rút gọn đáng kể thao tác tìm giao điểm của cạnh ứng với dòng quét. Chính vì vậy mà trong thông tin của một cạnh chúng ta có hai biến DxPerScan và xIntersect.

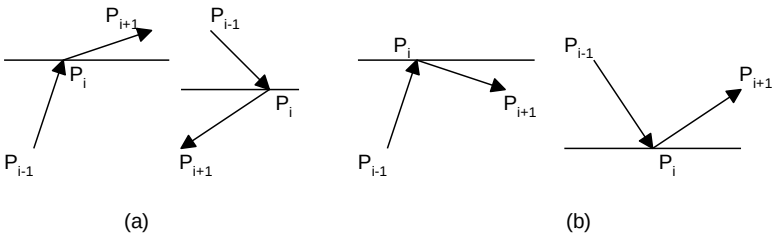



**Hình 2.21** – Công thức tìm giao điểm nhanh

### 6.1.3. Giải quyết trường hợp dòng quét đi ngang qua đỉnh

Người ta đưa ra quy tắc sau để tính số giao điểm khi dòng quét đi ngang qua đỉnh :

- Tính một giao điểm nếu chiều của hai cạnh kề của đỉnh đó có xu hướng tăng hay giảm.
- Tính hai giao điểm nếu chiều của hai cạnh kề của đỉnh đó có xu hướng thay đổi, nghĩa là tăng-giảm hay giảm-tăng.

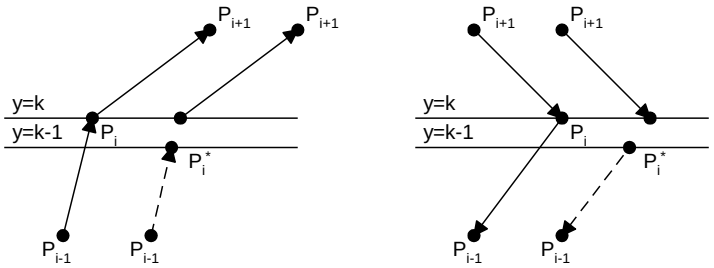



**Hình 2.22** – Quy tắc tính một giao điểm (a) và hai giao điểm (b)

Khi cài đặt để khỏi phải xét điều kiện này cho phức tạp, khi xây dựng dữ liệu cho mỗi cạnh trước khi đưa vào ET, người ta sẽ xử lí các cạnh có đỉnh tính hai giao điểm bằng cách loại đi một pixel trên cùng của một trong hai cạnh như hình 2.23 :

**Hình 2.23** – Cạnh $P_{i-1}P_i$ được lưu trong ET chỉ là $P_{i-1}P_i^*$

Cài đặt minh họa sau sử dụng chung một danh sách EDGELIST cho cả ET và AET. AET được quản lí nhờ vào hai con trỏ FirstId và LastId.




***Cài đặt minh họa thuật toán tô màu scan-line***

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <graphics.h>
#include <dos.h>
#define MAXVERTEX       20
#define MAXEDGE         20
#define TRUE            1
#define FALSE           0

typedef struct {
    int x;
    int y;
}POINT;

typedef struct{
    int             NumVertex;
    POINTaVertex[MAXVERTEX];
}POLYGON;

typedef struct {
    int NumPt;
    float xPt[MAXEDGE];
}XINTERSECT;

typedef struct
{
    int     yMin; // Gia tri y nho nhat cua 2 dinh
```

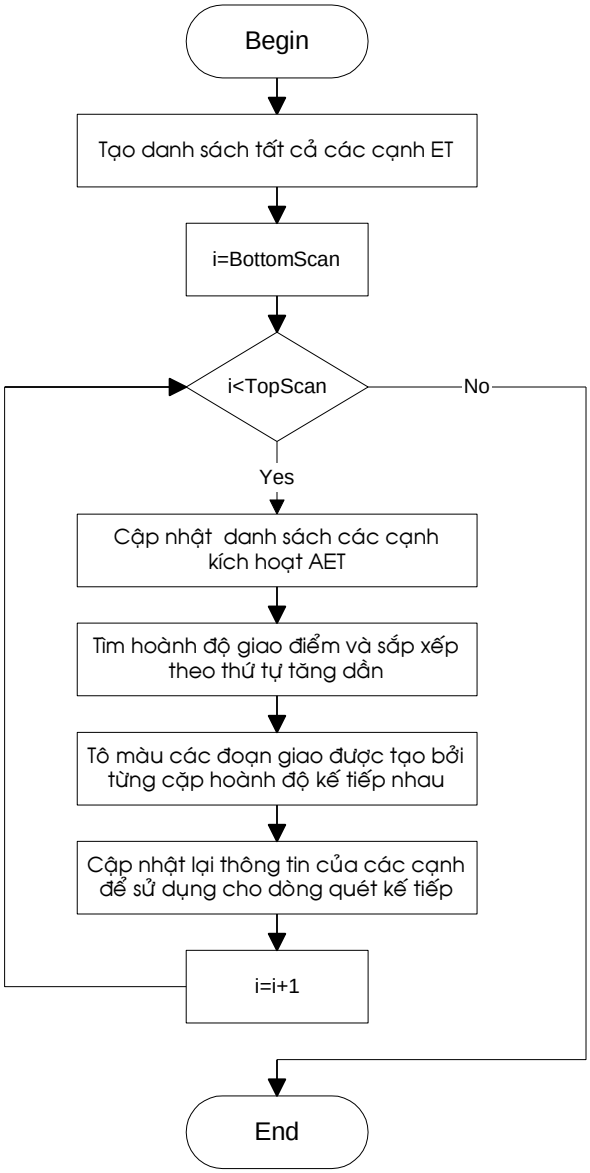
```
    float   xIntersect; // Hoanh do giao diem cua canh & dong quet
    float   dxPerScan; // Gia tri 1/m
    int     DeltaY;
}EDGE;

typedef struct
{
    int     NumEdge;
    EDGE   aEdge[MAXEDGE];
}EDGELIST;

/*
    Dat 1 canh vao danh sach canh.
    Cac canh duoc sap theo thu tu giam dan cua yMin (yMin la gia tri y lon nhat cua 2 dinh 1 canh)
    Xu li luon truong hop dong quet di ngang qua dinh ma tai do chi tinh 1 diem giao
*/
void PutEdgeInList(EDGELIST &EdgeList, POINT p1, POINT p2,
                   int NextY)
{
    EDGE EdgeTmp;

    EdgeTmp.dxPerScan = float(p2.x-p1.x)/(p2.y-p1.y); // 1/m
    if(p1.y < p2.y)
    {
    /*
        Truong hop dong quet di ngang qua dinh la giao diem
        cua 2 canh co huong y cung tang
    */
        if(p2.y < NextY)
        {
            p2.y--;
            p2.x -= EdgeTmp.dxPerScan;
        }
        EdgeTmp.yMin = p1.y;
        EdgeTmp.xIntersect= p1.x;
        EdgeTmp.DeltaY = abs(p2.y-p1.y)+1;
    } // if
    else
    {
    /*
        Truong hop dong quet di ngang qua dinh la giao diem cua 2 canh co huong y cung giam
    */
        if(p2.y > NextY)
        {
            p2.y++;
            p2.x+= EdgeTmp.dxPerScan;
```

```
            }
            EdgeTmp.yMin = p2.y;
            EdgeTmp.xIntersect= p2.x;
            EdgeTmp.DeltaY = abs(p2.y-p1.y)+1;
        }//else
        // xac dinh vi tri chen
        int j = EdgeList.NumEdge;
        while((j>0) && (EdgeList.aEdge[j-1].yMin>EdgeTmp.yMin))
        {
            EdgeList.aEdge[j] = EdgeList.aEdge[j-1];
            j--;
        }
        // tien hanh chen dinh moi vao canh
        EdgeList.NumEdge++;
        EdgeList.aEdge[j] = EdgeTmp;
} // PutEdgeInList
/*
    Tim dinh ke tiep sao cho khong nam tren cung duong thang voi dinh dang xet
*/
int FindNextY(POLYGON P, int id)
{
    int j = (id+1)%P.NumVertex;
    while((j<P.NumVertex)&&(P.aVertex[id].y == P.aVertex[j].y))
        j++;
    if(j<P.NumVertex)
        return (P.aVertex[j].y);
    return 0;
} // FindNextY

// Tao danh sach cac canh tu polygon da cho
void MakeSortedEdge(POLYGON P, EDGELIST &EdgeList,
                    int &TopScan, int &BottomScan)
{
    TopScan = BottomScan = P.aVertex[0].y;
    EdgeList.NumEdge = 0;
    for(int i=0; i<P.NumVertex; i++)
    {
        // Truong hop canh khong phai la canh nam ngang        if(P.aVertex[i].y != P.aVertex[i+1].y)
        PutEdgeInList(EdgeList, P.aVertex[i], P.aVertex[i+1], FindNextY(P, i+1));
        //else Xu li truong hop  canh nam ngang
            if(P.aVertex[i+1].y > TopScan)
                TopScan = P.aVertex[i+1].y;
    }
    BottomScan = EdgeList.aEdge[0].yMin;
} //MakeSortedEdge
// Cap nhat lai hai con tro FirstId, LastId cho biet danhsach cac canh active
```

```
void UpdateActiveEdgeList(EDGELIST EdgeList, int yScan, int &FirstId, int &LastId)
{
    while((FirstId<EdgeList.NumEdge-1) &&(EdgeList.aEdge[FirstId].DeltaY == 0))
        FirstId++;
    while((LastId<EdgeList.NumEdge-1) &&(EdgeList.aEdge[LastId+1].yMin<=yScan))
        LastId++;
} // UpdateActiveEdgeList
void SortOnX(XINTERSECT & aIntersectPt)
{
    for(int i=0; i<aIntersectPt.NumPt-1; i++)
    {
        int Min = i, t;
        for(int j=i+1; j<aIntersectPt.NumPt; j++)
            if( aIntersectPt.xPt[j] < aIntersectPt.xPt[Min])
                Min = j;
        t = aIntersectPt.xPt[Min];
        aIntersectPt.xPt[Min]      = aIntersectPt.xPt[i];
        aIntersectPt.xPt[i]   = t;
    }
} // SortOnX
/*
    Tim cac hoanh do giao diem cua cac canh cua da giac voi dong quet yScan. Sau khi tim cac hoanh do giao diem, ta sap xep lai
    theo chieu tang cua x
*/
void  FindXIntersection(EDGELIST EdgeList, XINTERSECT &aIntersectPt, int FirstId, int LastId)
{
    aIntersectPt.NumPt = 0;
    for(int i=FirstId; i<=LastId; i++)
    {
        if(EdgeList.aEdge[i].DeltaY>0)
        {
            aIntersectPt.xPt[aIntersectPt.NumPt] = EdgeList.aEdge[i].xIntersect;
            aIntersectPt.NumPt++;
        }
    }
    SortOnX(aIntersectPt);
} //FindXIntersection
#define Round(x) int(x+0.5)
void FillLine(XINTERSECT aIntersectPt, int yScan)
{
    for(int i=0; i<aIntersectPt.NumPt; i+=2)
        line(Round(aIntersectPt.xPt[i]), yScan, Round(aIntersectPt.xPt[i+1]), yScan);
} // FillLine

void UpdateEdgeList(EDGELIST &EdgeList,int FirstId,int LastId)
{
```

```
        for(int i=FirstId; i<=LastId; i++)
        {
                if(EdgeList.aEdge[i].DeltaY>0)
                {
                        EdgeList.aEdge[i].DeltaY--;
                        EdgeList.aEdge[i].xIntersect += EdgeList.aEdge[i].dxPerScan;
                }
        }
} //FillLine
void  ScanLineFill(POLYGON P)
{
        EDGELIST        EdgeList;
        XINTERSECT aIntersectPt;
        int TopScan,  BottomScan, FirstId, LastId;

        MakeSortedEdge(P, EdgeList, TopScan, BottomScan);
        FirstId = LastId = 0;
        for(int i=BottomScan; i<=TopScan; i++)
        {
                // Cap nhat lai danh sach cac canh active - tuc la cac canh cat dong quet i
                UpdateActiveEdgeList(EdgeList, i, FirstId, LastId);
                // Tim cac hoanh do giao diem cua dong quet voi cac canh cua da giac va sap xep lai cac hoanh do giao diem truc tiep tren
                        EdgeList
                FindXIntersection(EdgeList, aIntersectPt, FirstId, LastId);
                FillLine(aIntersectPt, i);
                UpdateEdgeList(EdgeList, FirstId, LastId);
        }
} //ScanLineFill
```

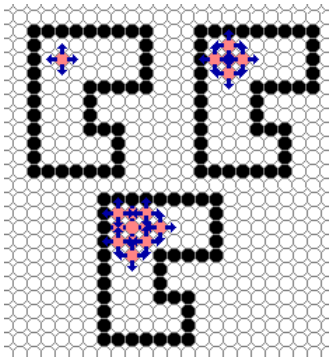
***Lưu đồ thuật toán tô màu theo dòng quét***




## 6.2. Thuật toán tô màu dựa theo đường biên

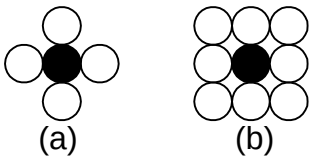
Khác với thuật toán tô màu dựa theo dòng quét, đường biên của vùng tô được xác định bởi tập các đỉnh của một đa giác, đường biên trong thuật toán được mô tả bằng một giá trị duy nhất đó là màu của tất cả các điểm thuộc về đường biên.

Bắt đầu từ điểm nằm bên trong vùng tô, ta sẽ kiểm tra các điểm lân cận của nó đã được tô màu hay có phải là điểm biên hay không, nếu không phải là điểm đã tô và không phải là điểm biên ta sẽ tô màu nó. Quá trình này được lặp lại cho tới khi nào không còn tô được điểm nào nữa thì dừng. Bằng cách này, toàn bộ các điểm thuộc vùng tô được kiểm tra và sẽ được tô hết.



**Hình 2.24** – Thuật toán tô màu dựa theo đường biên

Có hai quan điểm về cách tô này, đó là dùng bốn điểm lân cận hay tám điểm lân cận đối với điểm đang xét được tô bằng màu trắng *(xem hình 2.25)*.




**Hình 2.25** – 4 điểm lân cận (a) và 8 điểm lân cận (b)

Đoạn chương trình sau minh họa cài đặt thuật toán tô màu dựa theo đường biên sử dụng phương pháp tô 4 điểm lân cận.

***Cài đặt minh họa thuật toán tô màu dựa theo đường biên***

```
void BoundaryFill(int x, int y, int FillColor, int BoundaryColor)
{
```

```
    int CurrentColor;

    CurrentColor = getpixel(x,y);

    if((CurrentColor!=BoundaryColor)&&CurrentColor!= FillColor))
    {
        putpixel(x,y,FillColor);
        BoundaryFill(x-1, y, FillColor, BoundaryColor);
        BoundaryFill(x, y+1, FillColor, BoundaryColor);
        BoundaryFill(x+1, y, FillColor, BoundaryColor);
        BoundaryFill(x, y-1, FillColor, BoundaryColor);
    }

} // Boundary Fill
```

***Nhận xét***

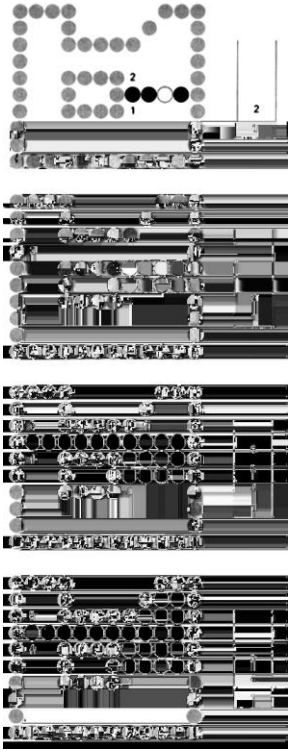
- Thuật toán này có thể sẽ không hoạt động chính xác khi có một số điểm nằm trong vùng tô có màu là màu cần tô của vùng (FillColor). Để khắc phục điều này, trước khi tô màu cần phải đảm bảo rằng toàn bộ các điểm thuộc về vùng tô có màu khác màu tô.
- Nhận xét rằng trong cài đặt thuật toán ở trên, việc gọi thực hiện đệ quy thuật toán cho bốn điểm lân cận của điểm hiện hành không quan tâm tới một trong bốn điểm đó đã được xét ở bước trước hay chưa. Ví dụ khi ta xét bốn điểm lân cận của điểm hiện hành (x,y), thì khi gọi thực hiện đệ quy với điểm hiện hành là một trong bốn điểm lân cận trên, (x,y) vẫn được xem là điểm lân cận của chúng và lại được gọi thực hiện lại. Ta sẽ đưa ra một cải tiến nhỏ để khắc phục điểm này, bằng cách mỗi lần xét điểm hiện hành (x,y) ta sẽ gọi 4 thủ tục riêng để tô các điểm lân cận và trong 4 thủ tục này ta sẽ tránh gọi lại việc xét điểm (x,y).

```
void BoundaryFillEnhanced(int x, int y, int F_Color, int B_Color)
{
    int CurrentColor;

    CurrentColor = getpixel(x,y);
    if((CurrentColor!=B_Color)&&CurrentColor!= F_Color))
    {
        putpixel(x,y,F_Color);
        FillLeft(x-1, y, F_Color, B_Color);
        FillTop(x, y+1, F_Color, B_Color);
        FillRight(x+1, y, F_Color, B_Color);
        FillBottom(x, y-1, F_Color, B_Color);
    }

} // BoundaryFillEnhanced

void FillLeft(int x, int y, int F_Color, int B_Color)
{
    int CurrentColor;
    CurrentColor = getpixel(x,y);
    if((CurrentColor!=B_Color)&&CurrentColor!= F_Color))
    {
        putpixel(x,y,F_Color);
```

```
            FillLeft(x-1, y, F_Color, B_Color);
            FillTop(x, y+1, F_Color, B_Color);
            FillBottom(x, y-1, F_Color, B_Color);
        }
} // FillLeft

void FillTop(int x, int y, int F_Color, int B_Color)
{
        int CurrentColor;

        CurrentColor = getpixel(x,y);
        if((CurrentColor!=B_Color)&&CurrentColor!= F_Color))
        {
            putpixel(x,y,F_Color);
            FillLeft(x-1, y, F_Color, B_Color);
            FillTop(x, y+1, F_Color, B_Color);
            FillRight(x+1, y, F_Color, B_Color);
        }

} // FillTop

void FillRight(int x, int y, int F_Color, int B_Color)
{
        int CurrentColor;

        CurrentColor = getpixel(x,y);
        if((CurrentColor!=B_Color)&&CurrentColor!= F_Color))
        {
            putpixel(x,y,F_Color);
            FillTop(x, y+1, F_Color, B_Color);
            FillRight(x+1, y, F_Color, B_Color);
            FillBottom(x, y-1, F_Color, B_Color);
        }

} // FillRight

void FillBottom(int x, int y, int F_Color, int B_Color)
{
        int CurrentColor;

        CurrentColor = getpixel(x,y);
        if((CurrentColor!=B_Color)&&CurrentColor!= F_Color))
        {
            putpixel(x,y,F_Color);
            FillLeft(x-1, y, F_Color, B_Color);
```

```
        FillRight(x+1, y, F_Color, B_Color);
        FillBottom(x, y-1, F_Color, B_Color);
    }
} // FillBottom
```

- Thuật toán này có tính đệ quy, do đó khi cài đặt thường gây lỗi tràn bộ nhớ khi vùng tô khá lớn, do đó để cải tiến chúng ta sẽ tiến hành loang dần và lần lượt tô từng đoạn giao theo dòng quét ngang thay vì tô theo 4 điểm lân cận. Như vậy chúng ta chỉ cần lưu lại thông tin của điểm bắt đầu mỗi đoạn giao của dòng quét ngang thay vì phải lưu hết tất cả các điểm lân cận chưa được tô của điểm hiện hành. Chúng ta sẽ cho các dòng quét loang từ điểm bắt đầu theo hướng lên biên trên, sau khi đã tô xong, các dòng quét còn lại theo hướng xuống biên dưới sẽ được tô. Ứng với mỗi dòng quét ngang, ta sẽ loang và tìm pixel trái nhất (có hoành độ nhỏ nhất) để lưu lại. Trong hình 2.26, đoạn giao đầu tiên chứa điểm bắt đầu (tô màu trắng) sẽ được tô trước. Sau đó các vị trí 1, 2 ứng với các đoạn giao của các dòng quét kế tiếp sẽ được lưu lại (hình 2.26a). Bước tiếp theo, điểm ứng với vị trí 2 sẽ được lấy ra và tiến hành tô màu bằng cách loang từ điểm này ra theo chiều ngang, sau đó pixel ứng vị trí 3 của dòng quét kế tiếp sẽ được lưu lại (hình 2.26b). Sau khi dòng quét ứng với điểm 3 đã được xử lí tương tự như trên xong, stack lưu các vị trí của các điểm "hạt giống" cho các dòng quét kế tiếp như trong hình 2.26c. Hình 2.26d minh họa khi thuật toán đã tô được toàn bộ một phần vùng phía trên bên phải của vùng tô. Khi pixel ứng với vị trí 5 được xử lí xong, ta có phần còn lại phía trên bên trái sẽ được tô. Sau đó pixel ứng với vị trí 4 sẽ được xử lí, các dòng quét phía dưới sẽ được tô tiếp theo.




**Hình 2.26** – Thuật toán tô màu theo dòng quét cải tiến

## TÓM TẮT

Các đối tượng đồ họa cơ sở cung cấp các công cụ cơ bản nhất cho việc xây dựng các ảnh đồ họa của các đối tượng phức tạp. Các đoạn thẳng, đường cong, vùng tô, kí tự, … là các đối tượng đồ họa cơ sở được hầu hết tất cả các công cụ lập trình đồ họa hỗ trợ.

Mỗi đối tượng đồ họa cơ sở được mô tả thông qua dữ liệu về tọa độ và các thuộc tính của nó. Hệ tọa độ thường được dùng để mô tả đối tượng là hệ tọa độ Descartes. Các thuộc tính của đối tượng như màu sắc, kiểu, độ rộng, … cho biết kiểu cách mà đối tượng được hiển thị trên thiết bị.

Để có thể hiển thị các đối tượng đồ họa trên thiết bị hiển thị dạng điểm mà điển hình là màn hình, cần phải có một quá trình chuyển các mô tả hình học của các đối tượng này trong hệ tọa độ thế giới thực về dãy các pixel tương ứng gần với chúng nhất trên hệ tọa độ của thiết bị. Quá trình này còn được gọi là quá trình chuyển đổi bằng dòng quét. Yêu cầu quan trọng nhất đối với quá trình này ngoài việc phải cho kết quả xấp xỉ tốt nhất còn phải cho tốc độ tối ưu.

Ba cách tiếp cận để vẽ đoạn thẳng gồm thuật toán DDA, thuật toán Bresenham, thuật toán MidPoint đều tập trung vào việc đưa ra cách chọn một trong hai điểm nguyên kế tiếp khi đã biết được điểm nguyên ở bước trước. Thuật toán DDA đơn giản chỉ dùng thao tác làm tròn nên phải dùng các phép toán trên số thực, trong khi đó thuật toán Bresenham và thuật toán MidPoint đưa ra cách chọn phức tạp hơn nhưng cho kết quả tốt hơn. Đối với trường hợp vẽ đoạn thẳng, hai thuật toán Bresenham và thuật toán MidPoint cho kết quả giống nhau và tốc độ tối ưu.

Các đối tượng khác như đường tròn, ellipse và các đường conics khác cũng được vẽ tương tự bằng cách sử dụng thuật toán MidPoint. Riêng với các đường phức tạp hơn như đường spline, sẽ được xây dựng từ các đoạn thẳng xấp xỉ với đường cong thay vì phải xấp xỉ chúng từ các điểm *(xem phần sau)*.

Các thuật toán tô màu các vùng tô thường chia làm hai công đoạn : công đoạn thứ nhất là xác định các điểm nào để tô và công đoạn còn lại đơn giản hơn đó là quyết định tô các điểm đó bằng giá trị màu nào. Công đoạn thứ hai chỉ thực sự phức tạp nếu ta tô theo một mẫu tô nào đó không phải là tô thuần một màu. Có hai cách tiếp cận chính để tô màu một vùng tô đối với thiết bị hiển thị dạng điểm đó là : tô theo dòng quét và tô dựa theo đường biên. Cách tô theo dòng quét thường được dùng để tô màu các đa giác, đường tròn, ellipse, và một số đường cong đơn giản khác, còn cách tô theo đường biên thường được dùng cho các vùng tô có dạng đường biên phức tạp hơn.

Thuật toán tô màu đa giác theo dòng quét xác định các điểm thuộc vùng tô bằng cách xác định phần giao của các dòng quét với các đoạn thẳng biên của đa giác. Điểm độc đáo của thuật toán này ở chỗ đưa ra cấu trúc dữ liệu danh sách các cạnh kích hoạt AET và cách hoạt động của chúng để có thể hạn chế tối đa các cạnh cần tìm giao điểm ứng với mỗi dòng quét. Đây là điểm mấu chốt trong vấn đề cải thiện tốc độ của thuật toán. Thuật toán này có thể được áp dụng cho nhiều dạng đa giác khác nhau như đa giác lồi, đa giác lõm, và cả đa giác tự cắt, …

Thuật toán tô màu dựa theo đường biên xuất phát từ điểm nằm bên trong vùng tô và tiến hành loang dần ra các điểm lân cận cho tới khi gặp các điểm thuộc biên thì dừng. Cách làm này gặp hạn chế về bộ nhớ khi cài đặt bằng đệ quy. Một phương pháp cải tiến đã được đề cập đó là loang theo từng dòng quét. Việc tô màu theo cách này thực sự là thuận tiện cho các chương trình đồ họa ứng dụng có khả năng tương tác cao.

## BÀI TẬP

**1.** Thiết kế và cài đặt hàm vẽ hình chữ nhật, đường gấp khúc, đa giác từ hàm vẽ đoạn thẳng.

**2.** Trong phần trình bày thuật toán Bresenham để vẽ đường thẳng, hãy cho biết với cách đặt $d_1$, $d_2$ như vậy, có khi nào $d_1$, $d_2$ lấy giá trị âm hay không ? Nếu có hãy cho ví dụ minh họa.

**3.** Tại sao phải so sánh $p_i$ với giá trị 0 trong các thuật toán Bresenham, MidPoint. Bản chất của việc so sánh này là gì?

**4.** Cài đặt các thuật toán DDA, Bresenham, MidPoint vẽ đoạn thẳng qua hai điểm cho trước trong trường hợp tổng quát với hệ số góc m lấy giá trị bất kì.

**5.** Người ta có thể cải thiện tốc độ cài đặt thuật toán vẽ đoạn thẳng bằng cách chỉ cần vẽ một nửa đoạn thẳng, phần còn lại lấy

đối xứng nửa đoạn thẳng đã vẽ. Hãy cài đặt minh họa.

**6.** Cho biết các điểm nguyên vẽ phát sinh khi sử dụng các thuật toán DDA, MidPoint cho các đoạn thẳng đi qua các điểm lần lượt là $A_1(5,10)$, $B_1(15,17)$; $A_2(-2,3)$, $B_2(-12,7)$; $A_3(6,3)$, $B_3(9,13)$; $A_4(2,4)$, $B_4(-5,14)$; $A_5(0,10)$, $B_5(15,10)$; $A_6(5,-1)$, $B_6(5,-11)$;

**7.** Trình bày thuật toán MidPoint vẽ cung tròn 1/8, bán kính R, tâm $I(x_C, y_C)$ và được giới hạn bởi :

$$\begin{cases} R\frac{\sqrt{2}}{2} \le x \le R \\ 0 \le y \le R\frac{\sqrt{2}}{2} \end{cases}$$

**8.** Sử dụng ý tưởng của thuật toán Bresenham, xây dựng thuật toán vẽ đường tròn có tâm là gốc tọa độ, bán kính R.

**9.** Giải thích tại sao chỉ chọn cung 1/8 để vẽ rồi lấy đối xứng mà không mở rộng cho cung 1/16 hay 1/32.

**10.** Giải thích tại sao có thể thay công thức $p_0 = 5/4 - R$ bằng công thức $p_0 = 1- R$ khi cài đặt thuật toán MidPoint vẽ đường tròn.

**11.** Trình bày thuật toán Bresenham vẽ đường tròn bán kính R, từ đó nhận xét về cách tiếp cận của thuật toán MidPoint có gì lợi hơn so với thuật toán Bresenham.

**12.** Xây dựng và cài đặt thuật toán vẽ ellipse có tâm là gốc tọa độ với bán kính trục chính, bán kính trục phụ lần lượt là A, B.

**13.** Dựa vào thuật toán vẽ đường tròn để xây dựng thủ tục vẽ một cung tròn (arc) tâm (x,y) bán kính R, biết góc bắt đầu và kết thúc của cung lần lượt là $\alpha$, $\beta$..

**14.** Dựa vào thuật toán vẽ ellipse để xây dựng thủ tục vẽ một cung (pie slice) tâm (x,y) và bán kính trục chính, trục phụ lần lượt là A, B, góc bắt đầu và kết thúc của cung lần lượt là $\alpha$, $\beta$.

**15.** Hãy tìm hiểu các cài đặt tối ưu hơn cho các thuật toán vẽ đoạn thẳng và vẽ đường tròn, ellipse.

**16.** Xây dựng và cài đặt thuật toán vẽ các parabol $y = \frac{x^2}{A}$, và $y^2 = Ax$ với A là số nguyên bất kì.

**17.** Xây dựng và cài đặt thuật toán vẽ các hyperbol $\frac{x^2}{A^2} - \frac{y^2}{B^2} = 1$, và $\frac{x^2}{A^2} - \frac{y^2}{B^2} = -1$ với A, B là các số nguyên bất kì.

**18.** Xây dựng và cài đặt thuật toán vẽ các đường cong sau :

- $y = A\sin(\omega x + \theta)$, với A nguyên.
- $y = A\cos(\omega x + \theta)$, với A nguyên.
- $xy = A^2$, với A nguyên.
- $y = x^3 / A$, với A nguyên.
- $x^2 + y^2 + 4x + 2y - 4 = 0$
- $9x^2 + 25y^2 - 18x - 216 = 0$
- $100y + 100x - x^2 = 0$
- $xy + 5x - 2y - 74 = 0$
- $100x^2 - 25y^2 + 2500 = 0$

**19.** Các bước chính của các thuật toán vẽ đường dạng $y = f(x)$. Minh họa cho các trường hợp vẽ đường thẳng, đường tròn.

**20.** Bản chất của quá trình vẽ các đường đơn giản theo từng điểm là rời rạc hóa và nguyên hóa. Hãy cho biết lí do tại sao, bước nào trong thuật toán tổng quát thể hiện hai ý trên. Minh họa bằng các đường đã học.

**21.** Các thuật toán vẽ đường bao hàm rất lớn kĩ thuật tối ưu hóa chương trình. Hãy minh họa qua các đối tượng đã học.

**22.** Ý nghĩa của danh sách kích hoạt AET trong thuật toán tô màu đa giác theo dòng quét. Cấu trúc dữ liệu và nguyên tắc hoạt động của AET.

**23.** Cài đặt thuật toán tô màu đa giác theo dòng quét bằng cách dùng xâu liên kết thay vì dùng mảng như cài đặt minh họa.

**24.** Cài đặt thuật toán tô màu theo đường biên không dùng đệ quy.

**25.** Xây dựng và cài đặt thuật tô màu đường tròn, ellipse.

*CHƯƠNG 3*

# CÁC PHÉP BIẾN ĐỔI TRONG ĐỒ HỌA HAI CHIỀU

Một trong những ưu điểm quan trọng của đồ họa là cho phép dễ dàng thao tác lên các đối tượng đã được tạo ra. Một nhà quản lí có nhu cầu thu nhỏ các biểu đồ trong một báo cáo, một kiến trúc sư muốn nhìn tòa nhà ở những góc nhìn khác nhau, một nhà thiết kế muốn quan sát và chỉnh sửa các mẫu đối tượng trong quá trình thiết kế, … Tất cả các thao tác này có thể được hỗ trợ một cách dễ dàng nhờ vào các phép biến đổi hình học. Các phép biến đổi hình học sẽ làm thay đổi mô tả về tọa độ của các đối tượng, từ đó làm cho đối tượng bị thay đổi về hướng, kích thước và hình dạng. Các phép biến đổi hình học cơ sở bao gồm : tịnh tiến (translation), quay (rotation) và biến đổi tỉ lệ (scaling). Ngoài ra một số phép biến đổi khác cũng thường được áp dụng đó là phép đối xứng (reflection) và biến dạng (shearing).

Có hai quan điểm về phép biến đổi hình học đó là : biến đổi đối tượng (object transformation) và biến đổi hệ tọa độ (coordinate transformation). Biến đổi đối tượng là thay đổi tọa độ của các điểm mô tả nó theo một quy tắc nào đó, còn biến đổi hệ tọa độ là tạo ra một hệ tọa độ mới và tất cả các điểm mô tả đối tượng sẽ được chuyển về hệ tọa độ mới. Hai cách này có những mối liên hệ chặt chẽ với nhau và mỗi cách đều có những lợi thế riêng. Chúng ta sẽ bàn về phép biến đổi đối tượng trước.

## 1. CÁC PHÉP BIẾN ĐỔI HÌNH HỌC CƠ SỞ

Một phép biến đổi hai chiều sẽ biến đổi điểm P trong mặt phẳng thành điểm có tọa độ mới Q theo một quy luật nào đó. Về mặt bản chất, một phép biến đổi điểm là một ánh xạ T được định nghĩa :

$$T : \mathbf{R}^2 \to \mathbf{R}^2$$
$$P(x, y) \mapsto Q(x', y')$$

Nói cách khác, T là hàm số $T(x, y)$ theo hai biến $(x, y)$:

$$\begin{cases} x' = f(x, y) \\ y' = g(x, y) \end{cases}$$

Phép biến đổi affine là phép biến đổi với $f(x, y)$ và $g(x, y)$ là các hàm tuyến tính. Phép biến đổi này có dạng :

$$\begin{cases} x' = ax + cy + e \\ y' = bx + dy + f \end{cases} \quad , a, b, c, d, e, f \in R, ad - bc \neq 0 .$$

Ta chỉ khảo sát các phép biến đổi affine nên từ nay về sau ta dùng cụm từ "phép biến đổi" thay cho "phép biến đổi affine".

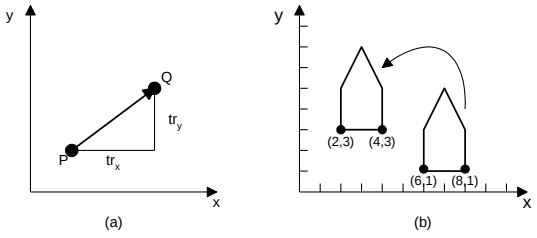
### 1.1. Phép tịnh tiến

Để tịnh tiến một điểm $P(x, y)$ từ vị trí này sang vị trí khác trong mặt phẳng, ta cộng thêm các giá trị mô tả độ dời vào các tọa độ của P. Nếu gọi $tr_x$ và $tr_y$ lần lượt là độ dời theo trục hoành và trục tung thì tọa độ của điểm mới $Q(x', y')$ sẽ là :

$$\begin{cases} x' = x + tr_x \\ y' = y + tr_y \end{cases},$$

$(tr_x, tr_y)$ còn được gọi là vector tịnh tiến hay vector độ dời.

Chúng ta có thể dịch chuyển toàn bộ một đối tượng bằng cách áp dụng quy tắc trên cho mọi điểm thuộc đối tượng. Để tịnh tiến một đoạn thẳng, đơn giản chỉ cần tịnh tiến hai điểm đầu và cuối của nó rồi sau đó vẽ lại đoạn thẳng nối hai điểm mới. Với đa giác, ta tịnh tiến các đỉnh của nó sau đó vẽ lại đa giác với các đỉnh mới. Một cách tương tự, để tịnh tiến các đối tượng như đường tròn, ellipse, ta tịnh tiến tâm của chúng tới vị trí mới rồi vẽ lại.




**Hình 3.1** – Phép tịnh tiến một điểm (a) và đối tượng với vector tịnh tiến (-4,2) (b)

### 1.2. Phép biến đổi tỉ lệ

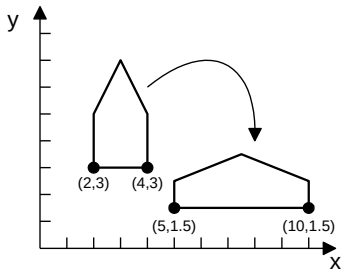
Phép biến đổi tỉ lệ làm thay đổi kích thước đối tượng. Để co hay giãn tọa độ của một điểm $P(x, y)$ theo trục hoành và trục tung lần lượt là $s_x$ và $s_y$, ta nhân $s_x$ và $s_y$ lần lượt cho các tọa độ của P.

$$\begin{cases} x' = s_x . x \\ y' = s_y . y \end{cases} \quad , s_x \text{ và } s_y \text{ được gọi là các hệ số tỉ lệ.}$$

Khi các giá trị $s_x$ , $s_y$ nhỏ hơn 1, phép biến đổi sẽ thu nhỏ đối tượng, ngược lại khi các giá trị này lớn hơn 1, phép biến

đổi sẽ phóng lớn đối tượng. Khi $s_x$ , $s_y$ bằng nhau, ta gọi đó là phép đồng dạng (uniform scaling), phép đồng dạng là phép biến đổi bảo toàn tính cân xứng của đối tượng.

Tâm tỉ lệ là điểm không bị thay đổi qua phép biến đổi tỉ lệ. Phép biến đổi tỉ lệ mô tả như trên còn gọi là phép biến đổi tỉ lệ quanh gốc tọa độ vì có tâm tỉ lệ là gốc tọa độ. Nhận xét rằng khi phép biến đổi tỉ lệ thu nhỏ đối tượng, đối tượng sẽ được dời về gần gốc tọa độ hơn, tương tự khi phóng lớn đối tượng, đối tượng sẽ được dịch chuyển xa gốc tọa độ hơn.

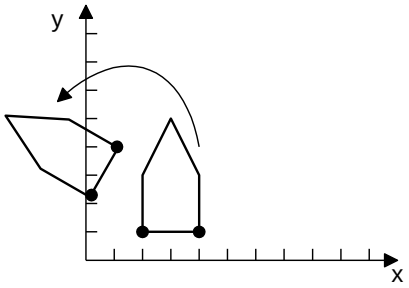



**Hình 3.2** – Phép biến đổi tỉ lệ với $s_x = 2.5$ và $s_y = 0.5$

### 1.3. Phép quay

Phép quay làm thay đổi hướng của đối tượng. Một phép quay đòi hỏi phải có tâm quay, góc quay. Góc quay dương thường được quy ước là chiều ngược chiều kim đồng hồ. Ta có công thức biến đổi của phép quay điểm $P(x, y)$ quanh gốc tọa độ một góc $\alpha$ :

$$\begin{cases} x' = \cos\alpha.x - \sin\alpha.y \\ y' = \sin\alpha.x + \cos\alpha.y \end{cases}$$




**Hình 3.3** – Phép quay một đối tượng quanh gốc tọa độ một góc $60^0$

### 1.4. Biểu diễn ma trận của phép biến đổi

Trong nhiều ứng dụng đồ họa, người dùng thường xuyên có nhu cầu thực hiện nhiều phép biến đổi hình học khác nhau trên một đối tượng để tạo ra các hiệu quả như mong muốn. Ví dụ trong các ứng dụng thiết kế, chúng ta cần phải thực hiện nhiều phép tịnh tiến, quay, tỉ lệ để có thể khớp từng phần của đối tượng vào đúng vị trí của chúng, hay sau khi thực hiện các phép biến đổi nhưng không được ưng ý, người dùng muốn trở lại hiện trạng trước khi biến đổi (undo), … Do đó cần phải có một cách nào đó để có thể xử lí dãy các phép biến đổi trên được nhanh chóng và hiệu quả.

Nếu ta biểu diễn tọa độ của điểm $P(x, y)$ và $Q(x', y')$ dưới dạng các vector dòng lần lượt là $(x \quad y)$ và $(x' \quad y')$thì các phép biến đổi tịnh tiến, tỉ lệ, quay có thể được biểu diễn dưới dạng ma trận như sau :

***Phép tịnh tiến***

$$(x' \quad y') = (x \quad y) + (tr_x \quad tr_y)$$

hay $Q = P + T$ với $T = (tr_x \quad tr_y)$

***Phép biến đổi tỉ lệ***

$$(x' \quad y') = (x \quad y)\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

hay $Q = P.S$ với $S = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$

***Phép quay quanh gốc tọa độ***

$$(x' \quad y') = (x \quad y)\begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

hay $Q = P.R$ với $R = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$

Với cách biểu diễn này, chúng ta sẽ gặp khó khăn khi muốn kết hợp các phép biến đổi lại với nhau vì biểu diễn của phép tịnh tiến khác với dạng của các phép biến đổi tỉ lệ và quay. Chính vì vậy mà cần phải có một cách nào đó để biểu diễn ba phép biến đổi này về một dạng duy nhất để có thể dễ dàng xử lí sau này.

#### 1.4.1. Hệ tọa độ thuần nhất (hormogeneous coordinates)

Tọa độ thuần nhất của một điểm trên mặt phẳng được biểu diễn bằng bộ ba số tỉ lệ $(x_h, y_h, h)$ không đồng thời bằng 0 và liên hệ với các tọa độ $(x, y)$ của điểm đó bởi công thức :

$$x = \frac{x_h}{h}, \qquad y = \frac{y_h}{h}$$

Nếu một điểm có tọa độ thuần nhất là $(x, y, z)$ thì nó cũng có tọa độ thuần nhất là $(h.x, h.y, h.z)$ trong đó h là số thực khác 0 bất kì. Tọa độ thuần nhất của một điểm trong không gian ba chiều hay có số chiều lớn hơn cũng được xác định một cách tương tự.

Về mặt toán học, việc đưa tọa độ thuần nhất vào là do sự cần thiết phải bổ sung cho mặt phẳng Euclid các điểm xa vô tận $(x, y, 0)$ (điểm phi chính) có tọa độ thứ ba bằng 0, điều này dẫn đến khái niệm mặt phẳng xạ ảnh trong hình học xạ ảnh. Trong hệ tọa độ thuần nhất, các điểm xa vô tận không đóng một vai trò gì đặc biệt so với các điểm khác của mặt phẳng. Với các phép biến đổi hình học đang khảo sát, nếu một điểm được biểu diễn dưới dạng tọa độ thuần nhất, cả ba phép biến đổi trên đều được biểu diễn dưới dạng tích các ma trận. Điều này giúp cho việc khảo sát các tính chất và sự kết hợp của các phép biến đổi này được thuận tiện do mỗi phép biến đổi được đại diện bởi một ma trận duy nhất.

Bộ ba các tọa độ thường biểu diễn các điểm trong không gian ba chiều, nhưng ở đây ta sử dụng chúng để biểu diễn các điểm trong không gian hai chiều. Mối liên hệ ở đây là : nếu chúng ta xét tất cả các bộ ba tọa độ thuần nhất biểu diễn cho cùng một điểm, nghĩa là bộ ba số có dạng $(h.x, h.y, h.)$, với $h \neq 0$, chúng ta sẽ nhận được một đường thẳng trong không gian ba chiều. Để đơn giản hóa chúng ta có thể chọn $h = 1$, lúc này mỗi điểm $P(x, y)$ sẽ được biểu diễn dưới dạng tọa độ thuần nhất là $(x, y, 1)$.

### 1.4.2. Biểu diễn các phép biến đổi dưới dạng tọa độ thuần nhất

***Phép tịnh tiến***

$$(x' \quad y' \quad 1) = (x \quad y \quad 1).\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_x & tr_y & 1 \end{pmatrix}$$

hay $Q = P.M_T(tr_x, tr_y)$ với $M_T(tr_x, tr_y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_x & tr_y & 1 \end{pmatrix}$

***Phép biến đổi tỉ lệ***

$$(x' \quad y' \quad 1) = (x \quad y \quad 1).\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

hay $Q = P.M_S(s_x, s_y)$ với $M_S(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

***Phép quay quanh gốc tọa độ***

$$(x' \quad y' \quad 1) = (x \quad y \quad 1).\begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

hay $Q = P.M_R(\alpha)$ với $M_R(\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$

## 2. KẾT HỢP CÁC PHÉP BIẾN ĐỔI

Quá trình áp dụng các phép biến đổi liên tiếp để tạo nên một phép biến đổi tổng thể được gọi là sự kết hợp các phép biến đổi (composing transformation).

### 2.1. Kết hợp các phép tịnh tiến

Nếu ta thực hiện phép tịnh tiến lên $P(x, y)$ được P' , rồi lại thực hiện tiếp một phép tịnh tiến khác lên P', ta được điểm $Q(x', y')$. Như vậy, Q là ảnh của phép biến đổi kết hợp hai phép tịnh tiến liên tiếp $M_{T1}(tr_{x1}, tr_{y1})$ và $M_{T2}(tr_{x2}, tr_{y2})$ có tọa độ :

$$Q = \{P.M_{T1}(tr_{x1}, tr_{y1})\}.M_{T2}(tr_{x2}, tr_{y2}) = P.\{M_{T1}(tr_{x1}, tr_{y1}).M_{T2}(tr_{x2}, tr_{y2})\}$$

Ta có :

$$M_{T1}(tr_{x1}, tr_{y1}).M_{T2}(tr_{x2}, tr_{y2}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_{x1} & tr_{y1} & 1 \end{pmatrix}.\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_{x2} & tr_{y2} & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_{x1} + tr_{x2} & tr_{y1} + tr_{y2} & 1 \end{pmatrix}$$

hay : $M_{T1}(tr_{x1}, tr_{y1}).M_{T2}(tr_{x2}, tr_{y2}) = M_T(tr_{x1} + tr_{x2}, tr_{y1} + tr_{y2})$

Vậy kết hợp hai phép tịnh tiến là một phép tịnh tiến. Từ đó ta có kết hợp của nhiều phép tịnh tiến cũng là một phép tịnh tiến.

**2.2. Kết hợp các phép tỉ lệ**

Tương tự như phép tịnh tiến, ta có tọa độ điểm $Q(x', y')$ là điểm có được sau khi kết hợp hai phép tỉ lệ $M_{S1}(s_{x1}, s_{y1})$ và $M_{S2}(s_{x2}, s_{y2})$ là :

$$Q = \{P.M_{S1}(s_{x1}, s_{y1})\}.M_{S2}(s_{x2}, s_{y2}) = P.\{M_{S1}(s_{x1}, s_{y1}).M_{S2}(s_{x2}, s_{y2})\}$$

Ta có :

$$M_{S1}(s_{x1}, s_{y1}).M_{S2}(s_{x2}, s_{y2}) = \begin{pmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{pmatrix}.\begin{pmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} s_{x1}.s_{x2} & 0 & 0 \\ 0 & s_{y1}.s_{y2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

hay : $M_{S1}(s_{x1}, s_{y1}).M_{S2}(s_{x2}, s_{y2}) = M_S(s_{x1}.s_{x2}, s_{y1}.s_{y2})$

Vậy kết hợp hai phép tỉ lệ là một phép tỉ lệ. Dễ dàng mở rộng cho kết quả : kết hợp của nhiều phép tỉ lệ cũng là một phép tỉ lệ.

**2.3. Kết hợp các phép quay**

Tương tự, ta có tọa độ điểm $Q(x', y')$ là điểm phát sinh sau khi kết hợp hai phép quay quanh gốc tọa độ $M_{R1}(\alpha_1)$ và $M_{R2}(\alpha_2)$ là :

$$Q = \{P.M_{R1}(\alpha_1)\}.M_{R2}(\alpha_2) = P.\{M_{R1}(\alpha_1).M_{R2}(\alpha_2)\}$$

Ta có :

$$M_{R1}(\alpha_1).M_{R2}(\alpha_2) = \begin{pmatrix} \cos\alpha_1 & \sin\alpha_1 & 0 \\ -\sin\alpha_1 & \cos\alpha_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.\begin{pmatrix} \cos\alpha_2 & \sin\alpha_2 & 0 \\ -\sin\alpha_2 & \cos\alpha_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\alpha_1 + \alpha_2) & \sin(\alpha_1 + \alpha_2) & 0 \\ -\sin(\alpha_1 + \alpha_2) & \cos(\alpha_1 + \alpha_2) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

hay : $M_{R1}(\alpha_1).M_{R2}(\alpha_2) = M_R(\alpha_1 + \alpha_2)$

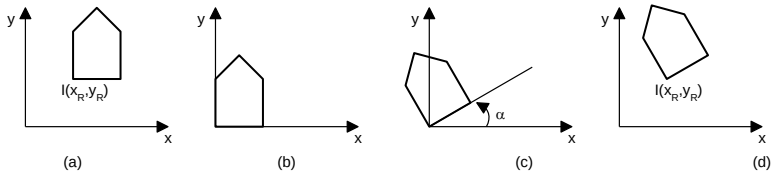
Vậy kết hợp hai phép quay quanh gốc tọa độ là một phép quay quanh gốc tọa độ. Từ đó dễ dàng suy ra kết hợp của nhiều phép quay quanh gốc tọa độ cũng là một phép quay quanh gốc tọa độ.

**2.4. Phép quay có tâm quay là điểm bất kì**

Giả sử tâm quay có tọa độ $I(x_R, y_R)$, ta có thể xem phép quay quanh tâm I một góc $\alpha$ được kết hợp từ các phép biến đổi cơ sở sau:

- Tịnh tiến theo vector tịnh tiến $(-x_R, -y_R)$ để dịch chuyển tâm quay về gốc tọa độ (đưa về trường hợp quay quanh gốc tọa độ).

- Quay quanh gốc tọa độ một góc $\alpha$.
- Tịnh tiến theo vector tịnh tiến $(x_R, y_R)$ để đưa tâm quay về lại vị trí ban đầu.




**Hình 3.4** – Phép quay quanh tâm là điểm bất kì. Đối tượng trước khi biến đổi(a), Sau khi tịnh tiến về gốc tọa độ(b), Sau khi quay góc $\alpha$ (c), Sau khi tịnh tiến về tâm quay ban đầu(d).

Ta có ma trận của phép biến đổi :

$$M_R(x_R, y_R, \alpha) = M_T(-x_R, -y_R).M_R(\alpha).M_T(x_R, y_R)$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_R & -y_R & 1 \end{pmatrix}\begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_R & y_R & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ (1-\cos\alpha)x_R + \sin\alpha.y_R & -\sin\alpha.x_R + (1-\cos\alpha)y_R & 1 \end{pmatrix}$$

## 3. MỘT SỐ TÍNH CHẤT CỦA PHÉP BIẾN ĐỔI AFFINE

***Phép biến đổi affine bảo toàn đường thẳng***

Ảnh của đường thẳng qua phép biến đổi affine là đường thẳng.

Thật vậy, ta có phương trình tham số của đường thẳng qua hai điểm A, B là : $P(t) = (1-t)A + tB$. $Q(t)$ các điểm nhận được sau phép biến đổi M.

$$Q(t) = P(t).M = [(1-t)A + tB]M = (1-t)AM + tBM$$

Nếu gọi A', B' lần lượt là ảnh của A, B qua phép biến đổi M, ta sẽ có $A' = AM, B' = BM$. Lúc này $Q(t) = (1-t)A' + tB'$. Đây chính là dạng của phương trình tham số đoạn thẳng qua A', B'.

Từ kết quả trên, để biến đổi một đoạn thẳng đi qua hai điểm A và B, ta chỉ cần áp dụng phép biến đổi cho hai điểm A, B rồi vẽ lại đoạn thẳng qua hai điểm mới.

***Tính song song của các đường thẳng được bảo toàn***

Ảnh của hai đường thẳng song song là hai đường song song.

Chúng ta có thể viết lại phương trình tham số của đường thẳng dưới dạng tia xuất phát từ A ứng với t=0 và theo phương $\beta = B - A$ như sau : $A + \beta t$. Lúc này ta biểu diễn hai đường thẳng song song dưới dạng tia : $L_1(t) = A_1 + \beta t$ và $L_2(t) = A_2 + \beta t$ có cùng phương $\beta t$ nhưng xuất phát từ hai điểm khác nhau. Lúc này áp dụng phép biến đổi lên hai đường thẳng song song này, dễ dàng nhận ra ảnh của chúng sẽ có phương $\beta M$ nên chúng song song.

Một hệ quả quan trọng của tính chất này đó là ảnh của các hình bình hành sau phép biến đổi là các hình bình hành.

***Tính tỉ lệ về khoảng cách được bảo toàn***

Giả sử C là điểm chia đoạn AB theo tỉ số t. Nếu A', B', C' lần lượt là ảnh A, B, C qua phép biến đổi thì C' cũng sẽ chia A'B' theo tỉ số t.

Trong trường hợp đặc biệt, nếu C là trung điểm của AB thì C' cũng là trung điểm của A'B', từ đó ta có thể suy ra một số tính chất sau :

- Trong hình vuông, các đường chéo cắt nhau tại trung điểm của mỗi đường nên các đường chéo của bất cứ hình bình hành nào cũng cắt nhau tại trung điểm của mỗi đường.
- Trong tam giác đều, giao điểm của ba đường trung tuyến chia mỗi đường theo tỉ số 1:2. Mặt khác, một tam giác bất kì là ảnh của tam giác đều qua phép biến đổi affine, nên giao điểm của các đường trung tuyến của nó cũng sẽ chia chúng theo tỉ lệ 1:2.

## 4. MỘT SỐ PHÉP BIẾN ĐỔI KHÁC

### 4.1. Phép đối xứng

Phép đối xứng trục có thể xem là phép quay quanh trục đối xứng một góc $180^0$. Nếu trục đối xứng là trục hoành hay trục tung, chúng ta có biểu diễn của phép đối xứng qua trục hoành, trục tung lần lượt là :

$$M_{Rfx} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_{Rfy} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

### 4.2. Phép biến dạng

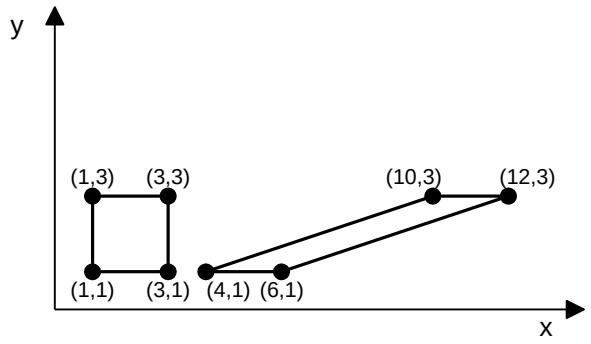
Phép biến dạng là phép biến đổi làm thay đổi, méo mó hình dạng của các đối tượng. Hai dạng phép biến dạng thường gặp đó là biến dạng theo phương trục x và biến dạng theo phương trục y bằng cách thay đổi tọa độ $(x, y)$của điểm ban đầu theo cách sau :

Biến dạng theo phương trục x sẽ làm thay đổi hoành độ còn tung độ vẫn giữ nguyên

$$M_{Shx} = \begin{pmatrix} 1 & 0 & 0 \\ sh_{xy} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Biến dạng theo phương trục y sẽ làm thay đổi tung độ còn hoành độ vẫn giữ nguyên

$$M_{Shy} = \begin{pmatrix} 1 & sh_{yx} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$




$sh_{xy}$ và $sh_{yx}$ lần lượt được gọi là các hệ số biến dạng.

**Hình 3.5** – Phép biến dạng theo phương trục x với hệ số biến dạng $sh_{xy} = 3$

### 4.3. Phép biến đổi ngược

Chúng ta thường dùng phép biến đổi ngược để có thể undo một phép biến đổi đã thực hiện.

Ta có Q là ảnh của P qua phép biến đổi T có ma trận biến đổi M là : $Q = PM$ , từ đó phép biến đổi ngược $T^{-1}$ sẽ có ma trận biến đổi là $M^{-1}$ với $M^{-1}$ là ma trận nghịch đảo của ma trận M.

Với giả thiết ban đầu về ma trận M là $ad - bc \neq 0$, ta có công thức tính ma trận nghịch đảo $M^{-1}$ của $M = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$ là :

$$M^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b & 0 \\ -c & a & 0 \\ cf - de & be - af & 1 \end{pmatrix}$$

Như vậy ta có ma trận của các phép biến đổi ngược của các phép biến đổi cơ sở tịnh tiến, tỉ lệ, quay lần lượt như sau :

$$M_T^{-1}(tr_x, tr_y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -tr_x & -tr_y & 1 \end{pmatrix} = M_T(-tr_x, -tr_y)$$

$$M_S^{-1}(s_x, s_y) = \frac{1}{s_x s_y} \begin{pmatrix} s_y & 0 & 0 \\ 0 & s_x & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{pmatrix} = M_S\left(\frac{1}{s_x}, \frac{1}{s_y}\right)$$

$$M_R^{-1}(\alpha)=\begin{pmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{pmatrix}=M_R(-\alpha)$$

### 4.4. Phân rã phép biến đổi

Một phép biến đổi bất kì có thể được phân rã thành tích các phép biến đổi cơ sở như tịnh tiến, quay, tỉ lệ.

Một phép biến dạng theo phương trục x có thể được phân rã thành tích của một phép biến đổi tỉ lệ và một phép biến dạng đơn vị, và với một phép biến đổi tỉ lệ khác theo công thức sau :

$$\begin{pmatrix}1 & 0 & 0\\ sh_{xy} & 1 & 0\\ 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}\frac{1}{sh_{xy}} & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}sh_{xy} & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

Phép biến dạng đơn vị còn có thể được phân rã tiếp :

$$\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\phi & 0 & 0\\ 0 & \frac{1}{\phi} & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\cos\beta & \sin\beta & 0\\ -\sin\beta & \cos\beta & 0\\ 0 & 0 & 1\end{pmatrix}$$

trong đó $\begin{cases}\alpha=\tan^{-1}(\phi)=58.28^0\\ \beta=\tan^{-1}\left(\frac{1}{\phi}\right)=31.72^0\end{cases}$

Từ đó, một phép biến đổi bất kì có thể được phân rã thành các phép biến đổi cơ sở sau :

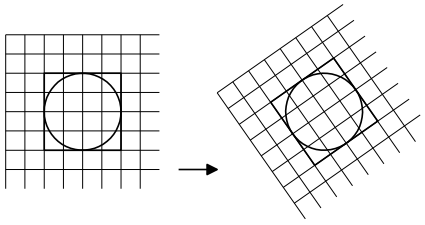
$$\begin{pmatrix}a & b & 0\\ c & d & 0\\ e & f & 1\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ \frac{ac+bd}{Q^2} & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}Q & 0 & 0\\ 0 & \frac{ad-bc}{Q} & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\frac{a}{Q} & \frac{b}{Q} & 0\\ -\frac{b}{Q} & \frac{a}{Q} & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ e & f & 1\end{pmatrix}$$

trong đó $Q^2=a^2+b^2$.

Với cách lập luận trên ta nhận thấy : bất kì phép biến đổi nào cũng được kết hợp từ các phép biến dạng, tỉ lệ, quay, và tịnh tiến. Tuy nhiên, theo kết quả ở bước trước, phép biến dạng là sự kết hợp của các phép quay, tỉ lệ, nên từ đó suy ra bất kì phép biến đổi nào cũng được kết hợp từ các phép tịnh tiến, tỉ lệ và quay.

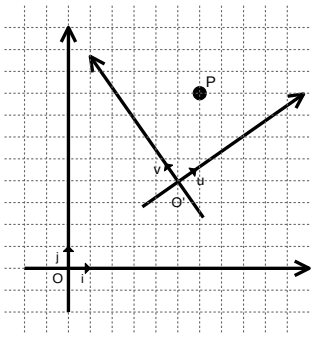
## 5. PHEP BIẾN DỔI GIỮA CAC HỆ TỌA DỘ

Để thuận tiện cho việc mô tả đối tượng, thông thường đối tượng sẽ được mô tả trong các hệ tọa độ cục bộ gắn với chúng. Tuy nhiên để có thể hiển thị toàn bộ một ảnh bao gồm nhiều đối tượng thành phần, các mô tả này phải được chuyển về một hệ tọa độ chung duy nhất. Việc chuyển đổi này thường được chia làm hai loại : chuyển từ các hệ tọa độ không phải là hệ tọa độ Descartes như hệ tọa độ cực, hệ tọa độ cầu, hệ tọa độ elliptic, … sang hệ tọa độ Descartes, và chuyển đổi giữa hai hệ tọa độ Descartes. Trong phần này chúng ta sẽ khảo sát phép biến đổi giữa hai hệ tọa độ Descartes với nhau.



**Hình 3.6** – Phép biến đổi giữa hai hệ tọa độ

Giả sử ta có hệ tọa độ (I) có gốc tọa độ O và các vector đơn vị lần lượt là $i,j$. Hệ tọa độ (II) là ảnh của hệ tọa độ (I) qua phép biến đổi T(M), có gốc tọa độ là O' và các vector đơn vị lần lượt là $u,v$. Lúc này một điểm $P(x,y)$ bất kì trong hệ tọa độ (I) sẽ được biến đổi thành điểm $Q(a,b)$ trong hệ tọa độ (II). Vấn đề đặt ra ở đây là mối liên hệ giữa $a,b$ với $x,y,M$ như thế nào.




Người ta chứng minh được rằng $Q=PM^{-1}$.

**Hình 3.7** – Tọa độ của một điểm qua phép biến đổi hệ tọa độ

## TOM TẮT

Các phép biến đổi hình học cho phép dễ dàng thao tác lên các đối tượng đã được tạo ra. Chúng làm thay đổi mô tả về tọa độ của các đối tượng, từ đó đối tượng sẽ được thay đổi về hướng, kích thước và hình dạng. Các phép biến đổi hình học cơ sở bao gồm tịnh tiến, quay và biến đổi tỉ lệ. Ngoài ra một số phép biến đổi khác cũng thường được áp dụng đó là phép đối xứng và biến dạng.

Có hai quan điểm về phép biến đổi hình học đó là : biến đổi đối tượng và biến đổi hệ tọa độ. Biến đổi đối tượng thay đổi tọa độ của các điểm mô tả nó theo một quy tắc nào đó, còn biến đổi hệ tọa độ sẽ tạo ra một hệ tọa độ mới và tất cả các điểm mô tả đối tượng sẽ được chuyển về hệ tọa độ mới.

Các phép biến đổi hình học đều được biểu diễn dưới dạng ma trận thuần nhất 3x3 để tiện cho việc thực hiện các thao tác kết hợp giữa chúng. Trong hệ tọa độ thuần nhất, tọa độ của một điểm được mô tả bởi một vector dòng bao gồm ba giá trị, hai giá trị đầu tương ứng với tọa độ Descartes của điểm đó, và giá trị thứ ba là 1. Với cách biểu diễn này, ma trận của phép biến đổi có được từ sự kết hợp của các phép biến đổi cơ sở sẽ bằng tích của các ma trận của các phép biến đổi thành phần.

Các phép biến đổi không làm thay đổi kết cấu về tính cân xứng của đối tượng như tịnh tiến, quay được gọi là các phép biến đổi bảo toàn kết cấu đối tượng, thuật ngữ tiếng Anh gọi là rigid-body transformation.

Việc chuyển đổi giữa hai hệ tọa độ Descartes với nhau thường gặp trong công đoạn chuyển các mô tả tọa độ của các đối tượng thành phần trong các hệ tọa độ cục bộ về các vị trí tương ứng trong một hệ tọa độ chung. Giữa hai hệ tọa độ Descartes với nhau, người ta thường sử dụng các phép biến đổi bảo toàn kết cấu như là tịnh tiến, quay.

## BÀI TẬP

**26.** Cho biết ma trận các phép biến đổi dùng để biến đổi một hình tròn thành hình ellipse và ngược lại.

**27.** Cho biết ma trận các phép biến đổi dùng để biến đổi một hình vuông thành hình chữ nhật, hình bình hành và ngược lại.

**28.** Xây dựng và cài đặt cấu trúc dữ liệu và các hàm dùng để thực hiện một phép biến đổi affine bất kì.

**29.** Cho biết ma trận của phép tỉ lệ với tâm tỉ lệ là điểm bất kì.

**30.** Cho biết ma trận của phép lấy đối xứng qua đường thẳng y=mx+b bất kì.

**31.** Cho biết ma trận của phép lấy đối xứng qua tâm là điểm bất kì.

**32.** Cho biết ma trận của phép biến dạng theo phương của đường thẳng y=mx+b.

**33.** Chứng minh rằng ma trận của phép lấy đối xứng qua đường thẳng $y = x$ tương đương với kết hợp của phép lấy đối xứng qua trục hoành và phép quay quanh gốc tọa độ một góc $90^0$.

**34.** Chứng minh rằng ma trận của phép lấy đối xứng qua đường thẳng $y = -x$ tương đương với kết hợp của phép lấy đối xứng qua trục tung và phép quay quanh gốc tọa độ một góc $90^0$.

**35.** Trong phép biến đổi tỉ lệ, $s_x, s_y$ được gọi là các hệ số tỉ lệ theo phương của trục hoành và phương của trục tung. Hãy cho biết công thức của phép biến đổi tỉ lệ theo phương của các trục nghiêng so với trục hoành (các trục này trực giao với nhau) một góc $\alpha$ với các hệ số tỉ lệ theo các phương trên là $s_x, s_y$.

**36.** Chứng minh rằng cặp hai phép tỉ lệ là giao hoán, nghĩa là $M_{S1}M_{S2} = M_{S2}M_{S1}$. Tương tự cho cặp hai phép quay.

**37.** Chứng minh rằng phép đồng dạng và phép quay tạo thành một cặp thao tác có tính giao hoán, nhưng phép biến đổi tỉ lệ thường và phép quay thì không vậy.

**38.** Trình bày ma trận của phép biến dạng dưới dạng tích ma trận của các phép quay và các phép tỉ lệ.

**39.** Trình bày ma trận của phép quay dưới dạng tích ma trận của các phép biến dạng và tỉ lệ.

**40.** Chứng minh rằng phép quay quanh gốc tọa độ có thể được phân tích thành ba phép biến dạng. Đây là cách để quay một ảnh nhanh vì phép biến dạng thường được thực hiện bằng cách di chuyển toàn bộ các khối điểm ảnh (block pixels).

**41.** Chứng minh một phép biến đổi affine bất kì có thể được phân tích thành tích của các phép tịnh tiến, tỉ lệ và quay.

**42.** Chứng minh công thức tính tọa độ của một điểm khi thực hiện phép biến đổi giữa các hệ tọa độ

**43.** Hệ tọa độ $x'O'y'$ nhận được bằng cách quay quanh gốc tọa độ một góc $\alpha$ rồi tịnh tiến theo vector tịnh tiến $(tr_x, tr_y)$ hệ tọa độ $xOy$. Hãy cho biết công thức tọa độ của điểm P trong hệ tọa độ $x'O'y'$ nếu $P(x, y)$ là tọa độ của P trong hệ tọa độ $xOy$.

**44.** Viết chương trình minh họa các bước kết hợp các phép biến đổi cơ sở để tạo thành phép quay một điểm quanh tâm bất kì. Thực hiện tương tự cho phép tỉ lệ có tâm tỉ lệ là điểm bất kì.

**45.** Viết chương trình cho phép người dùng sử dụng các phép biến đổi đã học thao tác lên một đối tượng cho trước.

*CHƯƠNG 4*

# HIỂN THỊ ĐỐI TƯỢNG HAI CHIỀU

Chương này sẽ đề cập tới các kĩ thuật để hiển thị các đối tượng hai chiều trên các thiết bị như màn hình, máy in, …

Các hệ đồ họa cho phép người dùng mô tả các hình ảnh bằng hệ tọa độ thế giới thực. Nó có thể là bất kì hệ tọa độ Descartes nào mà người dùng cảm thấy thuận tiện nhất khi sử dụng. Các hình ảnh được mô tả trong hệ tọa độ thực sau đó sẽ được các hệ đồ họa ánh xạ vào hệ tọa độ thiết bị. Thông thường các hệ đồ họa cho phép người dùng xác định vùng nào của hình ảnh được hiển thị và nó sẽ được hiển thị ở đâu trên màn hình. Ta có thể chọn một vùng hay một số vùng để hiển thị cùng một lúc, các vùng này có thể đặt ở các nơi khác nhau trên màn hình hay lồng vào nhau. Quá trình biến đổi này đòi hỏi các phép biến đổi như dịch chuyển, quay, biến đổi tỉ lệ; và các thao tác loại bỏ các vùng hình ảnh nằm ngoài vùng được định nghĩa, ….

## 6. QUY TRÌNH HIỂN THỊ ĐỐI TƯỢNG HAI CHIỀU

### 6.1. Một số khái niệm

Cửa sổ (window) là một vùng được chọn để hiển thị trong hệ tọa độ thế giới thực.

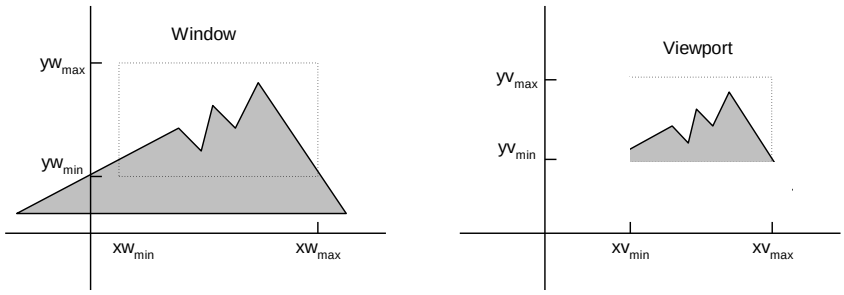
Vùng quan sát (viewport) là vùng được chọn trên thiết bị hiển thị để các đối tượng ở trong cửa sổ ánh xạ vào.

Cửa sổ xác định ***cái gì*** được thấy trên thiết bị hiển thị, còn vùng quan sát xác định ***nơi nào*** nó sẽ được hiển thị.

Ở đây chúng ta nên phân biệt khái niệm cửa sổ được dùng trong phần này với khái niệm cửa sổ được dùng trong các chương trình ứng dụng trên các hệ điều hành như Windows.

Thông thường cửa sổ và vùng quan sát có dạng hình chữ nhật, có các cạnh song song với các trục tọa độ. Tuy nhiên chúng cũng còn có một số dạng khác như đa giác, hình tròn, …

Quá trình ánh xạ một vùng định nghĩa trong hệ tọa độ thế giới thực vào một vùng trong hệ tọa độ thiết bị được gọi là phép biến đổi hệ quan sát (viewing transformation).




**Hình 4.1** – Phép biến đổi hệ quan sát với cửa sổ và vùng quan sát có dạng là các hình chữ nhật

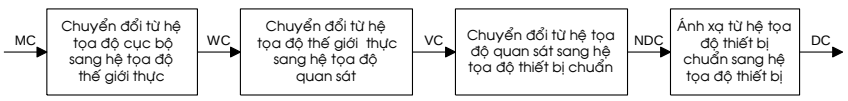
Quy trình hiển thị các đối tượng trong đồ họa hai chiều có thể được mô tả qua sơ đồ sau :

Trước tiên, các đối tượng sẽ được mô tả bằng các đối tượng đồ họa cơ sở và các thuộc tính của chúng trong từng hệ tọa độ cục bộ (modeling coordinates - MC) nhằm đơn giản hóa và tận dụng các đặc trưng riêng của từng loại. Sau đó, chúng ta sẽ dùng các phép biến đổi hệ tọa độ để chuyển các mô tả từ các hệ tọa độ cục bộ này sang một hệ tọa độ thế giới thực (world coordinates - WC) duy nhất chứa toàn bộ các đối tượng thành phần. Phép chuyển đổi này được gọi là phép chuyển đổi mô hình (modeling coordinates transformation).

Tiếp theo, chúng ta sẽ định một hệ tọa độ quan sát (viewing coordinates - VC), là hệ tọa độ mô tả vị trí của người quan sát đối tượng. Nhờ việc sử dụng hệ tọa độ này mà cùng một mô tả, các đối tượng có thể được quan sát ở nhiều góc độ và vị trí khác nhau.

Sau khi chuyển các mô tả đối tượng từ hệ tọa độ thế giới thực sang hệ tọa độ quan sát, chúng ta sẽ định nghĩa cửa sổ trong hệ tọa độ này, đồng thời định nghĩa vùng quan sát trong hệ tọa độ thiết bị chuẩn (normalized device coordinates - NDC) có tọa độ các chiều thay đổi trong khoảng từ 0 đến 1.

Sau khi thực hiện phép ánh xạ từ cửa sổ sang vùng quan sát, tất cả các phần của đối tượng nằm ngoài vùng quan sát sẽ bị xén (clip) và toàn bộ những gì nằm trong vùng quan sát sẽ được ánh xạ sang hệ tọa độ thiết bị (device coordinates - DC). Việc đưa ra hệ tọa độ thiết bị chuẩn nhằm giúp cho việc tương thích dễ dàng với nhiều loại thiết bị hiển thị khác nhau.

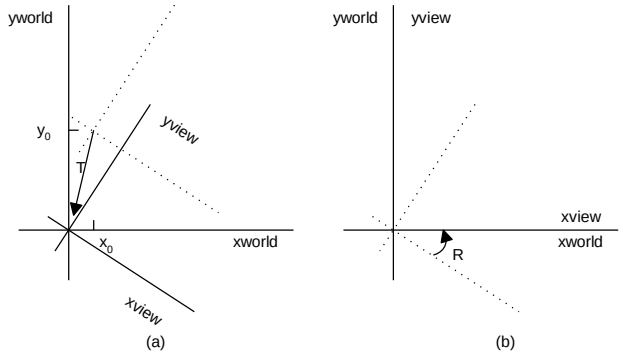



**Hình 4.2** – Quy trình hiển thị đối tượng hai chiều

Bằng cách thay đổi vị trí của vùng quan sát chúng ta có thể quan sát các đối tượng tại các vị trí khác nhau trên màn hình hiển thị, đồng thời, bằng cách thay đổi kích thước của vùng quan sát, chúng ta có thể thay đổi kích thước và tính cân xứng của các đối tượng được hiển thị. Chúng ta có thể thực hiện các hiệu ứng thu phóng bằng cách ánh xạ các cửa sổ có kích thước khác nhau vào vùng quan sát có kích thước cố định. Khi các cửa sổ được thu nhỏ, phần nằm trong cửa sổ sẽ được phóng to giúp chúng ta dễ dàng quan sát các chi tiết mà không thể thấy được trong các cửa sổ lớn hơn.

## 6.2. Hệ tọa độ quan sát và hệ tọa độ thiết bị chuẩn

### 6.2.1. Hệ tọa độ quan sát



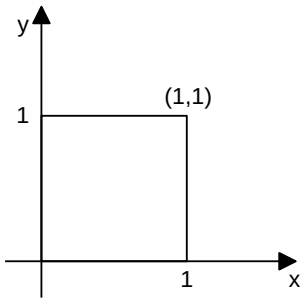

Để thiết lập hệ tọa độ quan sát, trước tiên ta sẽ chọn một điểm $P_0(x_0, y_0)$ trong hệ tọa độ thế giới thực làm gốc tọa độ. Sau đó chúng ta sẽ sử dụng một vector V mô tả hướng quan sát để định hướng cho trục tung $y_v$ của hệ tọa độ. Vector V được gọi là view-up vector.

Từ V chúng ta có thể tính được các vector đơn vị $v = (v_x, v_y)$ và $u = (u_x, u_y)$ tương ứng cho các trục tung $y_v$ và trục hoành $x_v$ của hệ tọa độ. Các vector đơn vị này sẽ được dùng để tạo thành hai dòng đầu tiên của ma trận quay $M_R$ để đưa các trục $x_v y_v$ trùng với các trục $x_w y_w$ của hệ trục tọa độ thế giới thực.

**Hình 4.3** – Phép biến đổi một điểm từ hệ tọa độ quan sát sang hệ tọa độ thực

Ma trận của phép chuyển một điểm trong hệ tọa độ thế giới thực sang hệ tọa độ quan sát là tích của hai ma trận của các phép biến đổi : phép tịnh tiến gốc tọa độ hệ quan sát về gốc tọa độ hệ tọa độ thế giới thực, phép quay đưa các trục của hệ tọa độ quan sát trùng với các trục của hệ tọa độ thế giới thực. $M_{WC,VC} = M_T M_R$.

### 6.2.2. Hệ tọa độ thiết bị chuẩn

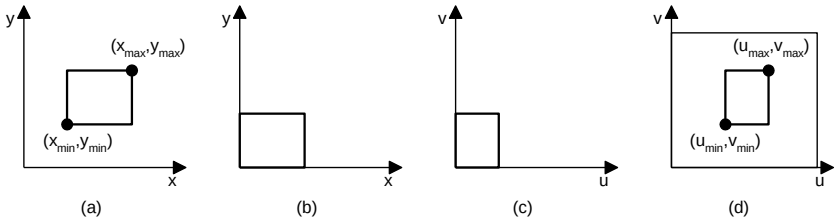



Do cách định nghĩa của các hệ tọa độ thiết bị khác nhau nên một hình ảnh hiển thị được trên thiết bị này chưa chắc hiển thị chính xác trên thiết bị kia. Chính vì vậy cần phải xây dựng hệ tọa độ thiết bị chuẩn đại diện chung cho các thiết bị để có thể mô tả các hình ảnh của thế giới thực mà không phụ thuộc vào bất cứ thiết bị nào.

Trong hệ tọa độ này, các tọa độ x, y sẽ được gán các giá trị trong khoảng từ 0 đến 1. Như vậy, vùng không gian của hệ tọa độ thiết bị chuẩn chính là hình vuông đơn vị có góc trái dưới là (0,0) và góc phải trên (1,1).

**Hình 4.4** – Hệ tọa độ thiết bị chuẩn

## 6.3. Chuyển đổi từ cửa sổ sang vùng quan sát




Phép chuyển đổi từ cửa sổ sang vùng quan sát bao gồm 3 phép biến đổi : phép tịnh tiến để dịch chuyển góc trái dưới về gốc tọa độ (hình 4.5a), phép biến đổi tỉ lệ để chỉnh kích thước của cửa sổ về cùng kích thước của vùng quan sát (hình 4.5b, hình 4.5c), cuối cùng là phép tịnh tiến dịch chuyển về góc trái dưới của vùng quan sát (hình 4.5d).

**Hình 4.5** – Phép chuyển đổi từ cửa sổ sang vùng quan sát

Ta có ma trận của phép biến đổi :

$$M_{WV} = M_{TW}(-x_{\min}, -y_{\min}) M_S\left(\frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}, \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}\right) M_{TV}(u_{\min}, v_{\min})$$

$$= \begin{pmatrix} \dfrac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & 0 \\ 0 & \dfrac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & 0 \\ -x_{\min}\dfrac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} + u_{\min} & -y_{\min}\dfrac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} + v_{\min} & 1 \end{pmatrix}$$

Như vậy nếu $P(x, y)$ là điểm trong cửa sổ thì nó sẽ có tọa độ trong vùng quan sát là : $(sx(x - x_{\min}) + u_{\min}, sy(y - y_{\min}) + v_{\min})$ với $sx = \dfrac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}}$, $sy = \dfrac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}}$ .

$sx, sy$ là các hệ số tỉ lệ của các kích thước của cửa sổ và vùng quan sát. Khi $sx = sy = 1$, các đối tượng qua phép chuyển đổi sẽ được giữ nguyên hình dáng và tính cân xứng.

### 6.4. Các thuật toán xén hình

Thao tác loại bỏ các phần hình ảnh nằm ngoài một vùng cho trước được gọi là xén hình. Vùng được dùng để xén hình gọi là cửa sổ xén (clip window).

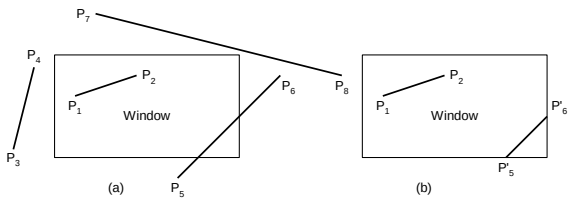
Tùy thuộc vào từng ứng dụng cụ thể mà cửa sổ xén có thể có dạng là đa giác hay là đường cong khép kín. Trong phần này chúng ta sẽ khảo sát các thuật toán xén hình vào cửa sổ xén là hình chữ nhật trước, sau đó sẽ khảo sát các cửa sổ xén có dạng khác. Để đơn giản, trong các thuật toán xén hình, cửa sổ xén được gọi là cửa sổ.

## 7. CÁC THUẬT TOÁN XÉN ĐIỂM, ĐOẠN THẲNG

Giả sử cửa sổ xén là cửa sổ hình chữ nhật có tọa độ của các điểm dưới bên trái và điểm trên bên phải lần lượt là $(x_{\min}, y_{\min})$ và $(x_{\max}, y_{\max})$.

Một điểm $P(x,y)$ được coi là nằm bên trong cửa sổ nếu thỏa hệ bất phương trình : $\begin{cases} x_{\min} \le x \le x_{\max} \\ y_{\min} \le y \le y_{\max} \end{cases}$

Bây giờ, ta sẽ xét bài toán xén đoạn thẳng được cho bởi hai điểm $P_1(x_1, y_1)$ và $P_2(x_2, y_2)$vào cửa sổ hình chữ nhật trên.




**Hình 4.6** – Minh họa thao tác xén các đoạn thẳng vào một cửa sổ hình chữ nhật. Trước khi xén (a). Sau khi xén (b).

Thao tác xén hình là một trong những thao tác cơ bản của quá trình hiển thị đối tượng, do đó vấn đề tối ưu tốc độ luôn là đích cho các thuật toán nhắm đến. Ý tưởng chung của các thuật toán xén đoạn thẳng đó là loại bỏ phép toán tìm giao điểm giữa đoạn thẳng với biên của cửa sổ một cách nhanh nhất đối với các đoạn thẳng đặc biệt như nằm hoàn toàn trong hoặc hoàn toàn bên ngoài cửa sổ (ví dụ như đoạn $P_1P_2$ và $P_3P_4$ trong hình 4.6). Đối với các đoạn thẳng có khả năng cắt cửa sổ, cần phải đưa ra cách tìm giao điểm thật nhanh.

Nhận xét rằng, các đoạn thẳng mà có cả hai điểm nằm hoàn toàn trong cửa sổ thì cả đoạn thẳng nằm trong cửa sổ, đây cũng chính là kết quả sau khi xén (ví dụ như đoạn thẳng $P_1P_2$), mặt khác đối với các đoạn thẳng mà có hai điểm nằm về cùng một phía của cửa sổ thì luôn nằm ngoài cửa sổ và sẽ bị mất sau khi xén (ví dụ như đoạn thẳng $P_3P_4$). Với các đoạn thẳng có khả năng cắt cửa sổ (ví dụ như đoạn thẳng $P_5P_6$ và $P_7P_8$) để việc tìm giao điểm nhanh cần rút gọn việc tìm giao điểm với những biên cửa sổ không cần thiết để xác định phần giao nếu có của đoạn thẳng và cửa sổ.

Người ta thường sử dụng phương trình tham số của đoạn thẳng trong việc tìm giao điểm giữa đoạn thẳng với cửa sổ.

$$x = x_1 + t(x_2 - x_1) = x_1 + tDx, \quad Dx = x_2 - x_1$$
$$y = y_1 + t(y_2 - y_1) = y_1 + tDy, \quad Dy = y_2 - y_1, \qquad 0 \le t \le 1$$

Nếu giao điểm ứng với giá trị t nằm ngoài đoạn $[0,1]$ thì giao điểm đó sẽ không thuộc về cửa sổ.

### 7.1. Thuật toán Cohen-Sutherland

Đây là một trong những thuật toán ra đời sớm nhất và thông dụng nhất.

Bằng cách kéo dài các biên của cửa sổ, người ta chia mặt phẳng thành chín vùng gồm cửa sổ và tám vùng xung quanh nó.

***Khái niệm mã vùng (area code)***

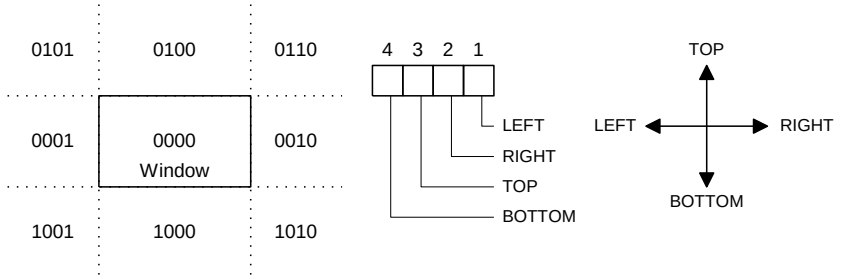
Một con số 4 bit nhị phân gọi là mã vùng sẽ được gán cho mỗi vùng để mô tả vị trí tương đối của vùng đó so với cửa sổ. Bằng cách đánh số từ 1 đến 4 theo thứ tự từ phải qua trái, các bit của mã vùng được dùng theo quy ước sau để chỉ một trong bốn vị trí tương đối của vùng so với cửa sổ bao gồm : trái, phải, trên, dưới.

Bit 1 : trái (LEFT)

Bit 2 : phải (RIGHT)

Bit 3 : trên (TOP)

Bit 4 : dưới (BOTTOM)




Giá trị 1 tương ứng với vị trí bit nào trong mã vùng sẽ chỉ ra rằng điểm đó ở vị trí tương ứng, ngược lại bit đó sẽ được đặt bằng 0. Ví dụ một vùng có mã là 1001, thì nó sẽ nằm phía dưới (bit 4 bằng 1), bên trái (bit 1 bằng 1) so với cửa sổ, vùng có mã là 0000 chính là cửa sổ.

**Hình 4.7** – Mã vùng quy định vị trí tương đối của vùng so với cửa sổ

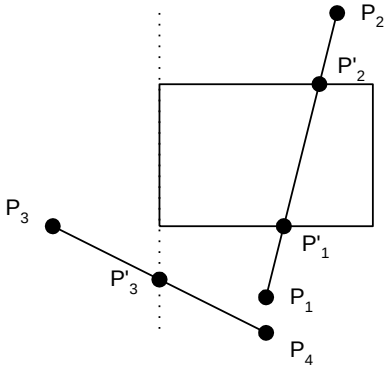
Các giá trị bit trong mã vùng được tính bằng cách xác định tọa độ của điểm $(x,y)$ thuộc vùng đó với các biên của cửa sổ. Bit 1 được đặt là 1 nếu $x < x_{\min}$ , các bit khác được tính tương tự.

***Thuật toán***

Gán mã vùng tương ứng cho các điểm đầu cuối $P_1, P_2$ của đoạn thẳng cần xén lần lượt là $c_1, c_2$. Ta có nhận xét :

- Các đoạn thẳng nằm hoàn toàn bên trong cửa sổ sẽ có $c_1 = c_2 = 0000$, ứng với các đoạn này, kết quả sau khi xén là chính nó.
- Nếu tồn tại $k \in \{1,..,4\}$, sao cho với bit thứ k của $c_1, c_2$ đều có giá trị 1, lúc này đoạn thẳng sẽ nằm về cùng phía ứng với bit k so với cửa sổ, do đó nằm hoàn toàn ngoài cửa sổ. Đoạn này sẽ bị loại bỏ sau khi xén. Ví dụ, nếu $c_1 = 1001, c_2 = 0101$, rõ ràng bit 1 của chúng đều bằng 1 (ứng với biên trái), do đó đoạn thẳng nằm hoàn toàn về biên trái của cửa sổ. Để xác định tính chất này, đơn giản chỉ cần thực hiện phép toán logic AND trên $c_1, c_2$. Nếu kết quả khác 0000, đoạn thẳng sẽ nằm hoàn toàn ngoài cửa sổ.
- Nếu $c_1, c_2$ không thuộc về hai trường hợp trên, đoạn thẳng có thể hoặc không cắt ngang cửa sổ (ví dụ đoạn $P_5P_6, P_7P_8$ trong hình 4.6) chắc chắn sẽ tồn tại một điểm nằm ngoài cửa sổ, không mất tính tổng quát giả sử điểm đó là $P_1$. Bằng cách xét mã vùng của $P_1$ là $c_1$ ta có thể xác định được các biên mà đoạn thẳng có thể cắt để từ đó chọn một biên và tiến hành tìm giao điểm $P_1'$ của đoạn thẳng với biên đó. Lúc này, đoạn thẳng ban đầu được xén thành $P_1P_1'$. Tới đây chúng ta lại lặp lại thao tác đã xét cho đoạn thẳng mới $P_1P_1'$ cho tới khi xác định được phần nằm trong hoặc loại bỏ toàn bộ đoạn thẳng.

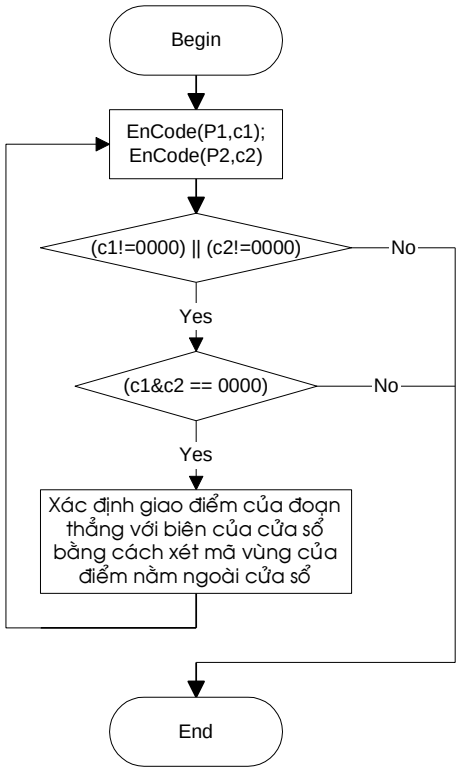
Chúng ta minh họa thuật toán bằng hình vẽ 4.8. Với đoạn thẳng $P_1P_2$, ta sẽ kiểm tra $P_1$ lần lượt với các biên trái, phải, dưới, trên và tìm ra điểm này nằm dưới cửa sổ, sau đó chúng ta tìm giao điểm $P_1'$ của đoạn thẳng với biên dưới. Lúc này đoạn thẳng ban đầu được xén ngắn lại thành $P_1'P_2$. Vì $P_2$ nằm ngoài cửa sổ nên bằng cách xét tương tự, chúng ta sẽ tiến hành tìm giao điểm $P_2'$ của $P_1'P_2$ với biên trên và lúc này đoạn $P_1'P_2'$, chính là phần nằm hoàn toàn trong cửa sổ.




Trong trường hợp đoạn $P_3P_4$, $P_3$ nằm bên trái cửa sổ nên chúng ta có thể xác định điểm giao $P_3'$, và từ đó loại bỏ đoạn thẳng $P_3P_3'$. Bằng cách kiểm tra mã vùng, chúng ta dễ dàng xác định được đoạn thẳng $P_3'P_4$ nằm hoàn toàn bên dưới cửa sổ nên có thể bỏ đi toàn bộ.

**Hình 4.8** – Minh họa thuật toán Cohen - Sutherland

Các điểm giao với các biên cửa sổ của đoạn thẳng có thể được tính từ phương trình tham số như đã đề cập ở phần trên. Tung độ y của điểm giao đoạn thẳng với biên đứng của cửa sổ có thể tính từ công thức $y = y_1 + m(x - x_1)$, trong đó x có thể là $x_{\min}$ hay $x_{\max}$. Tương tự, hoành độ x của điểm giao đoạn thẳng với biên ngang của cửa sổ có thể tính từ công thức : $x = x_1 + (y - y_1)/m$, trong đó y có thể là $y_{\min}$ hay $y_{\max}$.




***Lưu đồ thuật toán Cohen-Sutherland dùng để xén một đoạn thẳng qua hai điểm ($x_1$,$y_1$) và ($x_2$,$y_2$) vào cửa sổ hình chữ nhật cho trước***

***Cài đặt minh họa thuật toán Cohen - Sutherland***

```
#define TRUE            1
#define FALSE           0
#define LEFT            1
#define RIGHT           2
#define TOP         4
#define BOTTOM          8
typedef struct {
    int x, y;
}POINT;

typedef struct {
    int Left, Top, Right, Bottom;
}RECT;
```

```
typedef int CODE;

#define Accept(a,b)                (!(a|b))
#define Reject(a,b)      (a&b)

// Tra ve ma vung cua p la c
void EnCode(POINT p, CODE &c, RECT rWin)
{
    c = 0;
    if(p.x < rWin.Left)
       c |= LEFT;
    if(p.x > rWin.Right)
       c |= RIGHT;
    if(p.y > rWin.Top)
       c |= TOP;
    if(p.y < rWin.Bottom)
       c |= BOTTOM;
}

// Hoan vi hai diem p1 va p2 sao cho p1 luon nam ngoai cua so
void SwapPoint(POINT& p1, POINT &p2, CODE &c1, CODE &c2)
{

    if(!c1) // Neu p1 nam hoan toan trong cua so
    {
       POINT p;
       p = p1;
       p1 = p2;
       p2 = p;

       CODE c;
       c = c1;
       c1 = c2;
       c2 = c;
    }
}

// Tra ve TRUE neu co cat cua so. Nguoc lai tra ve FALSE
int CohenSutherlandClipping(POINT P1, POINT P2, POINT &Q1, POINT &Q2, RECT rWin)
{
    int fStop = FALSE, fResult = FALSE;
    CODE c1, c2;

    while(!fStop)
    {
```

```
EnCode(P1, c1, rWin);
EnCode(P2, c2, rWin);

// Neu duong thang nam hoan toan ben trong cua so
if(Accept(c1, c2))
{
    fStop       = TRUE; // break
    fResult     = TRUE;
} // Accept

else
{
   // Neu duong thang nam hoan toan ben ngoai cua so
   if(Reject(c1,c2))
   {
        fStop = TRUE; // break
   } // Reject
   else // Xet truong hop duong thang co the cat cua so
   {
        SwapPoint(P1, P2, c1, c2);
        float m;

        if(P2.x!=P1.x)
             m = float(P2.y-P1.y)/(P2.x-P1.x);
        if(c1 & LEFT)
        {
             P1.y += (rWin.Left-P1.x)*m;
             P1.x  = rWin.Left;
        } // Left
        else
        {
             if(c1 & RIGHT)
             {
                  P1.y      += (rWin.Right-P1.x)*m;
                  P1.x      = rWin.Right;
             } // Right
             else
             {
                  if(c1 & TOP)
                  {
                       if(P2.x!=P1.x)
                          P1.x += (rWin.Top - P1.y)/m;
                          P1.y  = rWin.Top;
                  } // Top
                  else // Bottom
```

```
                {
                    if(P2.x!=P1.x)
                      P1.x      += (rWin.Bottom - P1.y)/m;
                    P1.y  = rWin.Bottom;
                } // Bottom
            }
        }
    } // Xet truong hop duong thang co the cat cua so
  }
} //while

Q1 = P1;
Q2 = P2;
return (fResult);
} //CohenSutherlandClipping
```

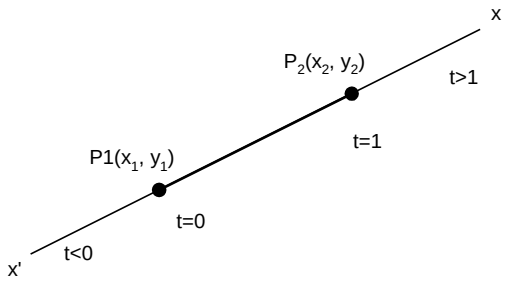
### 7.2. Thuật toán Liang-Barsky

Thuật toán Liang-Barsky được phát triển dựa vào việc phân tích dạng tham số của phương trình đoạn thẳng.

$$x = x_1 + t(x_2 - x_1) = x_1 + tDx, \quad Dx = x_2 - x_1$$
$$y = y_1 + t(y_2 - y_1) = y_1 + tDy, \quad Dy = y_2 - y_1, \qquad 0 \le t \le 1$$

Ứng với mỗi giá trị t, ta sẽ có một điểm P tương ứng thuộc đường thẳng.

- Các điểm ứng với $t \ge 1$ sẽ thuộc về tia $P_2x$.
- Các điểm ứng với $t \le 0$ sẽ thuộc về tia $P_2x'$.
- Các điểm ứng với $0 \le t \le 1$ sẽ thuộc về đoạn thẳng $P_1P_2$.




**Hình 4.9** – Phương trình tham số của đoạn thẳng

Tập hợp các điểm thuộc về phần giao của đoạn thẳng và cửa sổ ứng với các giá trị t thỏa hệ bất phương trình :

$$\begin{cases} x_{\min} \le x_1 + tDx \le x_{\max} \\ y_{\min} \le y_1 + tDy \le y_{\max} \\ 0 \le t \le 1 \end{cases}$$

Đặt
$$\begin{aligned} p_1 &= -Dx, & q_1 &= x_1 - x_{\min} \\ p_2 &= Dx, & q_2 &= x_{\max} - x_1 \\ p_3 &= -Dy, & q_3 &= y_1 - y_{\min} \\ p_4 &= Dy, & q_4 &= y_{\max} - y_1 \end{aligned}$$

Lúc này ta viết hệ phương trình trên dưới dạng :

$$\begin{cases} p_k t \le q_k, \quad k \in \{1,2,3,4\} \\ 0 \le \mathrm{t} \le 1 \end{cases}$$

Như vậy việc tìm đoạn giao thực chất là tìm nghiệm của hệ bất phương trình này. Có hai khả năng xảy ra đó là :

- Hệ bất phương trình vô nghiệm, nghĩa là đường thẳng không có phần giao với cửa sổ nên sẽ bị loại bỏ.

- Hệ bất phương trình có nghiệm, lúc này tập nghiệm sẽ là các giá trị t thỏa $t \in [t_1, t_2] \subseteq [0,1]$.

Ta xét các trường hợp :

- Nếu $\exists\ k \in \{1,2,3,4\} : (p_k = 0) \wedge (q_k < 0)$ thì rõ ràng bất phương trình ứng với k trên là vô nghiệm, do đó hệ vô nghiệm.
- Nếu $\forall\ k \in \{1,2,3,4\} : (p_k \neq 0) \vee (q_k \geq 0)$ thì với các bất phương trình mà ứng với $p_k = 0$ là các bất phương trình hiển nhiên, lúc này hệ bất phương trình cần giải tương đương với hệ bất phương trình có $p_k \neq 0$.
- Với các bất phương trình $p_k t \leq q_k$ mà $p_k < 0$, ta có $t \geq q_k / p_k$.
- Với các bất phương trình $p_k t \leq q_k$ mà $p_k > 0$, ta có $t \leq q_k / p_k$.

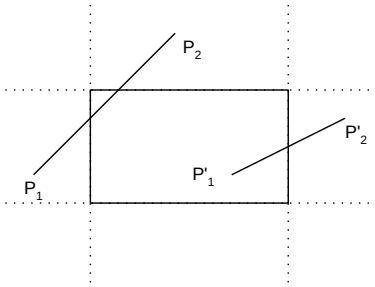
Vậy nghiệm của hệ bất phương trình là $[t_1, t_2]$ với :

$$\begin{cases} t_1 = \max(\left\{\dfrac{q_k}{p_k}, p_k < 0\right\} \cup \{0\}) \\ t_2 = \min(\left\{\dfrac{q_k}{p_k}, p_k > 0\right\} \cup \{1\}) \\ t_1 \leq t_2 \end{cases}$$

Nếu hệ trên có nghiệm thì đoạn giao $Q_1Q_2$ sẽ là $Q_1(x_1 + t_1 Dx, y_1 + t_1 Dy), Q_2(x_1 + t_2 Dx, y_1 + t_2 Dy)$.

Nếu xét thuật toán này ở khía cạnh hình học ta có :

- Trường hợp $\exists\ k \in \{1,2,3,4\} : (p_k = 0) \wedge (q_k < 0)$ tương ứng với trường hợp đoạn thẳng cần xét song song với một trong các biên của cửa sổ ( $p_k = 0$ ) và nằm ngoài cửa sổ ( $q_k < 0$ ) nên sẽ bị loại bỏ sau khi xén.
- Với $p_k \neq 0$, giá trị $t = r_k = q_k / p_k$ sẽ tương ứng với giao điểm của đoạn thẳng với biên k kéo dài của cửa sổ. Trường hợp $p_k < 0$, kéo dài các biên cửa sổ và đoạn thẳng về vô cực, ta có đường thẳng đang xét sẽ có hướng đi từ bên ngoài vào bên trong cửa sổ. Nếu $p_k > 0$, đường thẳng sẽ có hướng đi từ bên trong cửa sổ đi ra. Do đó hai đầu mút của đoạn giao sẽ ứng với các giá trị $t_1, t_2$ được tính như sau : Giá trị $t_1$ chính là giá trị lớn nhất của các $r_k = q_k / p_k$ mà $p_k < 0$ (đường thẳng đi từ ngoài vào trong cửa sổ) và 0; giá trị $t_2$ chính là giá trị nhỏ nhất của các $r_k = q_k / p_k$ mà $p_k > 0$ (đường thẳng đi từ trong cửa sổ đi ra) và 1.




**Hình 4.10** – Xét với biên trái đoạn thẳng $P_1P_2$ có hướng đi từ ngoài vào trong, nhưng so với biên phải đoạn thẳng $P'_1P'_2$ lại có hướng đi từ trong cửa sổ đi ra

Khi cài đặt thuật toán Liang-Barsky, ban đầu các giá trị $t_1$, $t_2$ được khởi động $t_1 = 0, t_2 = 1$. Với mỗi lần xén đoạn thẳng với một biên của cửa sổ, các giá trị $p, q$ sẽ được truyền cho hàm ClipTest để xác định đoạn thẳng có bị loại bỏ hay bị xén bớt một đoạn hay không. Khi $p < 0$, tham số $r = q / p$ sẽ được xem xét để cập nhật $t_1$, khi $p > 0$, r dùng để cập nhật $t_2$. Khi cập nhật $t_1$ và $t_2$ nếu $t_1 > t_2$, đoạn thẳng sẽ bị loại bỏ. Ngoài ra nếu (p=0 và q<0), chúng ta cũng sẽ loại bỏ đoạn thẳng vì nó song song và nằm ngoài cửa sổ. Nếu đoạn thẳng không bị loại bỏ sau bốn lần gọi với các tham số p, q tương ứng với các biên của cửa sổ, các giá trị $t_1$ và $t_2$ sẽ được dùng để suy ra tọa độ hai điểm đầu mút của đoạn giao.

***Cài đặt minh họa thuật toán Liang - Barsky***

```
// Tra ve TRUE neu khong xay ra truong hop nam ngoai cua so
int ClipTest(int p, int q, float &t1, float &t2)
{
    float r;

    if (p<0)
    {
```

```
            r = float(q)/p;
            if (r>t2)
                return FALSE;
            else
                if (r>t1)
                    t1 = r;
        }
        else
        {
            if (p>0)
            {
                r = float(q)/p;
                if (r<t1)
                    return FALSE;
                else
                    if (r<t2)
                        t2 = r;
            }
            else // p=0
            {
                if (q<0)
                    return FALSE;
            }
        }
        return TRUE;
}

int LiangBarskyClipping(POINT P1, POINT P2, RECT R, POINT *Q1, POINT *Q2)
{
        float t1, t2;
        int Dx, Dy, x1, y1, x2, y2, xmin, ymin, xmax, ymax;

        t1 = 0;
        t2 = 1;
        x1 = P1.x; y1 = P1.y;
        x2 = P2.x; y2 = P2.y;
        Dx = x2 - x1; Dy = y2 - y1;
        xmin = R.Left; ymin = R.Top;
        xmax = R.Right; ymax = R.Bottom;
        if (ClipTest(-Dx, x1 - xmin, t1, t2)) // Giai he bat phuong trinh 1
        {
            if (ClipTest(Dx, xmax - x1, t1, t2)) // Giai he bat phuong trinh 2
            {
               if (ClipTest(-Dy, y1 - ymin, t1, t2)) // Giai he bat phuong trinh 3
               {
                  if (ClipTest(Dy, ymax - y1, t1, t2)) // Giai he bat phuong trinh 4
```

```
            {
                Q1.x = x1 + t1. Dx;
                Q1.y = y1 + t1. Dy;
                Q2.x = x1 + t2. Dx;
                Q2.y = y1 + t2. Dy;
                return TRUE;
            } // Giai he bat phuong trinh 4
        } // Giai he bat phuong trinh 3
    } // Giai he bat phuong trinh 2
} // Giai he bat phuong trinh 1
return FALSE;
} // LiangBarskyClipping
```

***Nhận xét***

Thông thường, thuật toán Liang-Barsky cho tốc độ tốt hơn thuật toán Cohen-Sutherland vì rút gọn được số giao điểm cần tính. Mỗi lần cập nhật các giá trị $t_1, t_2$, chỉ cần một phép chia, và giao điểm của đoạn thẳng với cửa sổ chỉ được tính duy nhất một lần sau khi đã tìm ra được giá trị $t_1, t_2$. Trong khi đó thuật toán Cohen-Sutherland đôi lúc phải tính giao điểm cho các điểm không nằm trong biên của cửa sổ đòi hỏi nhiều phép toán hơn.

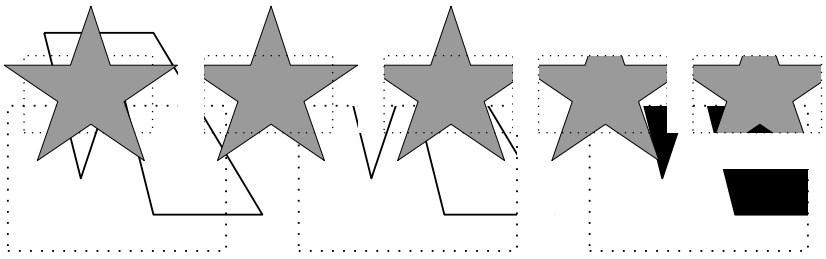
## 8. THUẬT TOÁN XÉN ĐA GIÁC

Chúng ta có thể hiệu chỉnh các thuật toán xén đoạn thẳng để xén đa giác bằng cách xem đa giác như là một tập các đoạn thẳng liên tiếp nối với nhau. Tuy nhiên, kết quả sau khi xén nhiều khi lại là tập các đoạn thẳng rời nhau. Điều chúng ta mong muốn ở đây đó là kết quả sau khi xén phải là một các đa giác để sau này có thể chuyển thành các vùng tô.

**Hình 4.11** – Kết quả sau khi xén đa giác ban đầu. Đa giác ban đầu (a). Kết quả là các đoạn rời nhau (b) và kết quả là các đa giác (c)

Trong phần này chúng ta sẽ khảo sát một trong các thuật toán xén đa giác đó là thuật toán Sutherland-Hodgeman.

**Hình 4.12** – Quá trình xén một đa giác vào cửa sổ

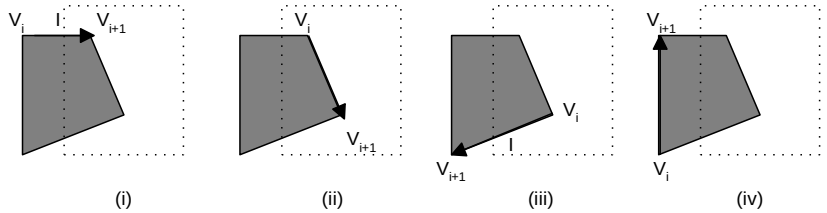


(a) (b) (c)

Thuật toán này sẽ tiến hành xén đa giác lần lượt với các biên cửa sổ. Đầu tiên, đa giác sẽ được xén dọc theo biên trái của cửa sổ, kết quả sau bước này sẽ được dùng để xén tiếp biên phải, rồi cứ tương tự như vậy cho các biên trên, dưới. Sau khi xén hết với bốn biên của cửa sổ, ta được kết quả cuối cùng.

Với mỗi lần xén đa giác dọc theo một biên nào đó của cửa sổ, nếu gọi $V_i, V_{i+1}$ là hai đỉnh kề cạnh $V_iV_{i+1}$, ta có 4 trường hợp có thể xảy ra khi xét từng cặp đỉnh của đa giác ban đầu với biên của cửa sổ như sau:

- (i) Nếu $V_i$ nằm ngoài, $V_{i+1}$ nằm trong, ta lưu giao điểm I của $V_iV_{i+1}$ với biên của cửa sổ và $V_{i+1}$.
- (ii) Nếu cả $V_i$, $V_{i+1}$ đều nằm trong, ta sẽ lưu cả $V_i$, $V_{i+1}$.
- (iii) Nếu $V_i$ nằm trong, $V_{i+1}$ nằm ngoài, ta sẽ lưu $V_i$ và I.
- (iv) Nếu cả $V_i$, $V_{i+1}$ đều nằm ngoài, ta không lưu gì cả.




(i) (ii) (iii) (iv)

**Hình 4.13** – Các trường hợp khi xét $V_i, V_{i+1}$ với các biên của cửa sổ

***Cài đặt minh họa thuật toán Sutherland-Hodgeman***

```
#include <graphics.h>
#include <mem.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

#define TRUE            1
#define FALSE           0
#define LEFT            1
#define RIGHT           2
#define TOP             4
#define BOTTOM          8

typedef struct {
    int x, y;
}POINT;

typedef struct {
    int Left, Top, Right, Bottom;
}RECT;

// Xac dinh p co nam ben trong cua so neu xet theo mot canh b
int Inside(POINT p, int Edge, RECT rWin)
{
    switch(Edge)
    {
        case LEFT   :
            if(p.x < rWin.Left)
                return FALSE;
            break;
        case RIGHT  :
            if(p.x > rWin.Right)
                return FALSE;
            break;
        case TOP :
            if(p.y > rWin.Top)
                return FALSE;
            break;
        case BOTTOM :
            if(p.y < rWin.Bottom)
                return FALSE;
            break;
    }
    return TRUE;
```

```
} //Inside

// Tra ve giao diem cua doan noi p1&p2 voi canh b
POINT Intersect(POINT p1, POINT p2, int Edge, RECT rWin)
{
    POINT iPt;
    float m;

        if(p1.x != p2.x)
                m = float(p2.y-p1.y)/(p2.x-p1.x);
        switch(Edge)
        {
                case LEFT :
                        iPt.x = rWin.Left;
                        iPt.y = p2.y + (rWin.Left-p2.x)*m;
                        break;
                case RIGHT :
                        iPt.x = rWin.Right;
                        iPt.y = p2.y + (rWin.Right-p2.x)*m;
                        break;
                case TOP :
                        iPt.y = rWin.Top;
                        if(p1.x != p2.x)
                            iPt.x = p2.x + (rWin.Top-p2.y)/m;
                        else
                            iPt.x = p2.x;
                        break;
                case BOTTOM :
                        iPt.y = rWin.Bottom;
                        if(p1.x != p2.x)
                            iPt.x = p2.x + (rWin.Bottom-p2.y)/m;
                        else
                            iPt.x = p2.x;
                        break;
        }
        return (iPt);
} // Intersect

// Tien hanh cat da giac voi mot canh nao do cua cua so.
void ClipEdge(POINT *pIn, int N, POINT *pOut, int &Cnt, int Edge,
                        RECT rWin)
{
        int FlagPrevPt = FALSE;
        Cnt = 0;
        POINT pPrev;
```

```
        pPrev = pIn[0];
        if(Inside(pPrev, Edge, rWin)) // Save point
        {
            pOut[Cnt] = pPrev;
            Cnt++;
            FlagPrevPt = TRUE;
        }

        for(int i=1; i<N; i++)
        {
            if(FlagPrevPt) // Diem bat dau nam trong
            {
                if(Inside(pIn[i], Edge, rWin)) // Save point P
                {
                    pOut[Cnt] = pIn[i];
                    Cnt++;
                }
                else // Save I
                {
                    FlagPrevPt = FALSE;
                    pOut[Cnt] = Intersect(pPrev, pIn[i], Edge, rWin);
                    Cnt++;
                }
            }
            else // Diem bat dau canh nam ngoai
            {
                if(Inside(pIn[i], Edge, rWin)) // Save point I, P
                {
                    FlagPrevPt = TRUE;
                    pOut[Cnt] = Intersect(pPrev, pIn[i], Edge, rWin);
                    Cnt++;
                    pOut[Cnt] = pIn[i];
                    Cnt++;
                }
            }
            pPrev = pIn[i];
        }

        // Neu Diem cuoi va dau giao voi bien cua cua so Save point I
        if(!(Inside(pIn[N], Edge, rWin) == Inside(pPrev, Edge, rWin)))
        {
             pOut[Cnt] = Intersect(pPrev, pIn[N], Edge, rWin);
             Cnt++;
        }
        pOut[Cnt] = pOut[0];
    } // Intersect
```

```
void ClipPolygon(POINT *pIn, int N, POINT *pOut, int &Cnt,
                 RECT rWin)
{

    POINT pTmp[20];

    _fmemcpy(pTmp, pIn, (N+1)*sizeof(POINT));
    ClipEdge(pTmp, N, pOut, Cnt, LEFT, rWin);
    N = Cnt;
    _fmemcpy(pTmp, pOut, (N+1)*sizeof(POINT));
    ClipEdge(pTmp, N, pOut, Cnt, RIGHT, rWin);
    N = Cnt;
    _fmemcpy(pTmp, pOut, (N+1)*sizeof(POINT));
    ClipEdge(pTmp, N, pOut, Cnt, TOP, rWin);
    N = Cnt;
    _fmemcpy(pTmp, pOut, (N+1)*sizeof(POINT));
    ClipEdge(pTmp, N, pOut, Cnt, BOTTOM, rWin);
} // ClipPolygon
```

***Nhận xét***

Thuật toán Sutherland-Hodgeman cho kết quả rất chính xác khi làm việc với các đa giác lồi, tuy nhiên với các đa giác lõm kết quả hiển thị có thể sẽ có đoạn thừa như hình 4.11. Điều này xảy ra khi đa giác sau khi xén bị tách thành hai hay nhiều vùng. Do chúng ta chỉ lưu kết quả xuất trong một danh sách các đỉnh nên đỉnh cuối của danh sách ứng với đa giác trước sẽ nối với đỉnh đầu của danh sách ứng với đa giác sau. Một trong nhiều cách để khắc phục điểm này là phân đa giác lõm thành hai hay nhiều đa giác lồi và xử lí mỗi đa giác lồi riêng.

## TÓM TẮT

Hiển thị đối tượng là quá trình đưa các mô tả đối tượng từ thế giới thực sang một thiết bị xuất cụ thể nào đó. Quy trình này bắt đầu bằng cách định nghĩa từng đối tượng thành phần trong hệ tọa độ cục bộ và kết thúc bằng việc chuyển toàn bộ đối tượng lên hệ tọa độ thiết bị. Bằng cách đưa ra định nghĩa hệ tọa độ quan sát và các khái niệm cửa sổ, vùng quan sát; mỗi đối tượng có thể được quan sát ở nhiều vị trí và góc độ khác nhau. Thông thường mỗi hình ảnh mà chúng ta quan sát được trên màn hình thiết bị được gọi là một thể hiện (view) của đối tượng.

Quá trình ánh xạ một vùng định nghĩa trong hệ tọa độ thế giới thực vào một vùng trong hệ tọa độ thiết bị được gọi là phép biến đổi hệ quan sát. Đây thực chất là phép biến đổi hệ tọa độ. Quá trình loại bỏ các phần hình ảnh nằm ngoài một vùng cho trước được gọi là xén hình. Vùng được dùng để xén hình gọi là cửa sổ xén.

Các thuật toán xén đoạn thẳng Cohen-Sutherland, Liang-Barsky đều tập trung giải quyết hai vấn đề chính nhằm tối ưu tốc độ thuật toán đó là : loại bỏ việc tìm giao điểm đối với các đoạn thẳng chắc chắn không cắt cửa sổ (như nằm hoàn toàn trong, nằm hoàn toàn ngoài), và với các đoạn có khả năng cắt cửa sổ, cần phải tìm cách hạn chế số lần cần tìm giao điểm với các biên không cần thiết. Thuật toán Cohen-Sutherland giải quyết hai ý này thông qua kiểu dữ liệu mã vùng mà về mặt bản chất đó chỉ là sự mô tả vị trí tương đối của vùng đang xét so với các biên của cửa sổ. Còn thuật toán Liang-Barsky thì tuy dựa vào phương trình tham số của đoạn thẳng để lập luận nhưng thực chất là dựa trên việc xét các giao điểm có thể có giữa đoạn thẳng kéo dài với các biên của cửa sổ (cũng được kéo dài). Các giao điểm này tương ứng với các giá trị $r_k = q_k / p_k$ . Cả hai thuật toán này đều có thể được mở rộng cho việc xén hình trong đồ họa ba chiều.

## BÀI TẬP

**46.** Phân tích các hệ tọa độ dùng trong quy trình hiển thị đối tượng hai chiều.

**47.** Viết đoạn chương trình minh họa quá trình chuyển đổi từ cửa sổ sang vùng quan sát.

**48.** Ý nghĩa của mã vùng trong thuật toán Cohen-Sutherland.

**49.** Hãy cho một đoạn thẳng minh họa mà trong trường hợp này thuật toán phải thực hiện việc tìm giao điểm 4 lần theo thứ tự LEFT, TOP, RIGHT, BOTTOM.

**50.** Cài đặt thuật toán Cohen-Sutherland để xén một đa giác. Phân tích các trường hợp thuật toán này cho kết quả là các đoạn thẳng rời rạc.

**51.** So sánh hai thuật toán Cohen-Sutherland và Liang-Barsky về số phép toán thực hiện trong các trường hợp chính.

**52.** Cài đặt thuật toán Liang-Barsky và so sánh với tốc độ thuật toán Cohen-Sutherland.

**53.** Hiệu chỉnh các thuật toán xén đoạn thẳng đã học để có thể xén đoạn thẳng vào cửa sổ hình chữ nhật nghiêng với trục hoành một góc $\alpha$.

**54.** Trình bày và cài đặt thuật toán phân một đa giác lõm thành các đa giác lồi.

**55.** Dựa trên kết quả của bài tập trên, hiệu chỉnh thuật toán Sutherland-Hodgeman để xén các đa giác lõm được chính xác.

***CHƯƠNG 5***

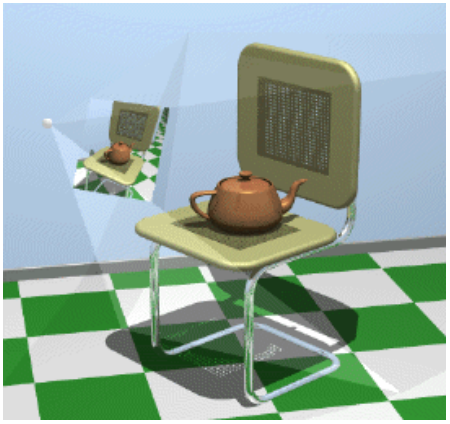
# GIỚI THIỆU ĐỒ HỌA BA CHIỀU

Các đối tượng trong thế giới thực phần lớn là các đối tượng ba chiều, nên việc thể hiện các đối tượng ba chiều trên máy tính là một công việc hết sức cần thiết để đưa tin học gần gũi với thực tế hơn. Cũng giống như các cách biểu diễn các đối tượng ba chiều trên mặt phẳng khác (như của máy ảnh, camera, ... ), biểu diễn bằng máy tính cũng phải tuân theo các quy luật về phối cảnh, sáng, tối, ... nhằm giúp người xem có thể tưởng tượng lại hình ảnh một cách gần đúng nhất. Ngoài ra biểu diễn trên máy tính có ưu thế giúp ta có thể quan sát đối tượng ở nhiều góc cạnh khác nhau, ở các khoảng cách khác nhau.

Chương này sẽ giới thiệu một số kĩ thuật biểu diễn các đối tượng ba chiều trên máy tính, từ các đối tượng đơn giản như các hình khối, các đa diện, ... đến các đối tượng tương đối phức tạp như các mặt đã được tìm hiểu ở các chương trước.

## 9. TỔNG QUAN VỀ ĐỒ HỌA BA CHIỀU

Khi chúng ta mô hình hóa và hiển thị một cảnh ba chiều, ta cần phải xem xét rất nhiều khía cạnh và vấn đề khác nhau chứ không đơn giản là thêm vào tọa độ thứ ba cho các đối tượng. Bề mặt đối tượng có thể xây dựng bởi nhiều tổ hợp khác nhau của các mặt phẳng và các mặt cong. Ngoài ra, đôi khi chúng ta cũng cần mô tả một số thông tin về bên trong các đối tượng. Các công cụ hỗ trợ đồ họa (graphics package) thường cung cấp một số hàm hiển thị các thành phần bên trong, những đường nét tiêu biểu hoặc hiển thị một phần của đối tượng ba chiều (solid object). Ngoài ra, các phép biến đổi hình học thường được sử dụng nhiều hơn và đa dạng hơn trong đồ họa ba chiều so với trong đồ họa hai chiều. Phép biến đổi hệ quan sát trong không gian ba chiều phức tạp hơn nhiều so với trong không gian hai chiều do chúng ta phải chọn lựa nhiều tham số hơn khi mô tả một cảnh ba chiều sẽ xuất hiện trên màn hình như thế nào.



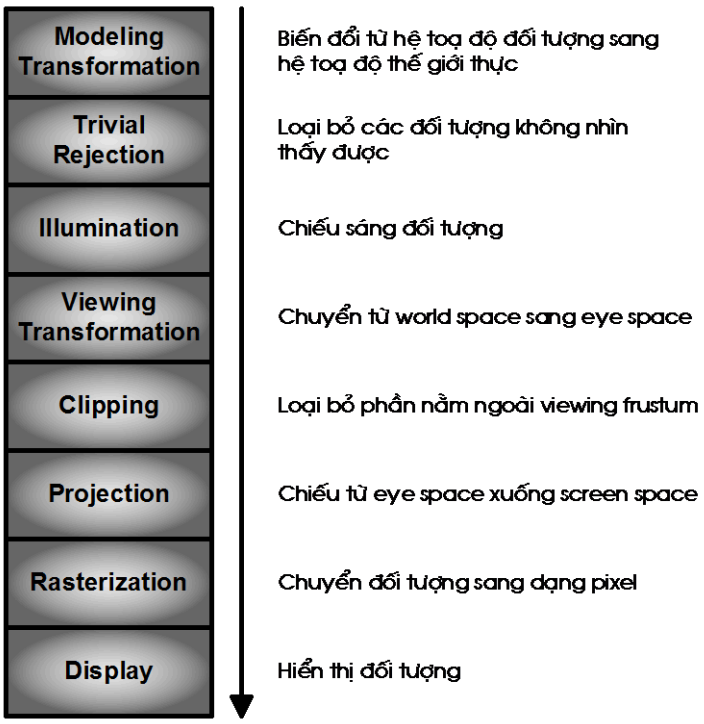
**Hình 5.1** – Một cảnh đồ họa ba chiều

Các mô tả về một cảnh ba chiều phải đi qua một quy trình xử lí gồm nhiều công đoạn như phép biến đổi hệ tọa độ quan sát và phép chiếu chuyển cảnh từ hệ tọa độ quan sát ba chiều xuống hệ tọa độ thiết bị hai chiều. Những phần nhìn thấy được của cảnh, ứng với một hệ quan sát được chọn nào đó, phải được xác định và cuối cùng, các thuật toán vẽ mặt sẽ được áp dụng nhằm tạo ra hình ảnh trung thực (gần với thực tế) của cảnh.

### 9.1. Sơ lược về quy trình hiển thị

Quy trình xử lí thông tin trong đồ họa ba chiều là một chuỗi các bước nối tiếp nhau, kết quả của mỗi bước sẽ là đầu vào của bước tiếp theo.

**Hình 5.2** – Quy trình hiển thị đối tượng ba chiều



Quy trình bắt đầu bằng việc xây dựng các mô hình đối tượng. Các mô hình này thường được mô tả trong không gian ba chiều (x,y,z). Các mô hình thường thể hiện vật thể (solid) hoặc bề mặt (boundaries) của đối tượng. Như vậy ta có hai kiểu mô hình hóa. Trong *solid modeling* các đối tượng đồ họa cơ sở thường được dùng để mô tả các đối tượng có thể tích (volume). Trong *boundary representations(B-reps)*, các đối tượng được định nghĩa bởi bề mặt của chúng.

Các mô hình thường được biểu diễn trong một hệ tọa độ cục bộ, mà ta gọi là hệ tọa độ đối tượng. Trong hệ tọa độ này chỉ có bản thân đối tượng được định nghĩa, vì vậy gốc tọa độ và đơn vị đo lường thường được chọn sao cho việc biểu diễn đối tượng tiện lợi nhất.

Bước đầu tiên trong quy trình hiển thị là biến đổi đối tượng từ không gian đối tượng (*object-space*) vào một không gian chung gọi là không gian thực (*world space*). Trong không gian này các đối tượng, nguồn sáng, và người quan sát cùng tồn tại. Bước này được gọi là giai đoạn *biến đổi mô hình* (*modeling transformation*).

Bước tiếp theo là một bước tối ưu hóa. Trong giai đoạn loại bỏ đơn giản (trivial rejection) ta cần loại trừ tất cả các đối tượng không thể nhìn thấy. Điều này giúp chúng ta tránh được việc xử lí một số phần không cần thiết của cảnh (scene) mà ta đang chuẩn bị hiển thị ở các bước sau.

Tiếp theo ta phải chiếu sáng (illumination) các đối tượng có thể nhìn thấy được bằng cách gán cho chúng màu sắc dựa trên các đặc tính của các chất tạo nên vật và các nguồn sáng tồn tại trong cảnh.

Sau khi chiếu sáng, ta phải thực hiện một phép biến đổi hệ tọa độ để đặt vị trí quan sát (viewing position) về gốc tọa độ và mặt phẳng quan sát (viewing plane) về một vị trí mong ước. Bước này gọi là bước đổi hệ quan sát. Sau bước này, các đối tượng được chuyển từ không gian thực sang không gian quan sát (*eye space*).

Trong không gian quan sát, ta phải thực hiện việc xén các đối tượng trong cảnh để cảnh nằm gọn trong một phần không gian chóp cụt mà ta gọi là ***viewing frustum***. Bước này sẽ loại bỏ hoàn toàn các đối tượng (các mảnh đối tượng) không nhìn thấy được trong ảnh.

Bước tiếp theo ta sẽ chiếu các đối tượng xuống mặt phẳng hai chiều. Bước Projection thực hiện phép biến đổi từ không gian quan sát sang không gian màn hình (*screen-space*).

Trong bước rời rạc hóa (rasterization) ta sẽ chuyển đối tượng thành các pixel. Cuối cùng, toàn cảnh sẽ được hiển thị lên màn hình.

## 9.2. Mô hình khung nối kết (Wireframe Model)

### 9.2.1. Khái niệm

Một phương pháp thông dụng và đơn giản để mô hình hóa đối tượng là mô hình khung nối kết.

Một mô hình khung nối kết gồm có một tập các đỉnh và tập các cạnh nối giữa các đỉnh đó. Khi thể hiện bằng mô hình này, các đối tượng ba chiều có vẻ rỗng và không giống thực tế lắm. Để hoàn thiện hơn, người ta dùng các kĩ thuật tạo bóng và loại bỏ các đường và mặt khuất. (Chúng ta sẽ đề cập vấn đề này ở các chương sau). Tuy nhiên vẽ bằng mô hình này thường nhanh nên người ta thường dùng nó trong việc xem phác thảo (preview) các đối tượng, đặc biệt là trong các hệ CAD.

### 9.2.2. Biểu diễn các vật thể ba chiều bằng mô hình khung nối kết

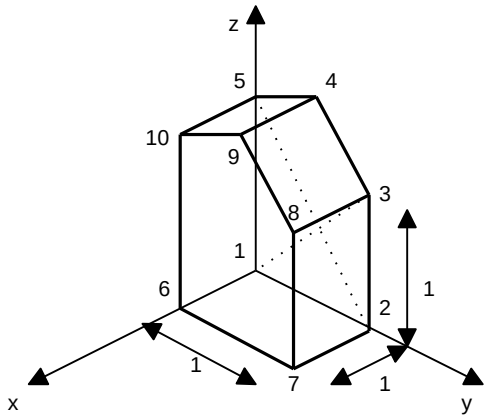
Với mô hình khung nối kết, hình dạng của đối tượng ba chiều được biểu diễn bằng hai danh sách (list) : danh sách các đỉnh (vertices) và danh sách các cạnh (edges) nối các đỉnh đó. Danh sách các đỉnh cho biết thông tin hình học đó là vị trí các đỉnh, còn danh sách các cạnh xác định thông tin về sự kết nối, nó cho biết cặp các đỉnh tạo ra cạnh. Chúng ta hãy quan sát một vật thể ba chiều được biểu diễn bằng mô hình khung nối kết như sau :

***Bảng danh sách các cạnh và đỉnh biểu diễn vật thể***

| Vertex List | | | | |
|---|---|---|---|---|
| **Vertex** | **x** | **y** | **z** | |
| 1 | 0 | 0 | 0 | back side |
| 2 | 0 | 1 | 0 | |
| 3 | 0 | 1 | 1 | |
| 4 | 0 | 0.5 | 1.5 | |
| 5 | 0 | 0 | 1 | |
| 6 | 1 | 0 | 0 | front side |
| 7 | 1 | 1 | 0 | |
| 8 | 1 | 1 | 1 | |
| 9 | 1 | 0.5 | 1.5 | |
| 10 | 1 | 0 | 1 | |

| Edge List | | |
|---|---|---|
| **Edge** | **Vertex1** | **Vertex2** |
| 1 | 1 | 2 |
| 2 | 2 | 3 |
| 3 | 3 | 4 |
| 4 | 4 | 5 |
| 5 | 5 | 1 |
| 6 | 6 | 7 |
| 7 | 7 | 8 |
| 8 | 8 | 9 |
| 9 | 9 | 10 |
| 10 | 10 | 6 |
| 11 | 1 | 6 |

| | | |
|---|---|---|
| **12** | **2** | **7** |
| **13** | **3** | **8** |
| **14** | **4** | **9** |
| **15** | **5** | **10** |
| **16** | **2** | **5** |
| **17** | **1** | **3** |




**Hình 5.3** – Vật thể ba chiều được biểu diễn bằng mô hình khung nối kết

Có nhiều cách để đặc tả mô hình khung nối kết trên máy tính như dùng xâu, mảng, ... và mỗi cách đều có các ưu điểm riêng trong từng ứng dụng cụ thể. Ở đây ta minh họa các biểu diễn mô hình khung nối kết bằng cấu trúc dữ liệu mảng như sau :

```
#define MAXVERTS        50 //số đỉnh tối đa có thể biểu diễn
#define MAXEDGES        100 //số cạnh tối đa

typedef struct {
        float       x, y, z;
} POINT3D;

typedef struct {
        int         NumVerts; //Số đỉnh trong mô hình
        int         NumEdges; //Số cạnh trong mô hình
        POINT3D     Vert[MaxVerts];
        int         Edge[MaxEdges][2];
}WIREFRAME;
```

Ngoài ra, đôi khi trong mô hình wireframe người ta còn mô tả các mặt (phẳng) của đối tượng. Mỗi mặt được định nghĩa bởi một đa giác bao. Ví dụ, đối tượng trong hình 5.3 có 7 mặt.

### 9.3. Vẽ các đối tượng theo mô hình khung nối kết bằng cách sử dụng các phép chiếu

Để vẽ các đối tượng biểu diễn bằng mô hình khung nối kết, đơn giản chúng ta chỉ cần vẽ các cạnh trong danh sách các cạnh mà thôi. Tuy nhiên do các đỉnh và cạnh đều được định nghĩa trong ba chiều nên vấn đề đặt ra ở đây là làm thế nào để vẽ các đường thẳng ba chiều trong mặt phẳng hai chiều. Để làm điều này, chúng ta phải thực hiện phép chiếu từ ba chiều vào hai chiều để bỏ bớt một chiều. Có hai loại phép chiếu đơn giản thường dùng đó là phép chiếu song song (parallel projection) và phép chiếu phối cảnh (perspective projection). Phép chiếu song song sử dụng các đường thẳng song song đi qua các đỉnh của đối tượng, trong khi đó phép chiếu phối cảnh dùng các đường thẳng qua các đỉnh của đối tượng hội tụ về một điểm gọi là tâm chiếu (center of projection). Các đường thẳng trên được gọi là tia chiếu và giao điểm của các đường thẳng này với mặt phẳng chiếu (hay còn gọi là mặt phẳng quan sát (view plane)) chính là các hình chiếu của các đỉnh hay còn gọi là điểm chiếu. Trong phần này, chúng ta giả sử rằng mặt phẳng chiếu là mặt phẳng z=0.

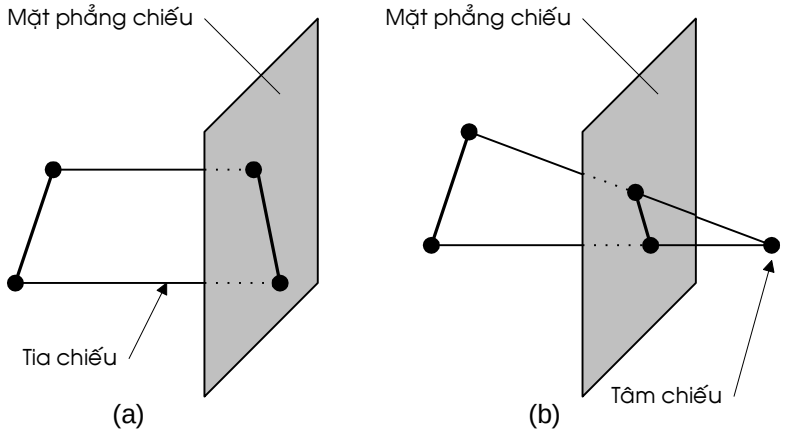
Phép chiếu song song bảo toàn được mối quan hệ giữa các chiều của đối tượng, đây chính là kĩ thuật được dùng trong phác thảo để tạo ra phần khung của đối tượng ba chiều. Người ta dùng phương pháp này để quan sát chính xác ở các mặt khác nhau của đối tượng. Tuy nhiên, phép chiếu song song không cho một biểu diễn thực của đối tượng ba chiều.

Trong khi đó, phép chiếu phối cảnh tạo ra được biểu diễn thực hơn nhưng lại không bảo toàn được mối liên hệ giữa các chiều. Các đường thẳng càng xa sẽ có các ảnh chiếu nhỏ hơn.

Nói chung, kĩ thuật để vẽ một đường thẳng ba chiều là :

- Chiếu mỗi điểm đầu mút thành các điểm hai chiều.

- Vẽ đường thẳng nối hai điểm ảnh qua phép chiếu.




**Hình 5.4** – Phép chiếu song song (a) và phép chiếu phối cảnh (b)

Sở dĩ chúng ta làm được điều này vì các phép chiếu mà chúng ta sử dụng bảo toàn đường thẳng.

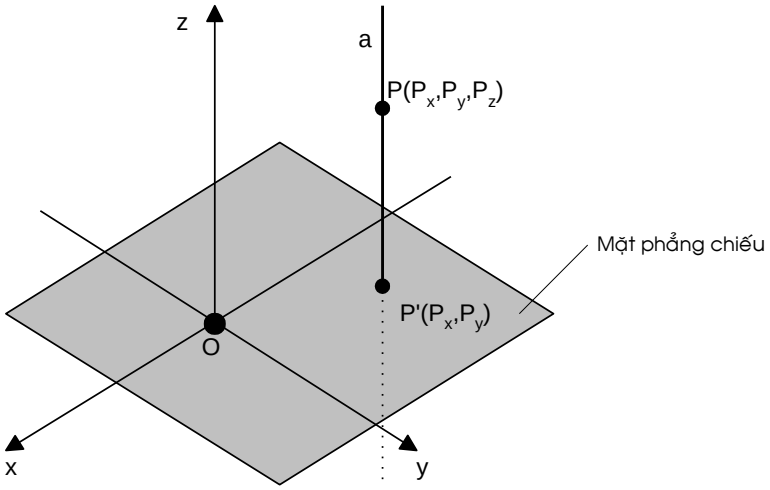
## 9.4. Phép chiếu song song (parallel projection)

Khi hướng của tia chiếu vuông góc với mặt phẳng chiếu ta có phép chiếu trực giao (orthographic projection). Ngược lại, ta có phép chiếu xiên (oblique projection).

### 9.4.1. Phép chiếu trực giao

Xét điểm ba chiều, $P = (P_x, P_y, P_z)$, cách đơn giản nhất là bỏ đi thành phần z để chiếu P thành $P' = (P_x, P_y)$. Điều này tương đương với chiếu điểm đó lên mặt phẳng xy theo phương của trục z. Mặt phẳng xy là mặt phẳng quan sát. Xem hình vẽ minh họa 5.5, ở đây điểm chiếu chính là giao điểm của tia a qua P và song song với trục z vuông góc với mặt phẳng xy. Tia a là tia chiếu.

Dễ dàng thấy rằng phép chiếu này bảo toàn đường thẳng.

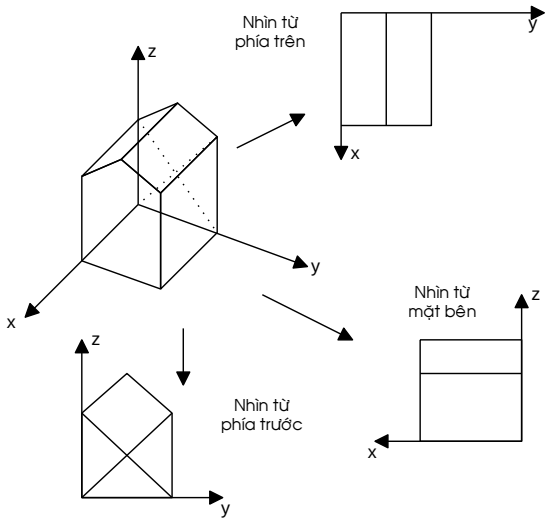



**Hình 5.5** – Phép chiếu trực giao

Phép chiếu trực giao ở trên thường được gọi là phép nhìn từ trên xuống (top-view) hoặc dưới lên (bottom-view). Có hai phép chiếu khác cũng khá thông dụng là:

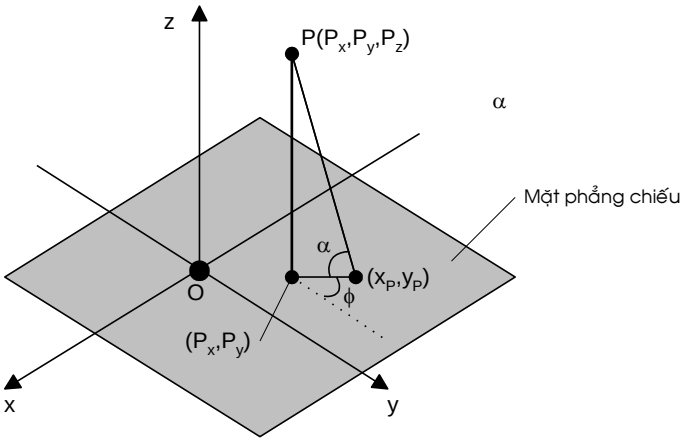
- Phép nhìn từ phía trước (front-view): Tia chiếu song song với trục x và mặt phẳng quan sát là yz. Phép chiếu này loại bỏ thành phần x của P.
- Phép nhìn từ phía bên cạnh (side-view): Tia chiếu song song với trục y và mặt phẳng quan sát là xz. Phép chiếu này loại bỏ thành phần y của P.

Hình 5.6 minh họa ba phép chiếu trực giao đã đề cập ở trên lên một vật thể là ngôi nhà. Nhận xét rằng với phép chiếu nhìn từ phía trước ta không phân biệt được tường trước và tường sau vì chúng nằm chồng lên nhau, cũng tương tự cho trường hợp phép chiếu nhìn từ phía bên cạnh.




**Hình 5.6** – Kết quả của ba phép chiếu trực giao

### 9.4.2. Phép chiếu xiên




**Hình 5.7** – Phép chiếu xiên

Hình 5.7 minh họa một phép chiếu xiên. Điểm $P(P_x, P_y, P_z)$ qua phép chiếu xiên sẽ nhận được điểm $P'(x_P, y_P)$. $(P_x, P_y)$ là hình chiếu của P qua phép chiếu trực giao. α là góc hợp bởi tia chiếu và đoạn nối $(x_P, y_P)$ và $(P_x, P_y)$. Giả sử đoạn nối này có độ dài là L. ϕ là góc giữa đoạn nối trên với trục y.

## 9.5. Phép chiếu phối cảnh (perspective projection)

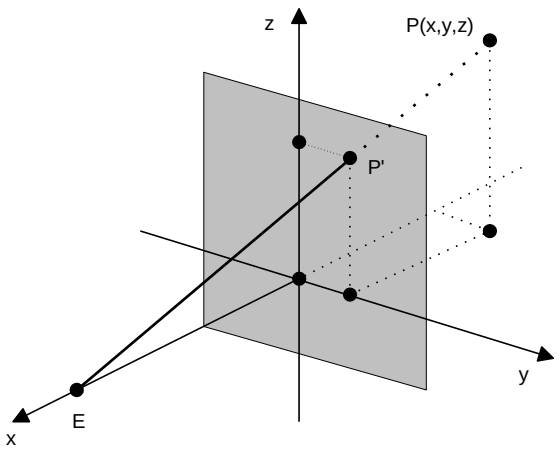
### 9.5.1. Phép chiếu phối cảnh đơn giản nhất

Phép chiếu phối cảnh phụ thuộc vào vị trí tương đối của hai đối tượng đó là mắt nhìn và mặt phẳng quan sát.

Quan sát hình sau, với mặt phẳng quan sát là yz và mắt nhìn $E(E,0,0)$ được đặt dọc theo trục x. Khoảng cách giữa mắt E và mặt phẳng quan sát được gọi là tầm nhìn (eye distance).Để xác định hình chiếu của $P(x,y,z)$, ta nối P với E và tìm giao điểm P' của đường thẳng này với mặt phẳng quan sát. Lúc này P' chính là điểm cần tìm.

**Hình 5.8** – Phép chiếu phối cảnh đơn giản

Trong phép chiếu phối cảnh các tia chiếu không song song với nhau mà hội tụ về một điểm duy nhất là mắt.



Chúng ta giả sử P không nằm phía sau mắt nhìn, tức là $x < E$. P có thể nằm sau mặt phẳng quan sát, hay trên mặt phẳng quan sát, hay giữa mắt và mặt phẳng quan sát. Ta có, tia từ mắt đến P có dạng :

$$r(t) = (E,0,0)(1-t) + (x,y,z)t$$

Tia này giao với mặt quan sát (mặt phẳng x=0) khi x = 0 nên giá trị t ứng với trường hợp này là :

$$t' = \frac{1}{(1 - x/E)}$$

Suy ra các tọa độ của điểm chiếu là :

$$y' = \frac{y}{(1 - x/E)}$$

$$z' = \frac{z}{(1 - x/E)}$$

Phép chiếu phối cảnh gần giống phép chiếu trực giao chỉ khác là hai tọa độ y, z được nhân lên thêm một lượng là $t' = \frac{1}{(1 - x/E)}$. Hệ số tỉ lệ này dẫn đến khái niệm phối cảnh theo luật xa gần (perspective foreshortening) nghĩa là : vật càng xa mắt (theo chiều âm của trục x, để luôn có $x < E$) thì t' càng nhỏ dẫn đến y', z' càng nhỏ do đó vật sẽ thấy nhỏ hơn, ngược lại nếu vật càng gần mắt thì sẽ thấy lớn hơn.

Cũng tương tự như trên, ta có thể dễ dàng kiểm chứng phép chiếu phối cảnh cũng bảo toàn đường thẳng.

Nhận xét rằng phép chiếu song song là một trường hợp đặc biệt của phép chiếu phối cảnh. Nếu chúng ta cho tầm nhìn E càng ngày càng lớn tiến dần đến vô cực thì các tia chiếu qua mắt sẽ trở nên song song và hệ số $t' = \frac{1}{(1 - x/E)}$ trở thành 1. Lúc này phép chiếu phối cảnh trở thành phép chiếu song song.

#### 9.5.2. Các trường hợp khác

Trường hợp trên có mắt nhìn nằm trên trục x và mặt phẳng quan sát là yz. Cũng như trong phép chiếu trực giao ta cũng có thể hoán đổi vị trí của mắt và mặt phẳng quan sát để có thể nhìn đối tượng ở các góc cạnh khác nhau. Ví dụ trong trường hợp mắt nằm trên trục z và mặt phẳng quan sát là xy thì các kết quả sẽ tương tự :

$$t' = \frac{1}{1 - z/E}$$

$$x' = \frac{x}{1 - z/E}$$

$$y' = \frac{y}{1 - z/E}$$

## 10. BIỂU DIỄN ĐỐI TƯỢNG BA CHIỀU

Các cảnh đồ họa có thể chứa nhiều dạng đối tượng khác nhau: cây, hoa, mây, núi, nước, sắt thép, cỏ, …Chính vì vậy, không ngạc nhiên khi có nhiều phương pháp khác nhau có thể sử dụng để mô tả các đối tượng sao cho phù hợp với thuộc tính của các loại đối tượng này. Các mặt đa giác và mặt bậc hai cung cấp cho chúng ta một mô tả gần đúng của các đối tượng Euclid đơn giản như là các khối ellipse, khối đa diện; các mặt tròn xoay, và các đối tượng dùng để thiết kế các mô hình máy bay, bánh răng và các cấu trúc công nghệ khác thường được biểu diễn thông qua mặt cong (curves); các phương pháp tiếp cận thủ tục (procedural method) như Fractal cho phép chúng ta biểu biễn một cách chính xác các đối tượng như mây, thảm cỏ và các đối tượng tự nhiên khác.

Sơ đồ biểu diễn một đối tượng lập thể thường được chia ra làm hai loại, dù không phải tất cả các biểu diễn đều có thể được phân chia một cách rõ ràng thuộc loại nào trong hai loại này. Phương pháp biểu diễn bề mặt ((B-reps)) mô tả các đối tượng ba chiều bằng một tập hợp các bề mặt giới hạn phần bên trong của đối tượng với môi trường bên ngoài. Ví dụ kinh điển của B-reps là việc biểu diễn các mặt đa giác và các mảnh tròn xoay. Phương pháp biểu diễn theo phân hoạch không gian (space-partitioning representation) thường được dùng để mô tả các thuộc tính bên trong của đối tượng bằng cách phân hoạch phần bên trong của đối tượng thành một tập hợp nhiều đối tượng nhỏ hơn.

Trong đồ họa máy tính, các đối tượng lập thể có thể được mô tả bằng các bề mặt (surfaces) của chúng. Ví dụ : Một hình lập phương được xây dựng từ sáu mặt phẳng, một hình trụ được xây dựng từ sự kết hợp của một mặt cong và hai mặt phẳng, và hình cầu được xây dựng chỉ từ một mặt cong. Thông thường để biểu diễn một đối tượng bất kì , người ta dùng các phương pháp xấp xỉ để đưa các mặt về dạng các mặt đa giác (polygon faces). Tuy nhiên trong trường hợp các đối tượng thực sự phức tạp, người ta thường dùng một hay nhiều mặt cong trơn (smoothly curved surfaces) ghép nối lại với nhau. Mỗi thành phần dùng để ghép nối được gọi là patch (mặt vá).

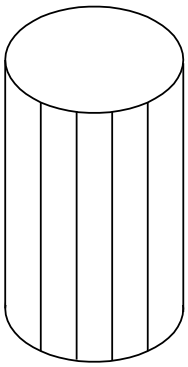
## 10.1. Biểu diễn mặt đa giác

Phương pháp B-reps chung nhất thường dùng để biểu diễn các đối tượng ba chiều là dùng một tập hợp các mặt đa giác xác định bề mặt của đối tượng. Rất nhiều hệ thống đồ họa lưu trữ các đối tượng như là một tập hợp các mặt đa giác. Với cách biểu diễn này ta có thể đơn giản hóa việc biểu diễn và tăng tốc độ hiển thị các đối tượng bởi vì tất cả các bề mặt đều được mô tả bởi các phương trình tuyến tính. Vì lí do này, mô tả các đối tượng thông qua các mặt đa giác thường được dùng cho các đối tượng đồ họa cơ sở.

Trong một số trường hợp, ta chỉ có một khả năng chọn lựa là sử dụng biểu diễn đa giác. Tuy nhiên, một số hệ thống đồ họa còn cho phép các khả năng biểu diễn khác ví dụ như bằng các mặt cong spline.

**Hình 5.9** – Mô hình wireframe của một hình trụ

Biểu diễn bằng mặt đa giác của các đa diện cho chúng ta một định nghĩa chính xác về các đặc tính của các đối tượng này.



Nhưng đối với những đối tượng khác ta chỉ nhận được một biểu diễn gần đúng. Hình 5.9 cho chúng ta biểu diễn một hình trụ như là một tập hợp các mặt đa giác. Biểu diễn dạng wireframe cho phép chúng ta hiển thị đối tượng rất nhanh. Khi cần thể hiện đối tượng thực hơn, ta có thể dùng kĩ thuật tạo bóng nội suy (interpolating shading).

### 10.1.1. Biểu diễn bằng bảng đa giác

Ta biểu diễn một mặt đa giác bằng một tập hợp các đỉnh và các thuộc tính kèm theo. Khi thông tin của mỗi mặt đa giác được nhập, dữ liệu sẽ được điền vào trong các bảng sẽ được dùng cho các xử lí tiếp theo, hiển thị và biến đổi. Các bảng dữ liệu mô tả mặt đa giác có thể tổ chức thành hai nhóm: các bảng hình học và các bảng thuộc tính. Các bảng lưu trữ dữ liệu hình học chứa tọa độ của các đỉnh và các tham số cho biết về định hướng trong không gian của mặt đa giác. Thông tin về thuộc tính của các đối tượng chứa các tham số mô tả độ trong suốt, tính phản xạ và các thuộc tính texture của đối tượng.

Một cách tổ chức thuận tiện để lưu trữ các dữ liệu hình học là tạo ra ba danh sách: một bảng lưu đỉnh, một bảng lưu cạnh và một bảng lưu đa giác. Các giá trị tọa độ cho mỗi đỉnh trong đối tượng được chứa trong bảng lưu đỉnh. Bảng cạnh chứa các con trỏ trỏ đến bảng đỉnh cho biết đỉnh nào được nối với một cạnh của đa giác. Và cuối cùng, bảng lưu đa giác chứa các con trỏ trỏ tới bảng lưu cạnh cho biết những cạnh nào tạo nên đa giác.

Ngoài ra, ta cũng có thể thêm một số thông tin bổ sung vào các bảng trên để xử lí nhanh hơn khi cần truy xuất thông tin. Ví dụ, ta có thể thêm một con trỏ từ một cạnh đến các đa giác chứa nó. Tương tự, ta có thể thêm thông tin trong bảng lưu đỉnh để biết những cạnh nào kề với một đỉnh cho trước ….

Vì các bảng lưu thông tin về đối tượng có thể rất phức tạp nên việc kiểm tra tính đúng đắn và đầy đủ của dữ liệu là rất quan trọng.

### 10.1.2. Phương trình mặt phẳng

Để thực hiện việc hiển thị một đối tượng ba chiều, ta phải xử lí dữ liệu nhập thông qua một quy trình gồm nhiều bước. Trong một số bước này, đôi khi ta cần thông tin về định hướng của đối tượng và cả thông tin về định hướng của từng mặt của đối tượng trong không gian. Những thông tin này có thể lấy được thông qua tọa độ của các đỉnh và phương trình mô tả các mặt đa giác của đối tượng.

Phương trình biểu diễn mặt phẳng có dạng:

$$Ax + By + Cz + D = 0 \qquad (5.1)$$

trong đó $(x, y, z)$ là một điểm bất kì của mặt phẳng và A, B, C, D là các hằng số diễn tả thông tin không gian của mặt phẳng. Như đã biết, để xác định phương trình mặt phẳng, ta chỉ cần biết ba điểm không thẳng hàng trên mặt phẳng này. Như vậy, để xác định phương trình mặt phẳng qua một đa giác, ta sẽ sử dụng tọa độ của ba đỉnh đầu tiên $(x_1, y_1)$, $(x_2, y_2)$, $(x_3, y_3)$, trong đa giác này. Từ (5.1) ta có:

$$Ax_k + By_k + Cz_k + D = 0, \quad k = 0,1,2,3 \qquad (5.2)$$

Dùng quy tắc Cramer, ta có thể xác định A, B, C, D theo công thức:

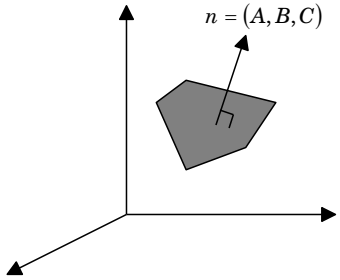
$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$

$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \tag{5.3}$$

Khai triển các định thức trên ta được công thức tường minh của các hệ số:

$$\begin{cases} A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1) \end{cases} \tag{5.4}$$

Hướng của mặt phẳng thường được xác định




thông qua vector pháp tuyến của nó. Vector pháp tuyến $n = (A, B, C)$, trong đó A, B, C là các hệ số của phương trình mặt phẳng ta vừa tính trong (5.4).

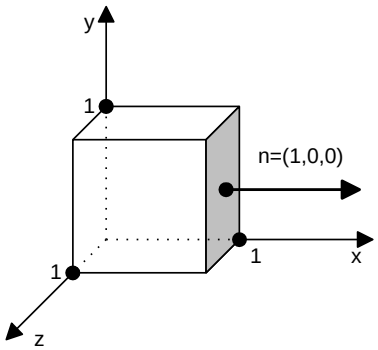
**Hình 5.10** – Vector pháp tuyến của mặt phẳng

Vì ta thường làm việc với các mặt bao quanh đối tượng nên ta cần phân biệt hai mặt của mặt phẳng. Mặt tiếp giáp với phần bên trong của đối tượng ta gọi là mặt trong, mặt kia là mặt ngoài. Nếu các cạnh của đa giác được mô tả theo chiều ngược chiều kim đồng hồ, vector pháp tuyến của mặt phẳng sẽ hướng từ trong ra ngoài (giả sử hệ tọa độ biểu diễn đối tượng là hệ tọa độ bàn tay phải). Ví dụ, trong hình 5.11, vector pháp tuyến của mặt phải của khối lập phương đơn vị (mặt được tô) có phương trình mặt phẳng là x-1=0 và có vector pháp tuyến tương ứng là (1,0,0);

**Hình 5.11** – Vector pháp tuyến hướng từ trong ra ngoài

Phương trình mặt phẳng còn có thể dùng để xác định vị trí tương đối giữa một điểm trong không gian với mặt phẳng. Nếu điểm P(x,y,z) không nằm trên mặt phẳng, lúc đó ta có:

$Ax + By + Cz + D \neq 0$




Ta có thể xác định P nằm ở phía trong hay nằm phía ngoài của mặt phẳng nhờ vào dấu của biểu thức Ax + By + Cz + D:

- Nếu Ax + By + Cz + D < 0, điểm P(x,y,z) nằm trong.
- Nếu Ax + By + Cz + D > 0, điểm P(x,y,z) nằm ngoài.

Dấu hiệu kiểm tra trên đúng cho hệ tọa độ bàn tay phải và phương trình mặt phẳng được tính từ tọa độ các đỉnh đa giác cho theo chiều ngược chiều kim đồng hồ.

Đôi khi, sẽ rất hữu ích nếu ta khảo sát các đường và mặt thông qua phương trình tham số của nó.

Phương trình tham số của một mặt là một phương trình có hai tham số u, v. Một điểm bất kì trên mặt sẽ có tọa độ được biểu diễn dưới dạng vector tham số : p(u, v) = (x(u, v), y(u, v), z(u, v)). Với mỗi cặp giá trị (u, v) ta sẽ có một bộ các tọa độ (x, y, z) biểu diễn một điểm trên bề mặt đã cho. Các mặt sẽ được phân biệt với nhau bằng các bộ hàm x(), y(), z() khác nhau.

Để giới hạn không gian của các mặt, thông thường người ta định lại các tọa độ biên sao cho u, v tương ứng biến đổi trong đoạn $[0,1]$. Ví dụ, một mặt cầu với bán kính r, tâm tại gốc tọa độ có thể biểu diễn bằng các phương trình sau :

$x(u, v) = r \sin(\pi u) \cos(2\pi v)$

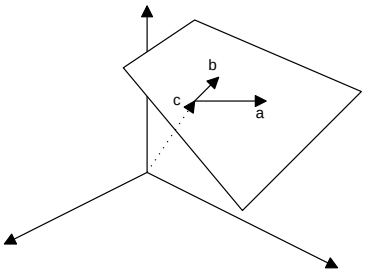
$y(u, v) = r \sin(\pi u) \sin(2\pi v)$

$z(u, v) = r \cos(\pi u)$

Trong đó u, v thay đổi trong đoạn $[0,1]$.

Một mặt phẳng có thể được xác định bằng một điểm với vector vị trí c và hai vector a, b không cùng phương (xem hình 5.12).

**Hình 5.12** – Minh họa cách xác định mặt phẳng

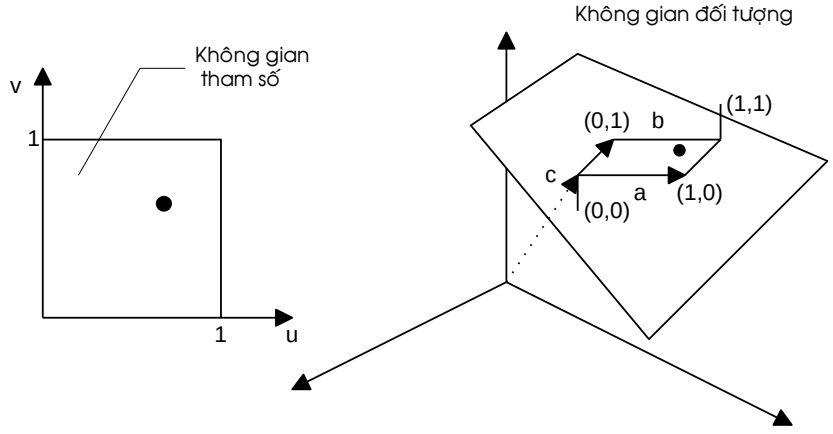



Nhận xét rằng bất kì điểm nào trên mặt phẳng cũng có thể được biểu diễn bằng một vector tổng sau : p(u, v) = c + au + bv. Đây chính là phương trình tham số của mặt phẳng.

Trong phương trình trên u, v có thể biến đổi trong khoảng $(-\infty,+\infty)$, do đó mặt phẳng sẽ trải dài đến vô tận. Tuy nhiên trong các trường hợp cụ thể ta chỉ muốn dùng một phần của mặt phẳng, một hình bình hành thôi chẳng hạn. Các phần như vậy được gọi là planar patch, lúc đó ta có thể tưởng tượng mặt phẳng như là sự ghép nối của các planar patch này.

Một planar patch được xác định bằng cách giới hạn khoảng biến đổi của các tham số u, v trong phương trình trên. Ví dụ nếu cho u, v biến đổi trong đoạn [0, 1] ta sẽ có một patch như hình vẽ 5.13.

Trong hình vẽ này u, v được biến đổi trong một khoảng không gian gọi là không gian tham số (parametric space), việc ánh xạ tương ứng các giá trị u, v đã được giới hạn trên sẽ tạo ra một patch trong không gian tương ứng gọi là không gian đối tượng.

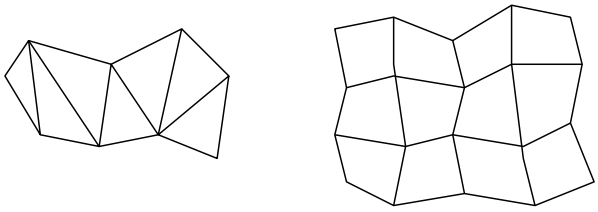



**Hình 5.13** – Cách tạo ra một patch

### 10.1.3. Lưới đa giác (polygon meshes)

Một số hệ đồ họa cung cấp một số hàm cho phép mô hình hóa các đối tượng. Một mặt phẳng có thể được diễn tả thông qua một hàm như *fillArea*. Nhưng khi ta cần lợp nhiều planar patch liên tiếp, dùng các hàm lưới (mesh function) sẽ thuận tiện hơn. Một dạng thông dụng của lưới đa giác là dãy các tam giác (triagle strip). Hàm này vẽ n-2 tam giác kề nhau khi biết n đỉnh. Dạng này của lưới đa giác dùng trong hầu hết các thư viện đồ họa chuẩn hiện nay như OpenGL hay DirectX. Một dạng hàm tương tự là lưới các tứ giác (quardrilateral mesh). Hàm này vẽ một lưới (n-1)x(m-1) tứ giác lồi từ dãy nxm đỉnh.

Khi đa giác được mô tả bởi nhiều hơn ba đỉnh, các đỉnh của nó có thể không đồng phẳng. Điều này có thể dẫn đến các lỗi tính toán. Một phương pháp đơn giản là phân đa giác này thành các tam giác.



**Hình 5.14 -** Triangle strip và quadrilateral mesh

## 10.2. Các đường cong và mặt cong

Hình ảnh của các đường cong và mặt cong có thể được tạo ra từ một tập hợp các hàm toán học định nghĩa các đối tượng hoặc từ một tập hợp các điểm trên đối tượng. Khi đối tượng được mô tả bằng các hàm toán học, thường các thư viện đồ họa cung cấp sẵn những hàm cho phép chiếu các đối tượng lên mặt phẳng hiển thị. Đối với các đường cong, các hàm này sẽ vẽ một loạt các điểm dọc theo hình chiếu của đường mô tả bởi hàm toán học. Đối với các mặt cong, một lưới đa giác xấp xỉ với mặt cong sẽ được tạo ra. Thường thì các hệ đồ họa tạo ra các lưới tam giác để đảm bảo tính đồng phẳng của các cạnh thuộc cùng một polygon patch.

Một đường cong hoặc mặt cong có thể được diễn tả bằng phương trình toán học dạng tham số hoặc không tham số. Tuy nhiên, trong đồ họa máy tính, thường thì dạng tham số sẽ thuận tiện cho xử lí hơn.

Khi đối tượng được mô tả bởi một tập hợp các điểm rời rạc, đối tượng sẽ được hiển thị thông qua một mặt cong xấp xỉ nào đó dựa trên những điểm đã cho. Các loại đường cong và mặt cong dạng spline hoặc Bezier là những đường cong và mặt cong xấp xỉ thường dùng.

Các mặt cong có thể có hình dạng rất phức tạp, đặc biệt khi nó bao gồm nhiều patch kết hợp lại với nhau. Trước tiên, chúng ta chỉ khảo sát các mặt cong khá đơn giản, kế tiếp chúng ta sẽ khảo sát các mặt phức tạp hơn.

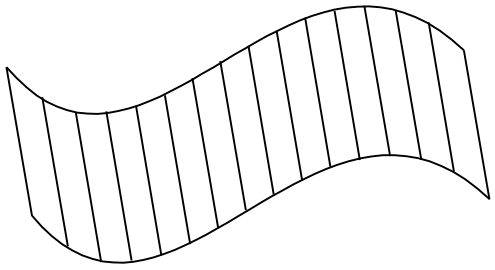
## 10.3. Các mặt có quy luật (ruled surfaces)

### 10.3.1. Định nghĩa

Ta có hai định nghĩa tương đương :

- Một mặt có quy luật là một mặt được tạo bằng cách quét (sweep) một đường thẳng trong không gian theo một cách nào đó.
- Một mặt được gọi là có quy luật nếu qua bất kì điểm nào thuộc nó đều có ít nhất một đường thẳng nằm hoàn toàn trên nó.

**Hình 5.15** – Minh họa một mặt có quy luật



### 10.3.2. Phương trình tham số

Vì mặt có quy luật hoàn toàn dựa trên cơ sở là đường thẳng với phương trình dạng tham số là $p(v) = (1-v)p_0 + vp_1$, nên ta có thể suy ra dạng của nó một cách tương tự như sau:

$$p(u,v) = (1-v)p_0(u) + vp_1(u) \qquad (5.5)$$

Nếu u biến đổi từ $u_{start}$ đến $u_{end}$ , ta thấy mặt cong sẽ là tập hợp của các đường thẳng nối các cặp điểm tương ứng $p_0(u')$ (thuộc đường cong $p_0(u)$) và $p_1(u')$ (thuộc đường cong $p_1(u)$) với u' nằm trong ($u_{start}$, $u_{end}$).

Nếu không giới hạn u, v ta sẽ có mặt cong trải dài ra vô tận, các mặt cong "ruled patch" sẽ được tạo bằng cách giới hạn u, v trong đoạn [0, 1].

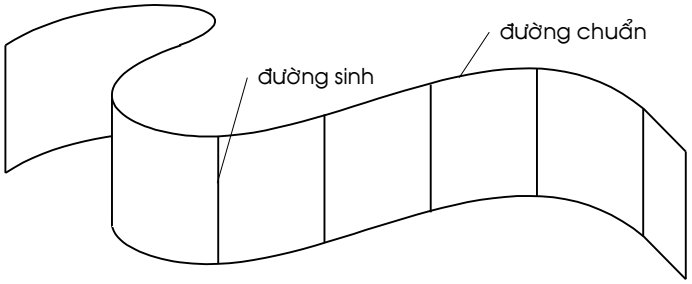
### 10.3.3. Khảo sát các mô hình minh họa

***Hình trụ (Cylinder)***

Hình trụ là hình được tạo ra khi một đường thẳng L, gọi là đường sinh (generator) được quét dọc theo một đường cong $p_0(u)$, gọi là đường chuẩn (directrix), đường cong $p_0(u)$ nằm trên một mặt phẳng nào đó.

**Hình 5.16** – Minh họa một hình trụ

Từ phương trình tổng quát của mặt cong có quy luật :

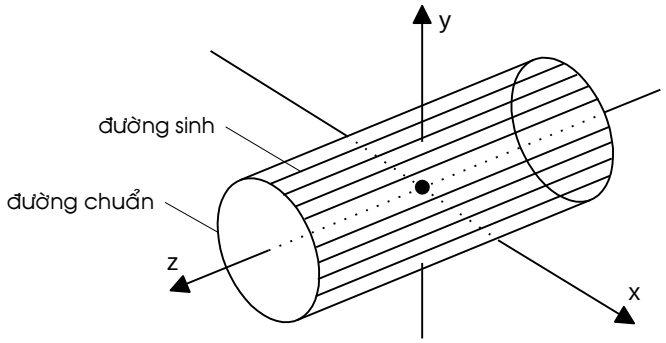



$$p(u,v) = p_0(u) + vd(u), \text{ trong đó } d(u) = p_1(u) - p_0(u) \qquad (5.6)$$

do khi quét các đường thẳng luôn song song với nhau nên ta có d là hằng số, và phương trình tham số của hình trụ là :

$$p(u,v) = p_0(u) + vd$$

Một trong những dạng quen thuộc của hình trụ là hình trụ tròn (circular cylinder) ứng với trường hợp đường chuẩn là hình tròn. Nếu đường tròn nằm trên mặt phẳng xy chúng ta sẽ có $p_0(u) = (\cos(u), \sin(u), 0)$



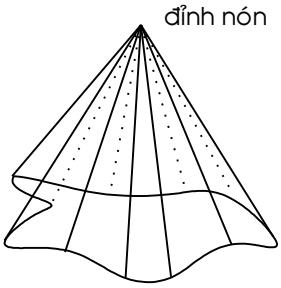

**Hình 5.17** – Minh họa một hình trụ tròn

***Hình nón (Cone)***

Hình nón là hình được tạo ra khi một đường thẳng di chuyển dọc theo một đường cong phẳng cho trước (plane curve), các đường thẳng này còn có thêm tính chất nữa là luôn đi qua một điểm cố định gọi là đỉnh của hình nón.

**Hình 5.18** – Minh họa hình nón

Phương trình tham số của hình nón có dạng tương tự dạng tổng quát nhưng $p_0(u)$ là hằng số :




$$p(u,v) = (1-v)p_0 + vp_1(u) \quad (5.7)$$

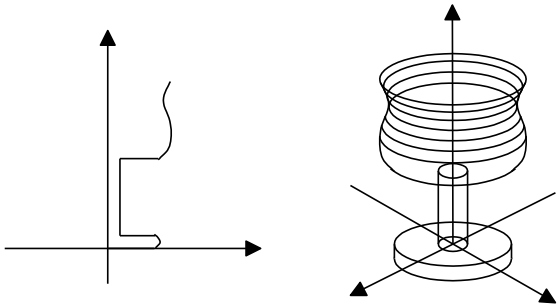
Trong trường hợp này tất cả các đường thẳng sẽ đi qua $p_0$ ứng với v = 0. Đường cong phẳng mà tất cả các đường thẳng đi qua ứng với v = 1.

### 10.4. Các mặt tròn xoay (surfaces of revolution)

Mặt tròn xoay được tạo ra khi chúng ta quay tròn một đường cong phẳng C nào đó quanh một trục. Hình vẽ 5.19 minh họa một đường cong C nằm trong mặt phẳng xz và quay quanh trục z. C thường được gọi là mặt cắt nghiêng và được cho bởi phương trình tham số $c(v) = (x(v), z(v))$ trong đó v biến đổi trong khoảng $(v_{start}, v_{end})$ nào đó.

**Hình 5.19** – Minh họa một mặt tròn xoay

Đối với mặt tròn xoay, mỗi điểm $(x(v), z(v))$ thuộc C được quét xung quanh một trục tọa độ dưới sự kiểm soát của tham số u, u là góc mà mỗi điểm được quay quanh trục. Các vị trí khác nhau của đường cong C quanh trục được gọi là các đường



kinh tuyến (meridians). Khi điểm $(x(v),0,z(v))$ được quay bởi u radian, nó sẽ trở thành $(x(v)\cos(u), x(v)\sin(u), z(v))$. Nếu quay điểm này đủ một vòng quanh trục chúng ta sẽ nhận được một hình tròn. Như vậy, ứng với v là hằng số, đường biên sẽ là các đường tròn và các đường này được gọi là các đường vĩ tuyến của mặt. Kinh tuyến tại v có bán kính là x(v) và nằm trên độ cao z(v) so với mặt phẳng xy, do đó một điểm bất kì trên mặt dạng này sẽ có vector vị trí :

$$p(u,v) = (x(v)\cos(u), x(v)\sin(u), z(v)) \qquad (5.8)$$

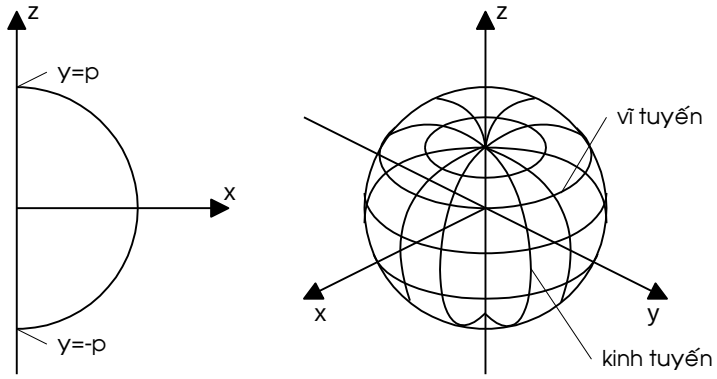
Nhận xét : Nếu đường cong c(v) là đường thẳng song song với trục z và cách z một đơn vị, tức là c(v) = (1, v) thì khi đường này quét quanh trục z sẽ tạo ra một hình trụ.

Mặt cầu là trường hợp đơn giản nhất của dạng mặt tròn xoay. Đường cong C trong trường hợp này chính là nửa đường tròn cho bởi các điểm $(R\cos(v), R\sin(v))$, v chạy trong khoảng từ -π/2 đến π/2. Lúc này phương trình hình cầu sẽ có dạng :

$$p(u,v) = (R\cos(v)\cos(u), R\cos(v)\sin(u), R\sin(v)) \qquad (5.9)$$

trong đó $-\pi/2 \le v \le \pi/2$, $0 \le u \le 2\pi$.

**Hình 5.20** - Minh họa mặt cầu




### 10.5. Các mặt cong bậc hai

Một lớp mặt cong rất thông dụng là các mặt cong bậc hai. Chúng được biểu diễn bởi các phương trình bậc hai. Mặt cầu cũng thuộc lớp mặt cong này. Ngoài ra còn có mặt ellipsoid, paraboloid và hyperboloid. Các mặt bậc hai thường là các đối tượng cơ sở của các hệ đồ họa. Những đối tượng khác phức tạp hơn có thể được tạo ra từ những đối tượng này. Phương trình tổng quát biểu diễn các mặt cong loại này là:

$Ax^2 + By^2 + Cz^2 + Dxy + Eyz + Fzx + Gx + Hy + Iz + J = 0$

#### 10.5.1. Mặt cầu

Trong hệ tọa độ Decartes, mặt cầu bán kính R với tâm đặt tại gốc tọa độ xác định bởi tập các điểm có tọa độ (x,y,z) thỏa phương trình:

$x^2 + y^2 + z^2 = R^2$ (5.10)

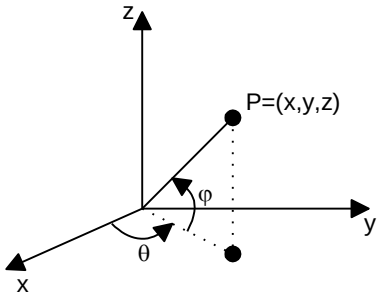
Phương trình (5.10) thường gọi là phương trình chính tắc của mặt cầu.

Như phần trước đã đề cập, ta có thể biểu diễn mặt cầu bằng phương trình tham số:

$x = R\cos\varphi \cos\theta, \quad -\pi/2 \le \varphi \le \pi/2$

$y = R\cos\varphi \sin\theta, \quad -\pi \le \theta \le \pi \qquad (5.11)$

$z = R\sin\varphi$




**Hình 5.21** – Các tham số biểu diễn mặt cầu

### 10.5.2. Ellipsoid

Ellipsoid có thể coi là một mở rộng của mặt cầu với ba bán kính khác nhau $R_x$, $R_y$, $R_z$ (xem hình 5.22). Phương trình chính tắc của một ellipsoid có dạng:

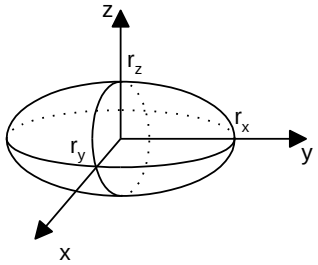
$$\frac{x^2}{R_x^2}+\frac{y^2}{R_y^2}+\frac{z^2}{R_z^2}=1 \quad (5.12)$$

Và phương trình tham số của ellipsoid theo hai góc φ và θ có dạng:

$x = R_x \cos\varphi \cos\theta, \quad -\pi/2 \le \varphi \le \pi/2$

$y = R_y \cos\varphi \sin\theta, \quad -\pi \le \theta \le \pi \qquad (5.13)$

$z = R_z \sin\varphi$




**Hình 5.22** - Ellipsoid với các bán kính $R_x$, $R_y$, $R_z$

## 10.6. VẼ ĐƯỜNG CONG VÀ MẶT CONG BẰNG BEZIER VÀ B-SPLINE

Chúng ta đã khảo sát các đường cong và mặt cong tương đối đơn giản và tìm ra các công thức toán học tương ứng để biểu diễn chúng. Tuy nhiên trong thực tế việc tìm ra các công thức để biểu diễn các đường và mặt phức tạp không đơn giản chút nào. Trong phần này chúng ta sẽ khảo sát các phương pháp cho phép tạo ra các đường cong và mặt cong khác nhau dựa trên dữ liệu mô tả chúng.

Bài toán đặt ra ở đây là : Với một đường cong cho trước mà ta chưa xác định được công thức hay công thức rất phức tạp, và tập nhỏ các điểm phân biệt $p_1$, $p_2$, ... mô tả hình dáng của đường cong này, làm thế nào để xây dựng được đường cong ban đầu với một độ chính xác nào đó.

Có hai cách giải quyết đó là :

- Định tọa độ của một số điểm nào đó thuộc đường cong, sau đó tìm các phương trình toán học và hiệu chỉnh chúng để chúng đi qua hết các điểm trên và trùng khớp với đường cong ban đầu.
- Cách khác là xác định một số các điểm gọi là điểm kiểm soát (control points) và dùng một giải thuật nào đó để xây dựng đường cong dựa trên các điểm này. Do đường cong nguyên thủy và đường cong do máy tính tạo ra thường không đồng nhất ở lần đầu tạo ra, chúng ta sẽ di chuyển một số điểm điều khiển và cho phát sinh lại đường cong mới dựa trên tập các điểm mới tạo. Quá trình này lặp đi lặp lại cho tới khi tìm ra đường cong thỏa mãn phù hợp với đường cong ban đầu thì thôi. Lúc này, đường cong được xây dựng bởi một tập rất ít các điểm điều khiển và có thể được phát sinh lại khi cần.

Trong phần này chúng ta sẽ nghiên cứu theo hướng tiếp cận thứ hai để xây dựng các đường cong và mặt cong đó là xây dựng dựa trên các đường cong Bezier và B-Spline.

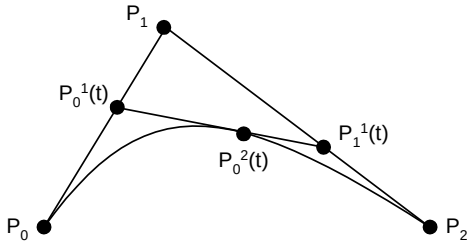
### 10.6.1. Vẽ các đường cong Bezier

***Thuật toán Casteljau***

Thuật toán này dựa trên tập các điểm cho trước để tìm ra các giá trị p(t) khi t thay đổi. Lúc này do đường cong được xây dựng phụ thuộc vào tập các điểm cho trước nên khi thay đổi các điểm này đường cong sẽ thay đổi theo.

Chúng ta bắt đầu quá trình với việc xây dựng đường cong từ ba điểm cho trước $p_0$, $p_1$, $p_2$ như hình vẽ 5.23.

**Hình 5.23** - Thuật toán Casteljau cho ba điểm




Chọn một giá trị t nào đó trong đoạn $[0,1]$, chia đoạn $p_0p_1$ theo tỉ số t được $p_0^1(t)$, chia $p_1p_2$ theo tỉ số t được $p_1^1(t)$. Ta có :

$$p_0^1(t) = (1-t)p_0 + tp_1 \qquad (5.14a)$$

$$p_1^1(t) = (1-t)p_1 + tp_2 \qquad (5.14b)$$

Lặp lại bước nội suy tuyến tính trên với các điểm $p_0^1(t)$ và $p_1^1(t)$ ta được $p_0^2(t)$. Bằng cách này khi cho t chạy trong đoạn [0,1], ta sẽ được đường cong $p(t) = p_0^2(t)$.

Ta có :
$$\begin{aligned} p_0^2(t) &= (1-t)p_0^1(t) + tp_1^1(t) \\ &= (1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2) \\ &= (1-t)^2 p_0 + 2(1-t)tp_1 + t^2 p_2 \end{aligned}$$

Đây là hàm bậc hai theo t nên đường cong sẽ có dạng parabol.

Tổng quát, cho (L+1) điểm $p_0$, $p_1$, .., $p_L$, bằng phương pháp nội suy tương tự, ứng với mỗi t thay đổi trong [0, 1] ta sẽ tìm ra được một giá trị p(t) qua L bước. Trong đó các điểm ở bước thứ r được tạo ra từ các điểm ở bước thứ (r-1) theo phương trình sau :

$$p_i^r(t) = (1-t)p_i^{r-1}(t) + tp_{i+1}^{r-1}(t) \qquad (5.15)$$

với r = 1, .., L; i = 0, .., L-r; và $p_i^0 = p_i$.

Các điểm tạo ra ở bước cuối cùng $p_0^L(t)$ được gọi là đường cong Bezier của các điểm $p_0$, $p_1$, ..., $p_L$. Các điểm $p_0$, $p_1$, ..., $p_L$ được gọi là các điểm kiểm soát (control points) hay điểm Bezier (Bezier points) và đa giác tạo bởi các điểm này được gọi là đa giác kiểm soát (control polygon) hay đa giác Bezier (Bezier polygon).

***Dạng Bernstein của đường cong Bezier***

Công thức đường cong Bezier dựa trên (L+1) điểm $p_0$, $p_1$, ..., $p_L$ có thể được viết lại như sau :

$$P(t) = \sum_{k=0}^{L} P_k B_k^L(t) \qquad (5.16)$$

trong đó $B_k^L$ được gọi là đa thức Bernstein (Bernstein polynomial) được cho bởi công thức sau :

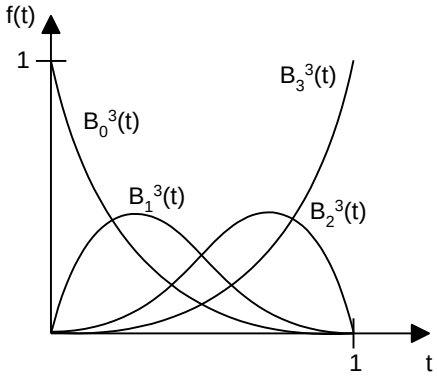
$$B_k^L(t) = \binom{L}{k}(1-t)^{L-k}t^k = C_L^k(1-t)^{L-k}t^k \qquad (5.17)$$

$$C_L^k = \frac{L!}{k!(L-k)!}$$

khi L>=k và bằng 0 cho các trường hợp còn lại.

Dễ dàng nhận thấy đa thức Bernstein $B_k^L$ (t) chính là các thành phần khi khai triển biểu thức $((1-t)+t)^L$, do đó tổng của các $B_k^L(t)$ luôn có giá trị 1với mọi giá trị của t.

$$\sum_{k=0}^{L} B_k^L(t) = 1 \qquad (5.18)$$



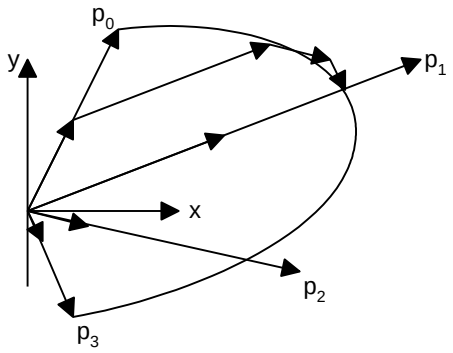

Hình vẽ sau minh họa bốn đa thức Bernstein bậc ba khi t biến đổi trong [0, 1]

**Hình 5.24** – Các đa thức Bernstein bậc ba

Các hàm $B_k^L$ (t) thường được gọi là các hàm trộn (blending functions) vì vector p(t) có thể được xem được "pha trộn" từ các vector $p_0$, $p_1$, ..., $p_L$. Với mỗi giá trị t, mỗi đa thức Bernstein xác định một tỉ lệ hay trọng lượng cho các vector tương ứng. Theo dõi hình vẽ 5.25, ta thấy khi t = 0.3, bốn đa thức tương ứng với $p_0$, $p_1$, $p_2$, $p_3$, $p_4$ cho các giá trị 0.343, 0.441, 0.189, 0.027. Tổng của bốn vector được gia trọng bởi các trọng lượng này chính là vector p(0.3).

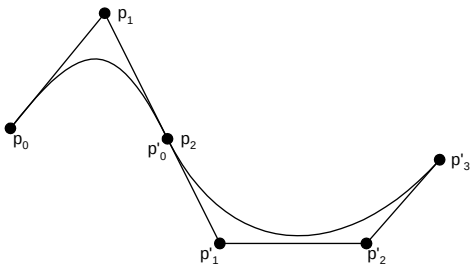
Hàm trộn này là một đa thức có bậc nhỏ hơn số lượng các điểm kiểm soát . Ba điểm sẽ cho một parabol, bốn điểm sẽ cho một đường cong bậc ba.

Thông thường, số lượng các điểm kiểm soát có thể định là tùy ý cho việc xây dựng đường cong Bezier , tuy nhiên điều này đòi hỏi những tính toán phức tạp khi làm việc với các hàm đa thức bậc cao. Chúng ta khắc phục điều này bằng nhận xét : Một đường cong phức tạp bao giờ cũng có thể ghép từ những đoạn khác nhau, do đó trên những đoạn con này chúng ta xây dựng các đường cong Bezier có bậc nhỏ hơn.




**Hình 5.25** – Hàm trộn của các đa thức

Việc tạo các đường cong phức tạp bằng cách ghép nối các đoạn nhỏ hơn cho phép người dùng kiểm soát những thay đổi cục bộ (local variation) của đường cong tốt hơn. Vì đường cong Bezier đi qua hai điểm đầu và cuối, nên rất dễ dàng kết hợp các đoạn cong (liên tục bậc 0). Hơn nữa, đường cong Bezier còn có một tính chất quan trọng nữa đó là tiếp tuyến với đường cong tại một điểm đầu hoặc cuối thì nằm trên đường thẳng nối điểm đó với điểm kiểm soát kế nó. Do đó, để nhận được sự liên tục bậc một giữa các đoạn cong, ta chỉ cần đặt các điểm kiểm soát sao cho các điểm $p_{n-1}$ và $p_n$ của một đoạn cong trước và các điểm $p_0$ và $p_1$ của đoạn cong kế tiếp nằm trên cùng một đường thẳng. Hình vẽ 5.26 minh họa quá trình nhận được sự liên tục bậc 0 và liên tục bậc 1 khi ghép nối hai đoạn cong Bezier bằng cách cho $P'_0 = P_2$ và cho các điểm $P_1$ , $P_2$ và $P'_1$ thẳng hàng. Đối với các đường cong Bezier thường không đòi hỏi tính liên tục bậc hai.




**Hình 5.26 – Ghép nối hai đoạn cong**

**Cài đặt minh họa thuật toán vẽ đường cong Bezier qua (N+1) điểm kiểm soát**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
```

```
#define MAXPOINTS 100 // So diem kiem soat toi da
#define MAXSEG                  100 // So diem toi da thuoc duong cong

typedef struct
{
    int x;
    int y;
}POINT;

// Kieu mang cac diem thuoc duong cong
typedef POINT BEZPOINT[MAXSEG+1];

// Kieu mang cac diem kiem soat
typedef POINT CTRLPOINT[MAXPOINTS+1];

// Kieu mang luu he so C(k, N)
typedef     long  COEFF[MAXPOINTS+1];

// Tinh he so aCoeff[i] = C(k, n) = N!/k!*(N-k)! = (k+1)(k+2)....N)/(N-k)!;
// aCoeff co (N+1) phan tu danh so tu 0 den N
void ComputeCoefficient(COEFF aCoeff, int N)
{
    for(int k=0; k<=N; k++)
    {
        aCoeff[k] = 1;
        for(int j=N; j>k; j--)
            aCoeff[k] *= j;
        for(j=2; j<=N-k; j++)
            aCoeff[k] /= j;
    }
}

// Tinh t^k*(1-t)^N-k
float BlendingFunc(float t, int N, int k)
{
    float fRes;

    fRes = 1;
    for(int i=0; i<k; i++) // Tinh t^k
        fRes *=t;
    for(i=0; i<N-k; i++) // Tinh (1-t)^N-k
        fRes *=(1-t);
    return fRes;
}

/*
```

```
    Phat sinh mot diem thuoc duong cong ung voi mot gia tri t nao do
    aCtrlPt : Mang cac diem kiem soat (N+1)
    N       : So diem kiem soat
    aCoeff  : Mang luu he so C(k, N) da duoc tinh truoc khi goi
    t       : gia tri thuc.
*/

POINT   FindBezPt(CTRLPOINT aCtrlPt, int N, COEFF aCoeff, float t)
{
    POINT   BezPt;
    float   B, x, y;

    x = 0;
    y = 0;

    for(int k=0; k<=N; k++)
    {
        B   = BlendingFunc(t, N, k);
        x += aCtrlPt[k].x*aCoeff[k]*B; // pk*B(k, N)
        y += aCtrlPt[k].y*aCoeff[k]*B;
    }

    BezPt.x = x;
    BezPt.y = y;

    return BezPt;
}

// Phat sinh cac diem thuoc duong cong Bezier voi (N+1) diem kiem soat
void BezCurve(BEZPOINT aBezPt, int NumBezPt, CTRLPOINT aCtrlPt,
                        int N)
{
    COEFF   aCoeff;
    POINT   Pt;

    ComputeCoefficient(aCoeff, N);

    for(int i=0; i<=NumBezPt ; i++)
    {
        aBezPt[i] = FindBezPt(aCtrlPt, N, aCoeff, (1.0*i)/NumBezPt);
    }
}

// Ve da giac kiem soat
void  DrawCtrlPt(CTRLPOINT aCtrlPt, int N)
{
```

```
    for(int i=0; i<N; i++)
        line(aCtrlPt[i].x, aCtrlPt[i].y,  aCtrlPt[i+1].x, aCtrlPt[i+1].y);
}

// Ve duong cong Bezier
void  DrawBezCurve(BEZPOINT aBezPt, int NumBezPt)
{

    for(int i=0; i<NumBezPt; i++)
        line(aBezPt[i].x, aBezPt[i].y,  aBezPt[i+1].x, aBezPt[i+1].y);
}
```

***Các đường cong Bezier bậc ba***

Như đã nhận xét ở trên, độ phức tạp tính toán của các đường cong Bezier tăng nhanh theo bậc của chúng. Trong thực tế, nhiều hệ đồ họa chỉ cung cấp các hàm vẽ đường cong Bezier bậc ba, các đường cong này được phát sinh bởi bốn hàm trộn $B_0^3(t)$ , $B_1^3(t)$ , $B_2^3(t)$ , $B_3^3(t)$ . Ta có công thức tường minh của các đa thức này như sau:

$$\begin{aligned} B_0^3(t) &= (1-t)^3 \\ B_1^3(t) &= 3t(1-t)^2 \\ B_2^3(t) &= 3t^2(1-t) \\ B_3^3(t) &= u^3 \end{aligned} \qquad (5.19)$$

Khai triển các đa thức biểu diễn các hàm trộn trên, ta có thể viết hàm Bezier bậc ba dưới dạng ma trận như sau:

$$p(t) = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} M_{Bez} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} \qquad (5.20)$$

trong đó ma trận Bezier có giá trị:

$$M_{Bez} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} (5.21)$$

Tại hai đầu cuối của đường cong Bezier bậc ba, phương tiếp tuyến (đạo hàm bậc một) có giá trị:

$p'(0) = 3(p_1 - p_0)$,  $p'(1) = 3(p_3 - p_2)$

Đạo hàm bậc hai tại các điểm này lần lượt sẽ là:

$p''(0) = 6(p_0 - 2p_1 + p_2)$,  $p'(1) = 6(p_1 - 2p_2 + p_3)$

Ta có thể dùng các biểu thức trên để tạo ra các đường cong có độ trơn bậc một hoặc bậc hai từ các đoạn cong vẽ bằng hàm Bezier bậc ba.

***Các tính chất của đường cong Bezier***

*- Luôn đi qua điểm đầu và điểm cuối*

Đường cong Bezier dựa trên các điểm kiểm soát $p_0$, $p_1$, .., $p_L$ không hoàn toàn đi qua hay nội suy từ tất cả các điểm kiểm soát nhưng nó luôn luôn đi qua điểm đầu và điểm cuối. Đây là tính chất cực kì thú vị bởi vì nó cho phép chúng ta biết chính xác nơi bắt đầu và kết thúc của đường cong Bezier.

Thật vậy, ta có đa thức Bernstein cho các điểm đầu $p_0$ và cuối $p_L$ lần lượt là $B_0^L(t) = (1-t)^L$ và $B_L^L(t) = t^L$. Do đó, với t=0, ta có : $p(0) = p_0$ và với t =1 thì $p(1) = p_L$.

*- Tính bất biến affine (Affine invariance)*

Khi thực hiện phép biến đổi affine cho một đường cong Bezier ta không cần phải biến đổi hết các điểm thuộc đường cong mà chỉ cần biến đổi các điểm kiểm soát, sau đó tạo lại đường cong Bezier dựa trên tập các điểm kiểm soát mới này. Điều này

có nghĩa là đường cong Bezier bất biến với phép biến đổi affine.

Thật vậy : Xét đường cong

$$q(t) = \sum_{k=0}^{L} (p_k M + tr) B_k^L(t) \quad (5.22)$$

q(t) là đường cong Bezier được tạo ra bởi tập điểm kiểm soát $q_k=(p_kM+tr)$ là ảnh của $p_k$ qua phép biến đổi affine với ma trận biến đổi M và vector tịnh tiến tr. Ta sẽ chứng minh đường cong này cũng chính là ảnh $p'(t)$ : $p'(t) = \left( \sum_{k=0}^{L} p_k B_k^L(t) \right) M + tr$ (5.23) của đường cong Bezier p(t).

Từ (5.15) ta có :

$$q(t) = \sum_{k=0}^{L} p_k M B_k^L(t) + \sum_{k=0}^{L} tr B_k^L(t)$$

$$q(t) = \sum_{k=0}^{L} p_k M B_k^L(t) + tr \sum_{k=0}^{L} B_k^L(t)$$

nhưng tổng $\sum_{k=0}^{L} B_k^L(t) = 1$ nên ta suy ra điều cần chứng minh.

- *Tính chất bao lồi (Convex hull property)*

Đường cong Bezier không bao giờ nằm ngoài bao lồi của nó.

Ta biết bao lồi của một tập các điểm $p_0$, $p_1$, ..., $p_L$ là một đa giác lồi nhỏ nhất chứa tất cả các điểm đó. Nó cũng chính là tập tất cả các tổ hợp lồi :

$\sum_{k=0}^{L} \alpha_k p_k$ trong đó $\alpha_k \geq 0$ và $\sum_{k=0}^{L} \alpha_k = 1$ (5.24)

p(t) chính là tổ hợp lồi của các điểm kiểm soát của nó với mọi giá trị của t vì các giá trị của các đa thức Bernstein không âm và có tổng là 1 nên mọi điểm của đường cong Bezier sẽ luôn nằm trong bao lồi của các điểm kiểm soát.

- *Tính chất chính xác tuyến tính (Linear precision)*

Đường cong Bezier có thể trở thành một đường thẳng khi tất cả các điểm kiểm soát nằm trên một đường thẳng, bởi vì lúc này bao lồi của đường cong Bezier là đường thẳng.

Số giao điểm của một đường thẳng hay mặt phẳng bất kì với đường cong Bezier luôn nhỏ hơn số giao điểm của nó với đa giác kiểm soát.

***Dạng ma trận của đường cong Bezier***

Ta biểu diễn lại tập các đa thức Bernstein và tập các điểm kiểm soát dưới dạng vector như sau :

$$B^L(t) = (B_0^L(t), B_1^L(t), ..., B_L^L(t))$$

$$P = (p_0, p_1, ..., p_L)$$

Lúc này $p(t) = B^L(t).P$

Hay viết dưới dạng nhân ma trận là $p(t) = B^L(t).P^T$. Với $P^T$ là chuyển vị của P.

Ta có thể viết lại đa thức Bernstein dưới dạng sau :

$$B_k^L(t) = a_0 + a_1 t + a_2 t^2 + ... + a_L t^L$$
$$= (t^0, t^1, ..., t^L)(a_0, a_1, ..., a_L)$$

Do đó ta có : $p(t) = Pow^L(t) Bez^L P^T$

Trong đó : $Pow^L(t) = (t^0, t^1, ..., t^L)$.

và $Bez^L$ là ma trận mà mỗi dòng i của nó chính là bộ $(a_0, a_1, ..., a_L)$ của biểu diễn đa thức $B_i^L(t)$. Ta sẽ tính được :

$$Bez^L = (m_{ij}) = (-1)^{j-i} \binom{L}{i} \binom{i}{j} \quad (5.25)$$

### 10.6.2. Đường cong Spline và B-Spline

Với đường cong Bezier, ta có thể tạo ra các dạng đường cong khác nhau bằng cách hiệu chỉnh các điểm kiểm soát cho tới khi có được dạng đường cong thỏa mãn yêu cầu đặt ra ban đầu. Tuy nhiên, việc hiệu chỉnh thật không đơn giản chút nào nếu ta quan sát quá trình được mô tả bằng hình 5.27, trong đó một phần của đường cong Bezier đã đúng và phần còn lại thì cần phải hiệu chỉnh thêm.

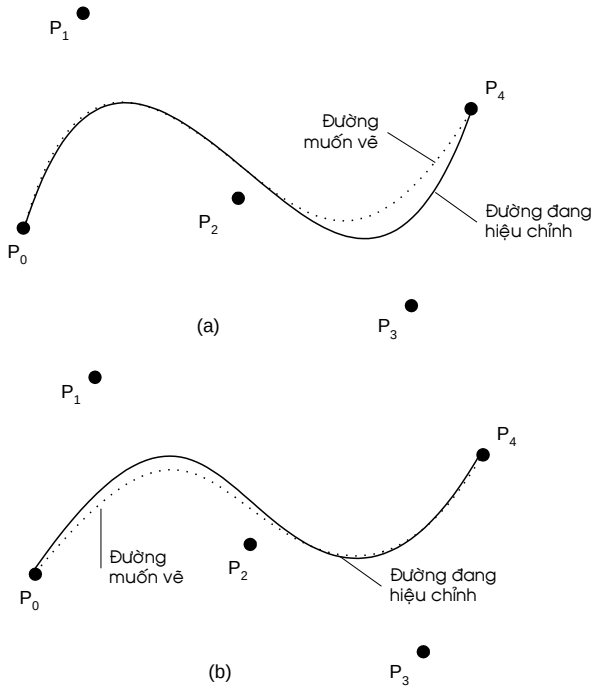
Chúng ta có 5 điểm kiểm soát và đường cong Bezier được tạo ra từ chúng có nét liền so với đường cong mà ta cần phải vẽ có nét gạch đứt quãng. Ta nhận thấy rằng với t gần 0 thì đường cong Bezier có vẻ khớp so với đường cong cần vẽ nhưng lại lệch khi t gần 1. Chúng ta sẽ di chuyển $p_2$, $p_3$ lên một tí để đường cong Bezier khớp với đường cần vẽ, tuy nhiên điều này lại gây ra hiệu ứng làm cho phần đầu của đường cong lệch đi.

**Hình 5.27** – Hiệu chỉnh một đường cong

Như vậy, khó khăn ở đây là do khi ta thay đổi bất kì một điểm kiểm soát nào thì toàn bộ đường cong cũng sẽ bị thay đổi theo. Điều này thật dễ hiểu do tất cả các đa thức Bernstein đều khác 0 trên toàn đoạn [0,1].

Để giải quyết bài toán này ta sẽ sử dụng một tập các hàm trộn khác nhau $R_0(t)$, $R_1(t)$, ... chứ không phải chỉ một hàm $B_k^L(t)$ như trong trường hợp Bezier. Các hàm trộn này có giá mang (đoạn trên đó hàm lấy giá trị khác 0) chỉ là một phần của đoạn [0, 1], ngoài giá mang này chúng có giá trị là 0. Bằng cách này, đường cong chỉ phụ thuộc vào một số điểm kiểm soát mà thôi.

Các hàm trộn mà ta đề cập đến ở đây chính là tập các đa thức được định nghĩa trên các đoạn kề nhau để khi nối lại với




nhau tạo nên một đường cong liên tục. Các đường cong như vậy được gọi là đa thức riêng phần (piecewise polynomials).

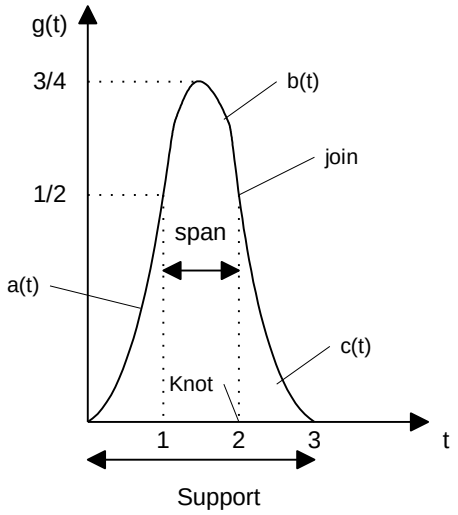
Ví dụ ta định nghĩa hàm g(t) bao gồm ba đa thức a(t), b(t), c(t) như sau :

$$g(t) = \begin{cases} a(t) = \dfrac{1}{2}t^2 \\ b(t) = \dfrac{3}{4} - \left(t - \dfrac{3}{2}\right)^2 \\ c(t) = \dfrac{1}{2}(3-t)^2 \end{cases} \quad (5.26)$$

Giá mang của g(t) là [0, 3], của a(t) là [0, 1], của b(t) là [1, 2], của c(t) là [2, 3].

Các điểm tại các đoạn đường cong gặp nhau được gọi là các điểm nối (joints), và giá trị t tại các điểm đó được gọi là nút (knot).

Có thể kiểm chứng được g(t) liên tục tại mọi nơi trên giá mang của nó, nên đường cong tại các chỗ nối là trơn. g(t) là một ví dụ của hàm Spline.




**Hình 5.28** – Các thành phần của một đa thức riêng phần

***Định nghĩa hàm Spline***

Một hàm Spline cấp M là một đa thức riêng phần cấp M có các đạo hàm cấp (M-1) liên tục ở mỗi nút.

Rõ ràng theo định nghĩa thì g(t) là một Spline bậc hai.

***Định nghĩa đường cong Spline***

Ta xây dựng đường cong p(t) dựa trên (L+1) điểm kiểm soát bằng cách sử dụng các hàm Spline làm các hàm trộn như sau :

- Xây dựng tập các nút $t_0$, $t_1$, .., với $t_i \in \mathbf{R}$ và $t_i \le t_{i+1}$.
- Vector T = ($t_0$, $t_1$, ... ) được gọi là vector nút.
- Với mỗi điểm kiểm soát $p_k$ ta kết hợp nó với một hàm trộn tương ứng là $R_k(t)$. $R_k(t)$ là đa thức riêng phần liên tục trên mỗi đoạn con $[t_i, t_{i+1}]$ và liên tục tại mỗi nút.

Khi đó : $p(t) = \sum_{k=0}^{L} p_k R_k(t)$

Các đoạn đường cong riêng phần này gặp nhau tại các điểm nút và làm cho đường cong liên tục. Ta gọi những đường cong như vậy là đường cong Spline.

Vấn đề được đặt ra tiếp ở đây : Cho trước một vector nút, có tồn tại hay không họ các hàm trộn sao cho chúng có thể phát sinh ra mọi đường cong Spline được định nghĩa trên vector nút đó. Một họ các hàm như vậy được gọi là cơ sở cho Spline, nghĩa là bất kì đường cong Spline nào cũng có thể được đưa về cùng một công thức bằng cách chọn đa giác kiểm soát phù hợp.

Câu trả lời là có nhiều họ hàm như vậy, nhưng đặc biệt có một họ hàm trộn có giá mang nhỏ nhất đó là B-Spline (B là từ viết tắt của basis).

***Định nghĩa đường cong B-Spline***

Một đường cong B-Spline cấp m xây dựng dựa trên vector nút T và (L+1) điểm kiểm soát $p_k$ có dạng :

$$p(t) = \sum_{k=0}^{L} p_k N_{k,m}(t)$$

Trong đó $N_{k,m}(t)$ là đa thức có bậc (m-1) có công thức đệ quy :

$$N_{k.m}(t) = \left( \frac{t - t_k}{t_{k+m-1} - t_k} \right) N_{k,m-1}(t) + \left( \frac{t_{k+m} - t}{t_{k+m} - t_{k+1}} \right) N_{k+1,m-1}(t),$$

k=0, 1, .., L

với $N_{k,1}(t) = \begin{cases} 1 & \text{nếu } t_k \le t < t_{k+1} \\ 0 & \text{nếu } t \text{ còn lại} \end{cases}$

Các điểm $t_i$ có thể được xác định theo nhiều cách khác nhau. Một trong các cách đó là cho $t_i = i$, lúc này khoảng cách giữa các điểm nút là bằng nhau. Hay ta có một cách định nghĩa khác :

$$t_i = \begin{cases} 0, \text{nếu } i < m \\ L - m + 1, \text{nếu } m \le i \le L \\ L - m + 2, \text{nếu } i > L \end{cases}$$

với i = 0, …,L+m.

### 10.6.3. Thiết kế các mặt cong dựa trên Bezier và B-Spline.

Để mô tả và vẽ các mặt cong ta cũng có thể dùng các hàm trộn Bezier và B-Spline tương tự như trong trường hợp đường cong.

***Các mảnh Bezier (Bezier surface patches)***

Xét đường cong Bezier như là một hàm theo tham số v và có các điểm kiểm soát thay đổi theo u. Ta có công thức :

$$p(u,v) = \sum_{k=0}^{L} p_k(u) B_k^L(v)$$

Lúc này, khi u thay đổi ta sẽ có các điểm kiểm soát thay đổi kéo theo đường cong Bezier cũng thay đổi theo. Sự biến thiên của các đường cong Bezier này trong không gian sẽ tạo ra một mặt cong.

Khi u thay đổi, các điểm $p_k(u)$ sẽ thay đổi trên một đường cong nào đó. Nếu cho các đường cong này chính là các đường cong Bezier, mỗi đường cong dựa trên (M+1) điểm kiểm soát thì :

$$p_k(u) = \sum_{i=0}^{M} p_{i,k} B_i^M(u)$$

Lúc này :

$$p(u,v) = \sum_{i=0}^{M} \sum_{k=0}^{L} p_{i,k} B_i^M(u) B_k^L(v)$$

Ta gọi đây là dạng tích tensor của mảnh Bezier.

***Dán các mảnh Bezier lại với nhau***

Mục đích là để tạo ra một dạng mặt cong phức tạp gồm nhiều mảnh Bezier kết hợp lại với nhau sao cho trơn tru tại các biên chung.

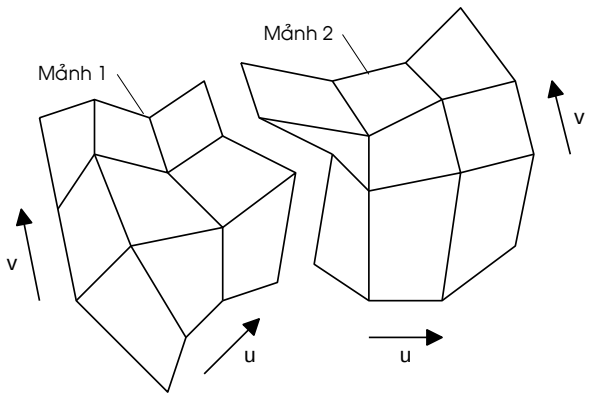
Khi dán hai mảnh Bezier lại với nhau (mỗi mảnh có một khối đa diện kiểm soát riêng và cùng sử dụng công thức ở trên với u,v biến thiên trong đoạn [0, 1]), vấn đề là làm sao để chúng có thể dán vào nhau một cách trơn tru ?

Hai mảnh sẽ gắn vào nhau ở tất cả các điểm dọc biên chung nếu các đa diện kiểm soát của chúng trùng khớp với nhau ở biên. Điều này có được là do dạng của đường cong Bezier biên chỉ phụ thuộc vào đa giác kiểm soát nằm ở biên của khối đa diện kiểm soát. Do đó, để dán được ta chỉ cần chọn các đa giác kiểm soát biên cho hai mặt là trùng nhau.

**Hình 5.29 -** Minh họa hai mảnh Bezier dán lại với nhau

Về tính liên tục tại tiếp tuyến, điều kiện đủ là mỗi cặp cạnh của các khối đa diện tại biên phải là cộng tuyến.

***Các mảnh B-Spline (B-Spline patches)***




Các hàm B-Spline có thể dùng ở dạng tích tensor thay cho dạng đa thức Bernstein để đạt được tính kiểm soát cao hơn khi thiết kế mặt cong :

$$p(u,v) = \sum_{i=0}^{M}\sum_{k=0}^{L} p_{i,k} N_{i,m}(u) N_{k,m}(v)$$

Khối đa diện kiểm soát có (M+1)x(L+1) đỉnh và u, v biến thiên từ 0 tới giá trị lớn nhất của nút trong các vector nút tương ứng của chúng.

Thông thường để thiết kế, người ta vẫn dùng các B-Spline cấp 4 (tức là cubic B-Spline) và do việc chọn số điểm kiểm soát không hạn chế (số lượng các điểm không ảnh hưởng đến bậc của đa thức như đối với đường cong Bezier) nên người ta có thể tạo ra các dạng mặt cong rất phức tạp. Tất nhiên trước đó, người ta phải chọn ra một đa diện nút (knot polyhedron) để tạo ra mặt cong có dạng mong muốn.

## TÓM TẮT

Chúng ta vừa tìm hiểu một trong các mô hình dùng để vẽ các đối tượng ba chiều trên máy tính : đó là mô hình khung nối kết. Theo mô hình này, một đối tượng ba chiều có thể được mô tả bởi tập các đỉnh và tập các cạnh, do đó các đối tượng được thể hiện chưa được gần thực tế lắm, nó mới chỉ là khung rỗng của đối tượng mà thôi. Sau này bằng các kĩ thuật tô màu, khử các đường và mặt khuất chúng ta sẽ khắc phục được các hạn chế này.

Để vẽ các đối tượng ba chiều bằng mô hình khung nối kết, mỗi cạnh phải được chiếu theo một cách nào đó từ tọa độ ba chiều sang hai chiều. Qua đó chúng ta cũng đã tìm hiểu hai phép chiếu khá đơn giản để làm việc này đó là phép chiếu trực giao và phép chiếu phối cảnh. Phép chiếu trực giao chỉ đơn giản là bỏ đi một trong ba tọa độ của điểm chiếu bằng cách cho các tia chiếu song song với một trong các trục tọa độ. Phép chiếu phối cảnh thì sử dụng một điểm cố định gọi là mắt và hình chiếu của các điểm được xác định bằng giao điểm của tia chiếu (nối điểm chiếu và mắt ) với mặt phẳng quan sát. Phép chiếu phối cảnh hội tụ tại mắt nên đối tượng càng xa trông càng nhỏ và ngược lại.

Các phép chiếu trực giao và phối cảnh đều bảo toàn đường thẳng, đây là một tính chất rất hay giúp ta vẽ các đường thẳng ba chiều đơn giản hơn vì chỉ cần xác định hai hình chiếu của hai điểm đầu và cuối mà thôi.

Biểu diễn các mặt trong đồ họa máy tính là một vấn đề luôn được đặt ra khi muốn mô tả các đối tượng lập thể trong thế giới thực. Chúng ta đã khảo sát về các phương pháp biểu diễn mặt phẳng và mặt cong thông qua dạng phương trình tham số. Trong đó, phương trình tham số của một mặt có dạng là một phương trình tham số hai biến p(u, v) và một điểm bất kì trên mặt sẽ được biểu diễn dưới dạng p(u, v) = (x(u, v), y(u, v), z(u, v)). Chúng ta đã khảo sát một số mặt đơn giản như các mặt có quy luật và các mặt tròn xoay để minh họa cho việc xác định các hàm x(), y(), z() trong biểu diễn trên.

Việc tạo ra các đường cong theo ý muốn cũng là vấn đề thường gặp khi làm việc với đồ họa máy tính. Chúng ta đã khảo sát cách tiếp cận vẽ đường cong bằng Bezier và B-Spline. Cách tiếp cận này dựa trên cơ sở để vẽ đường cong bằng một tập điểm mô tả hình dáng của đường cong gọi là tập điểm kiểm soát. Khi thay đổi tập điểm này, hình dáng của đường cong sẽ thay đổi theo. Cách tiếp cận này cho thấy sự thuận lợi và linh hoạt khi cần phải vẽ các đường cong phức tạp và do đó nó được dùng nhiều trong thiết kế.

Một nhược điểm trong cách vẽ đường cong bằng Bezier là khi một phần đường cong đã đạt yêu cầu, nhưng khi hiệu chỉnh phần còn lại sẽ làm mất đi phần đã đạt yêu cầu. Để khắc phục vấn đề này ta có cách tiếp cận cải tiến vẽ đường cong bằng B-Spline.

Trên cơ sở của việc vẽ các đường cong bằng Bezier và B-Spline chúng ta cũng có thể xây dựng được các mặt cong Bezier và B-Spline.

## BÀI TẬP

**56.** Viết chương trình cho phép người dùng định nghĩa một vật thể ba chiều bằng mô hình khung nối kết. Vẽ vật thể trên dùng lần lượt phép chiếu trực giao và phép chiếu phối cảnh.

**57.** Viết chương trình vẽ các mặt đơn giản đã được khảo sát như hình trụ, hình nón, hình cầu, ….

**58.** Nhận xét cách tiếp cận vẽ đường trong thuật toán Casteljau khác với cách tiếp cận vẽ các đối tượng đồ họa cơ sở ở chương trước như thế nào ?

**59.** Cài đặt thuật toán vẽ đường cong bằng Bezier cho phép người dùng định nghĩa tập điểm kiểm soát mô tả hình dạng đường cong và cho phép người dùng hiệu chỉnh một số điểm kiểm soát để hiệu chỉnh đường cong theo ý muốn.

**60.** Viết chương trình vẽ đường Spline

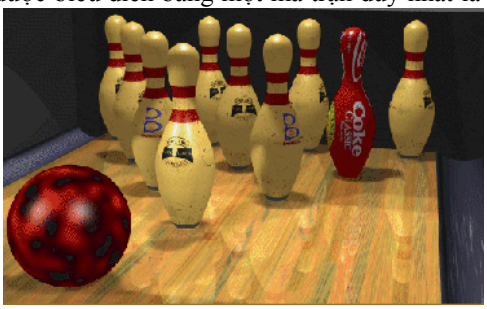
**61.** Viết chương trình vẽ các mặt Bezier.

***CHƯƠNG 6***

# CÁC PHÉP BIẾN ĐỔI TRONG ĐỒ HỌA BA CHIỀU

Các phép biến đổi trong đồ họa ba chiều là sự mở rộng của các phép biến đổi trong đồ họa hai chiều bằng cách thêm vào việc xem xét tọa độ thứ ba, tọa độ z. Bây giờ, chúng ta sẽ tịnh tiến một đối tượng thông qua việc mô tả một vector tịnh tiến ba chiều. Vector này xác định độ dời của vật theo ba chiều trong không gian. Tương tự như vậy, ta có thể thu phóng đối tượng với các tỉ lệ biến đổi theo cả ba chiều. Sự mở rộng của phép quay ít hiển nhiên hơn hai phép biến đổi cơ sở trên. Khi khảo sát các phép quay trong mặt phẳng hai chiều Oxy, ta chỉ cần khảo sát phép quay quanh một tâm, hay nói cách khác, phép quay quanh một trục vuông góc với mặt phẳng Oxy. Trong không gian ba chiều, ta có thể chọn một trục quay có phương bất kì. Phần lớn các hệ đồ họa xử lí phép quay trong không gian ba chiều như là tổ hợp của ba phép quay với trục quay là các trục tọa độ x, y và z. Như vậy, người dùng có thể dễ dàng xây dựng một phép quay bất kì bằng cách mô tả trục quay và góc quay.

Cũng như khi trình bày các phép biến đổi trong đồ họa hai chiều, trong chương này, ta sẽ khảo sát các phép biến đổi trong đồ họa ba chiều dưới dạng ma trận. Một chuỗi bất kì các phép biến đổi sẽ được biểu diễn bằng một ma trận duy nhất là tích của các ma trận tương ứng với các phép biến đổi thành phần.




**Hình 6.1** – Một cảnh ba chiều được tạo nhờ các phép biến đổi

## 11. CÁC PHÉP BIẾN ĐỔI HÌNH HỌC

Phép tịnh tiến, quay, biến đổi tỉ lệ, và phép biến dạng là các ví dụ của các phép biến đổi hình học. Chúng còn được biết tới như là các phép biến đổi affine cơ sở. Trong số đó, phép quay có thể nói là quan trọng và hữu dụng nhất vì nó cho phép chúng ta nhìn các đối tượng theo các hướng khác nhau, điều này cho phép chúng ta cảm nhận các hình vẽ ba chiều trực quan hơn, dễ chịu hơn.

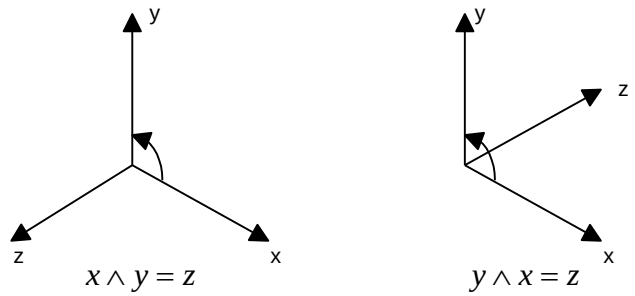
Ta có thể tạo ra nhiều phiên bản của cùng một đối tượng bằng cách vẽ đối tượng này sau khi áp dụng một dãy các phép biến đổi hình học lên nó (xem hình 6.1).

### 11.1. Một số khái niệm liên quan

#### 11.1.1. Phép biến đổi affine

Phép biến đổi affine là phép biến đổi tuyến tính, khả nghịch. Phép biến đổi này bảo toàn tính song song của các đường thẳng cũng như bảo toàn tính tỉ lệ về khoảng cách của các đoạn thẳng. Tuy nhiên, phép biến đổi này không bảo toàn góc nghiêng và chiều dài các đoạn thẳng. Các phép biến đổi này cũng bảo toàn tỉ lệ về khoảng cách (xem thêm chương 3)

Các hệ trục tọa độ theo quy ước bàn tay phải và bàn tay trái




**Hình 6.2** – Các hệ tọa độ theo quy ước bàn tay phải (a)
và quy ước bàn tay trái (b)

Hệ tọa độ Descartes ba chiều có thể định nghĩa theo quy ước bàn tay trái hoặc bàn tay phải (xem hình 6.2).

***Định nghĩa***

- Hệ tọa độ theo quy ước bàn tay phải là hệ tọa độ với các trục x, y, z thỏa điều kiện: Nếu để bàn tay phải sao cho ngón cái hướng cùng chiều với trục z, khi nắm tay lại, chiều các ngón tay chuyển động theo hướng từ trục x đến trục y.
- Hệ tọa độ theo quy ước bàn tay trái là hệ tọa độ với các trục x, y, z thỏa điều kiện: Nếu để bàn tay trái sao cho ngón cái hướng cùng chiều với trục z, khi nắm tay lại, chiều các ngón tay chuyển động theo hướng từ trục x đến trục y.

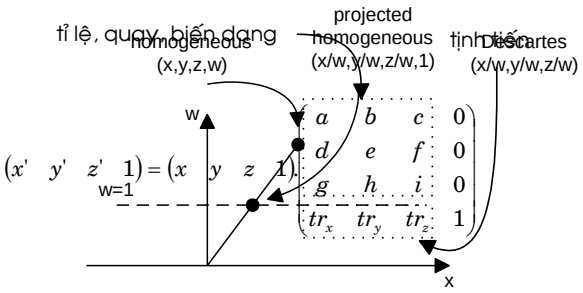
***Hệ tọa độ thuần nhất***

Trong hệ tọa độ thuần nhất, mỗi điểm $(x, y, z)$ trong không gian Descartes được biểu diễn bởi một bộ bốn tọa độ trong không gian 4 chiều thu gọn $(hx, hy, hz, h)$. Để tiện lợi, người ta thường chọn h=1. Như vậy, một điểm (x, y, z) trong hệ tọa độ Descartes sẽ biến thành điểm (x, y, z, 1) trong hệ tọa độ thuần nhất; còn điểm (x, y, z, w) trong hệ tọa độ thuần nhất (với w ≠ 0) sẽ tương ứng với điểm (x/w, y/w, z/w) trong hệ tọa độ Descartes (xem hình 6.3).

**Hình 6.3** – Các điểm trong hệ tọa độ thuần nhất và Descartes

***Dạng ma trận của phép biến đổi affine trong hệ tọa độ thuần nhất***

**Hình 6.4** – Dạng tổng quát của phép biến đổi affine ba chiều

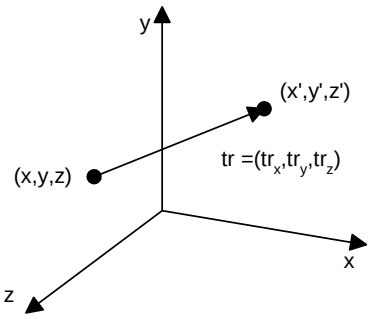



Phép biến đổi affine ba chiều biến điểm P thành điểm Q có dạng : $Q = P.M$, trong đó $Q = (Q_x, Q_y, Q_z, 1)$, $P = (P_x, P_y, P_z, 1)$ và M là ma trận biến đổi 4x4 trong hệ tọa độ thuần nhất $tr = (tr_x, tr_y, tr_z)$ là vector tịnh tiến.

Một số tính chất của các phép biến đổi ba chiều :

- Tính chất đường thẳng được bảo toàn. Nghĩa là, một đường thẳng trong không gian ba chiều khi biến đổi sẽ thành một đường thẳng.
- Tính song song được bảo toàn. Nghĩa là, hai đường thẳng song song khi biến đổi cũng sẽ thành hai đường thẳng song song.
- Tính tỉ lệ về khoảng cách được bảo toàn. Nghĩa là, ảnh của một điểm P chia đoạn thẳng AB theo tỉ lệ f, sẽ chia đoạn thẳng A'B' theo tỉ lệ f, với A'B' là ảnh của đoạn thẳng AB.

## 11.2. Phép tịnh tiến




**Hình 6.5** – Phép tịnh tiến với vector tịnh tiến tr=($tr_x$, $tr_y$, $tr_z$)

Vector tịnh tiến tr trong phép biến đổi ba chiều có một tác động rất trực quan: mỗi điểm được dịch đi một khoảng là $tr_x$, $tr_y$, $tr_z$ theo ba trục. Ma trận M cho phép tịnh tiến có dạng như sau:

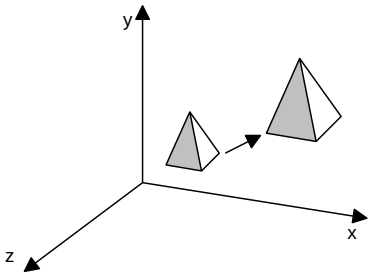
$$tr(tr_x, tr_y, tr_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tr_x & tr_y & tr_z & 1 \end{pmatrix} \quad (6.1)$$

## 11.3. Phép biến đổi tỉ lệ

Phép biến đổi tỉ lệ trong ba chiều là một sự mở rộng của phép biến đổi tỉ lệ trong hai chiều :

$$S(s_x,s_y,s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (6.2)$$

Trong đó các hằng số $s_x, s_y, s_z$ là các hệ số tỉ lệ tương ứng theo các trục x, y, z.




**Hình 6.6** – Phép biến đổi tỉ lệ

Như hình 6.6, đối tượng được phóng to gấp đôi, đồng thời với tác động của phép biến đổi làm cho đối tượng bị đẩy ra xa gốc tọa độ hơn.

Khi các hệ số tỉ lệ $s_x, s_y, s_z$ bằng nhau, ta có phép biến đổi đồng dạng.

Trong phép biến đổi $S(s_x,s_y,s_z)$, gốc tọa độ O sẽ có ảnh là chính nó. Ta gọi gốc tọa độ là điểm bất động (fixed point) của S, hay còn gọi O là tâm của phép biến đổi.

Tổng quát hơn, ta có thể mô tả một phép biến đổi tỉ lệ theo một tâm $(x_f, y_f, z_f)$ bất kì bằng một dãy ba phép biến đổi sau:

- Tịnh tiến điểm bất động $(x_f, y_f, z_f)$ về gốc tọa độ.
- Thực hiện phép biến đổi tỉ lệ theo công thức (6.2).
- Tịnh tiến ngược điểm bất động từ gốc tọa độ trở về vị trí ban đầu.

Như vậy, kết hợp ba bước biến đổi trên ta có được ma trận biến đổi của phép biến đổi tỉ lệ theo tâm $(x_f, y_f, z_f)$ với hệ số tỉ lệ $s_x, s_y, s_z$ là:

$$S_f(s_x,s_y,s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ (1\text{-}s_x)x_f & (1\text{-}s_y)y_f & (1\text{-}s_z)z_f & 1 \end{pmatrix} \qquad (6.3)$$

### 11.4. Phép biến dạng

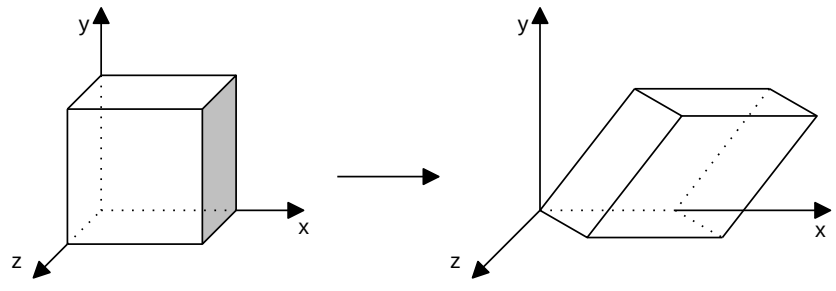
Biến dạng theo bất kì trục tọa độ nào cũng bị ảnh hưởng bởi tọa độ ứng với hai trục còn lại. Ma trận của phép biến dạng như sau:

$$Sh = \begin{pmatrix} 1 & h_{yx} & h_{zx} & 0 \\ h_{xy} & 1 & h_{zy} & 0 \\ h_{xz} & h_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (6.4)$$

Ta có mối quan hệ $Q_x$ với P : $Q_x = P_x + h_{xy}P_y + h_{xz}P_z$.

Ở đây có thể hiểu $h_{xy}$ là lượng mà tọa độ y của P tác động lên tọa độ x của Q.

**Hình 6.7** - Phép biến dạng theo trục x : $h_{xy}= h_{xz} = 1$, các hệ số khác bằng 0



Tương tự như trong trường hợp phép biến đổi tỉ lệ, phép biến dạng Sh (6.4) cũng có điểm bất động là gốc tọa độ O. Ta cũng có thể xây dựng phép biến dạng với tâm biến dạng tại một điểm $(x_f, y_f, z_f)$ bất kì.

Ma trận biến đổi của phép biến dạng với tâm tại $(x_f, y_f, z_f)$ là:

$$Sh = \begin{pmatrix} 1 & h_{yx} & h_{zx} & 0 \\ h_{xy} & 1 & h_{zy} & 0 \\ h_{xz} & h_{yz} & 1 & 0 \\ -y_f h_{xy} - z_f h_{xz} & -x_f h_{yx} - z_f h_{yz} & -x_f h_{zx} - y_f h_{zy} & 1 \end{pmatrix} \quad (6.5)$$

## 11.5. Phép quay

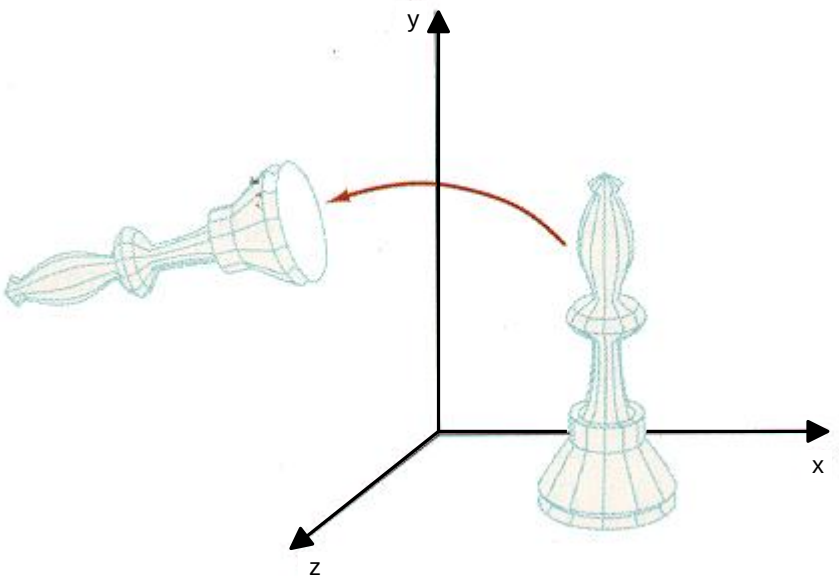
### 11.5.1. Phép quay quanh một trục tọa độ

Khác với phép quay trong hai chiều quanh một điểm bất kì, trong ba chiều ta có phép quay quanh một trục tọa độ. Ở đây ta sử dụng hệ tọa độ theo quy ước bàn tay phải và quy định chiều quay dương là ngược chiều kim đồng hồ.

Ta có các ma trận biểu diễn các phép quay quanh trục x, y, z một góc θ lần lượt là R(z, θ), R(y,θ), R(x, θ):

Quay quanh trục z :

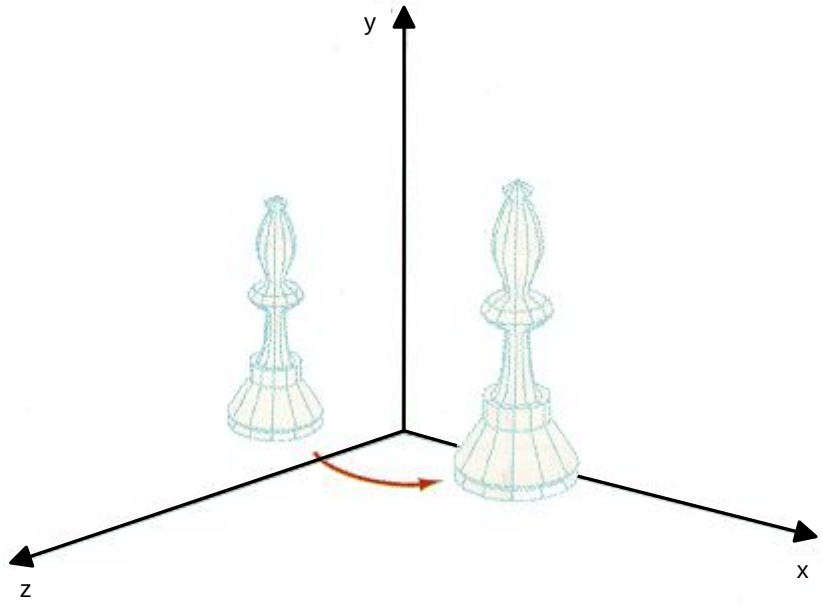
$$R(z,\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (6.6)$$




**Hình 6.8** - Phép quay quanh trục z

Quay quanh trục y :

$$R(y,\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (6.7)$$

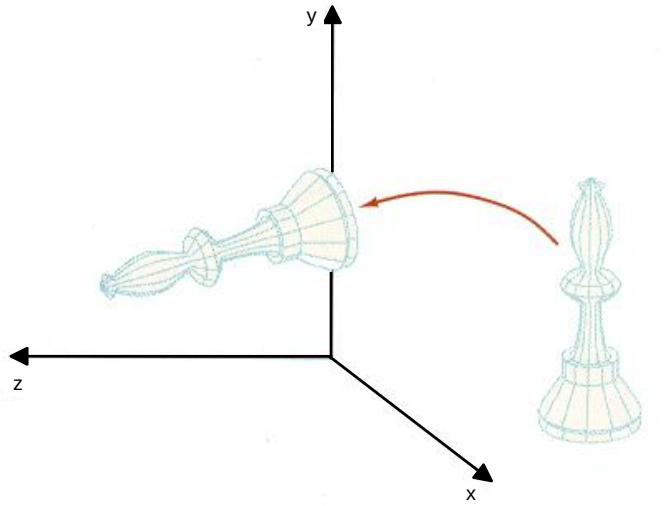



**Hình 6.9** - Phép quay quanh trục y

Quay quanh trục x :

$$R(x,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (6.8)$$

Nhận xét rằng các giá trị nằm trên dòng và cột tương ứng với trục x trong ma trận R(x,θ) sẽ có giá trị là 0 ngoại trừ giá trị nằm trên đường chéo chính là 1. Điều này đảm bảo cho tọa độ x của các điểm là không bị thay đổi qua phép biến đổi. Nhận xét này cũng tương tự cho trường hợp các ma trận còn lại.




**Hình 6.10** - Phép quay quanh trục x

***Ghi chú:***

Các định nghĩa về chiều quay được dùng chung cho cả hệ tọa độ theo quy ước bàn tay phải và bàn tay trái. Cụ thể chiều dương được định nghĩa như sau:

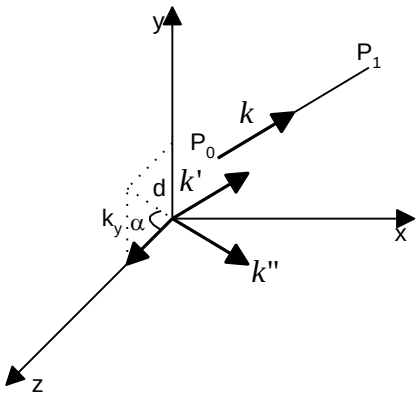
- *Quay quanh trục x: từ trục dương y đến trục dương z.*
- *Quay quanh trục y: từ trục dương z đến trục dương x.*
- *Quay quanh trục z: từ trục dương x đến trục dương y.*

**Hình 6.11**- Cách xác định chiều quay dương

### 11.5.2. Phép quay quanh một trục bất kì

Giả sử trục quay đi qua hai điểm $P_0$, $P_1$ nào đó với phương được biểu diễn bởi vector đơn vị **k.** Quay điểm (x, y, z) quanh trục **k** theo một góc θ sẽ biến thành điểm (x', y', z') *(xem hình 6.12).*

**Hình 6.12** - Phép quay quanh trục **k**




Để thực hiện phép quay quanh **k** một góc θ, ta có thể thực hiện một chuỗi các thao tác sau:

- Tịnh tiến trục **k** về gốc tọa độ: tr(-$P_0$) (thành trục **k'**).
- Quay quanh trục **x** một góc α để đặt trục **k'** nằm trên mặt phẳng Oxz: rot(x,α) (thành trục **k"**).
- Quay quanh trục **y** góc β để đưa trục **k"** về trục **z**: rot(y,-β).
- Thực hiện phép quay quanh trục **z** một góc θ: rot(z,θ).
- Thực hiện chuỗi các phép biến đổi ngược lại quá trình trên.

Góc quay α được xác định dựa trên chiếu của **k'** lên mặt phẳng yz. Ta không cần tính α cụ thể. Thay vào đó ta tính sin(α) và cos(α) một cách trực tiếp.

Từ hình 6.12 ta có:

$$\mathrm{k} = \frac{\overrightarrow{P_0P_1}}{\|P_0P_1\|},\ \mathrm{d} = \sqrt{k_y^2 + k_z^2}$$

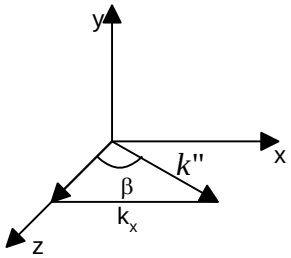
cos(α) = $k_z$/d, sin(α) = $k_y$/d.

Tương tự, từ hình 6.13 ta có:

$$\cos(\beta) = d/1 = d, \quad \sin(\beta) = k_x/1 = k_x$$

Như vậy, phép quay quanh một trục $P_0P_1$ bất kì một góc θ, rot($P_0P_1$, θ), có thể được phân rã thành chuỗi các biến đổi cơ sở sau: tr(-$P_0$) rot(x,α) rot(y, -β) rot(z, θ) rot(y, β) rot(x, -α) tr($P_0$)

**Hình 6.13** - Tính góc β




## 11.6. Kết hợp các phép biến đổi affine ba chiều

Để kết hợp các phép biến đổi affine ba chiều, chúng ta cũng thực hiện tương tự như ở phép biến đổi hai chiều bằng cách dùng hệ tọa độ thuần nhất.

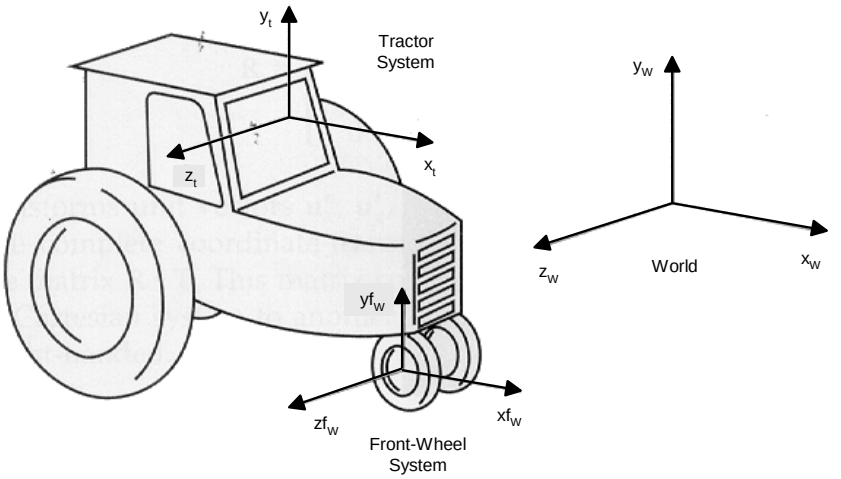
Bằng cách này chúng ta cũng có các kết quả tương tự như trong trường hợp biến đổi hai chiều. Nếu M1 biến đổi P thành Q và M2 biến đổi Q thành S thì M1M2 sẽ biến đổi P thành S, do đó ma trận kết hợp của nhiều phép biến đổi có thể được tính từ việc nhân các ma trận của các phép biến đổi thành phần.

## 12. PHÉP BIẾN ĐỔI MÔ HÌNH VÀ PHÉP BIẾN ĐỔI HỆ TRỤC TỌA ĐỘ

Cho đến thời điểm này, chúng ta đã khảo sát các phép biến đổi ba chiều như là thao tác dịch chuyển một điểm (một đối tượng) từ vị trí này sang vị trí khác trong một hệ trục tọa độ. Tuy nhiên, nhiều khi, ta cần xem xét các đối tượng trong các hệ tọa độ khác nhau, muốn chuyển từ một hệ tọa độ này sang hệ tọa độ khác. Ví dụ, trong quy trình hiển thị đối tượng ba chiều, ta cần đặt một đối tượng vào hệ tọa độ chung cho tất cả các đối tượng trong cảnh (hệ tọa độ thế giới thực), sau đó, xác định tia nhìn, ta chuyển đổi từ hệ tọa độ thế giới thực sang hệ tọa độ quan sát, và cuối cùng ta phải chuyển từ hệ tọa độ quan sát sang hệ tọa độ thiết bị, nơi các đối tượng sẽ được hiển thị.

Khi mô hình hóa đối tượng, ta thường mô tả chúng trong một hệ tọa độ cục bộ, thuận tiện nhất cho việc mô hình hóa. Sau đó, bằng các phép biến đổi ta sẽ đặt chúng vào cảnh cần hiển thị. Cách tiếp cận này cho phép ta không cần mô hình hóa quá nhiều đối tượng mà chỉ mô hình hóa theo chủng loại đối tượng. Ví dụ để tạo cảnh trong hình 6.1 ta chỉ cần mô hình hóa một trái banh, một con ki, bàn, … Sau đó phát sinh ra nhiều con ki như thấy trong hình vẽ. Một ví dụ khác có thể xem trong hình 6.14.

Việc chuyển đổi các mô tả đối tượng từ hệ tọa độ này sang hệ tọa độ khác thực hiện theo quy trình tương tự như trong đồ họa hai chiều. Ta cần xây dựng ma trận biến đổi để khớp được các trục tọa độ của hai hệ. Trước tiên, ta cần thực hiện phép tịnh tiến để hai gốc tọa độ trùng nhau. Sau đó, ta phải thực hiện tiếp một dãy các phép quay để khớp các trục tọa độ tương ứng lên



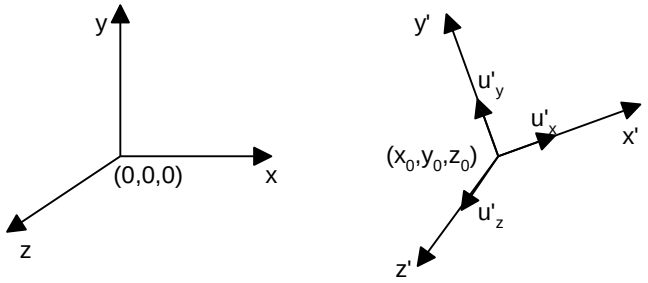

nhau. Nếu các hệ tọa độ sử dụng các tỉ lệ đo lường khác nhau, ta phải thực hiện thêm một phép biến đổi tỉ lệ nữa để đồng nhất các hệ tọa độ.

**Hình 6.14 -** Mô hình hóa và phép biến đổi hệ tọa độ

Nếu hệ tọa độ thứ hai có gốc tọa độ đặt tại $(x_0, y_0, z_0)$ và các vector cơ sở được mô tả như trong hình 6.15 (tương ứng hệ tọa độ thứ nhất), trước tiên ta cần thực hiện phép tịnh tiến $T(-x_0,-y_0,-z_0)$. Sau đó ta xây dựng ma trận quay R dựa trên các vector cơ sở. Ma trận này sẽ biến đổi các vector đơn vị $\mathbf{u}'_x$, $\mathbf{u}'_y$, $\mathbf{u}'_z$ tương ứng thành các trục x, y, z.

$$R = \begin{pmatrix} u'_{x1} & u'_{y1} & u'_{z1} & 0 \\ u'_{x2} & u'_{y2} & u'_{z2} & 0 \\ u'_{x3} & u'_{y3} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (6.9)$$

Ma trận của phép biến đổi hệ tọa độ chính là tích **T.R**. Ma trận này biến đổi hệ tọa độ Descartes này thành hệ tọa độ Descartes khác, cho dù chúng là hệ tọa độ theo quy ước bàn tay phải hay bàn tay trái.




**Hình 6.15 -** Chuyển đổi hệ tọa độ

### TÓM TẮT

Chúng ta vừa khảo sát các phép biến đổi affine ba chiều như là sự mở rộng của các phép biến đổi affine hai chiều. Cũng như các phép biến đổi affine hai chiều, trước tiên ta khảo sát các phép biến đổi cơ sở: tịnh tiến, tỉ lệ, quay; sau đó khảo sát các phép biến đổi phức tạp hơn. Đặc biệt, phép quay quanh một

trục bất kì được khảo sát chi tiết như là một minh họa cho các phân rã một phép biến đổi affine bất kì thành tích của các phép biến đổi affine cơ sở. Nhờ khảo sát các phép biến đổi affine với biểu diễn dạng ma trận trong hệ tọa độ thuần nhất nên công việc khá đơn giản và nhất quán.

Lưu ý một điều, các phép tịnh tiến và quay có chung thuộc tính là : sau khi biến đổi, hình dạng và kích thước của đối tượng không thay đổi mà chúng chỉ bị thay đổi vị trí và định hướng trong không gian. Vì vậy, người ta gọi hai phép biến đổi này là phép biến đổi rigid-body transformations.

Phần cuối chương, chúng ta đã xem xét các phép biến đổi hệ tọa độ. Các phép biến đổi này rất quan trọng trong quá trình hiển thị đối tượng ba chiều.

## BÀI TẬP

**62.** Hãy xác định ma trận của phép đối xứng gương qua mặt phẳng Oxy, Oxz, Oyz.

**63.** Hãy xác định ma trận biến đổi của phép đối xứng gương qua mặt phẳng đi qua điểm P(x,y,z) và có vector pháp tuyến $n$.

**64.** Hãy xác định ma trận của phép đối xứng qua trục tọa độ x, y, z.

**65.** Hãy xác định ma trận của phép đối xứng qua trục bất kì đi qua hai điểm $P_0$, $P_1$.

**66.** Hãy xác định ma trận biến đổi của phép đối xứng qua gốc tọa độ O.

**67.** Hãy xác định ma trận biến đổi của phép đối xứng qua một điểm P(x,y,z) bất kì.

**68.** Cài đặt chương trình cho phép người dùng sử dụng một trong các phép biến đổi affine (tịnh tiến, tỉ lệ, quay, biến dạng, …) để biến đổi đối tượng, sau đó hãy phục hồi lại đối tượng gốc.

**69.** Cài đặt chương trình minh họa sự kết hợp các phép biến đổi cơ sở để tạo thành phép quay quanh một trục bất kì.

**70.** Chứng minh công thức (6.3), (6.5).

**71.** Xây dựng ma trận biến đổi của phép quay quanh một trục bất kì.

***PHỤ LỤC***

# HƯỚNG DẪN SỬ DỤNG THƯ VIỆN ĐỒ HỌA TRONG BC

## 13. MỘT SỐ LƯU Ý CHUNG

- Yêu cầu tối thiểu phải có tập tin điều khiển màn hình **EGAVGA.BGI** (Thông thường các tệp này thường nằm trong thư mục **\BC\BGI** khi được cài đặt).
- Để dùng được thư viện các hàm đồ họa cần có dòng lệnh : **#include <graphics.h>** và đặt mục chọn **Graphics library** là **ON ([x])** trong menu **Options/Linker/Libraries.**
- Đặt **Model** là **Large** trong mục chọn **Options/Linker/Code generation…**
- Không nên dùng các hàm liên quan đến màn hình trong chế độ văn bản như **printf, scanf, …** trong khi dùng chế độ đồ họa.
- Khi cần tham khảo cú pháp, cách sử dụng của bất kì một hàm đồ họa nào, đưa con trỏ về tên hàm trong chương trình sau đó nhấn tổ hợp phím **CTRL+F1**. Muốn tham khảo danh sách toàn bộ các hàm của thư viện đồ họa nhấn tổ hợp phím **CTRL+F1** ngay tại dòng chữ **graphics.h**

## 14. KHỞI TẠO VÀ ĐÓNG CHẾ ĐỘ ĐỒ HỌA

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **initgraph**<br>**closegraph** | Khởi tạo và đóng chế độ đồ họa. Thông thường việc khởi tạo đồ họa được viết dưới dạng một hàm. |
| **graphdefaults** | Đặt các thiết lập đồ họa về giá trị mặc định bao gồm vị trí con trỏ, bảng màu **(palette color),** màu nền, màu vẽ, mẫu tô **(fill pattern),** kiểu tô **(fill style),** … |
| **graphresult** | Trả về mã lỗi của thao tác đồ họa sau cùng không thực hiện được. |
| **grapherrormsg** | Trả về con trỏ tới chuỗi thông báo tương ứng với một mã lỗi cho trước. (Dùng hàm này sau khi có được mã lỗi để biết được tại sao thao tác đồ họa thực hiện không thành công) |
| **detectgraph** | Xác định các trình điều khiển thiết bị và chế độ đồ họa để dùng**.** |
| **Tham khảo** | **getgraphmode, getdrivername, getmodename, getmaxmode, getmoderange, setgraphmode, restorecrtmode, registerbgidriver, registerbgifont, installuserdriver, setgraphbufsize….** |

**Xây dựng hàm khởi tạo chế độ đồ họa**

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
void InitGraph(void)
{
    int gdriver = DETECT, gmode, errorcode;
    /* Goi ham khoi tao che do do hoa voi duong dan den tap tin .BGI la thu muc hien hanh */
    initgraph(&gdriver, &gmode, "");

    /* Lay ket qua cua thao tac khoi tao */
    errorcode = graphresult();
    if (errorcode != grOk)  // Thao tac khoi tao gap loi
    {
```

```
            // In thong bao loi ra man hinh
            printf("Graphics error: %s\n", grapherrormsg(errorcode));
            printf("Press any key to halt:");
            getch();
            exit(1); //Ngung han chuong trinh
        }
    }
```

**Cấu trúc một chương trình đồ họa thông thường**

```
void main (void)
{
    …
    InitGraph(); // Khoi tao che do do hoa
    // Cac thao tac do hoa tiep theo sau day
    …
    closegraph(); // Ngung lam viec tren che do do hoa
    …
}
```

**Chương trình minh họa**

*Mô tả CT : Phát sinh ngẫu nhiên các đường thẳng với màu tùy ý trên màn hình. Chương trình kết thúc khi người dùng nhấn phím bất kì.*

```
// Khai bao cac tap tin INCLUDE
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Khai bao cac bien toan cuc
int MaxX, MaxY; // Chieu rong va chieu cao toi da cua man hinh
int MaxColors; // So mau toi da

// Khai bao cac ham dung trong chuong trinh
// Ham khoi tao che do do hoa
void InitGraph(void)
{
    int gdriver = DETECT, gmode, errorcode;

    /* Goi ham khoi tao che do do hoa voi duong dan den tap tin .BGI la thu muc hien hanh */
    initgraph(&gdriver, &gmode, "");
    /* Lay ket qua cua thao tac khoi tao */
    errorcode = graphresult();
    if (errorcode != grOk)  // Thao tac khoi tao gap loi
    {
        printf("Graphics error: %s\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
```

```
            getch();
            exit(1); //Ngung han chuong trinh
        }
        // Lay cac thong tin ve chieu rong va chieu cao toi da cua che do do hoa dang dung
        MaxX = getmaxx();
        MaxY = getmaxy();
        // Lay thong tin ve so mau toi da
        MaxColors = getmaxcolor() + 1;
}

// Ham phat sinh ngau nhien cac duong thang voi mau tuy y
void LineDemo(void)
{
        int x1, y1, x2, y2; // Toa do diem dau va diem cuoi
        int color; // Mau ve duong thang
        cleardevice(); // xoa man hinh
        do
        {
            // Phat sinh ngau nhien toa do duong thang
            x1 = random(MaxX);
            y1 = random(MaxY);
            x2 = random(MaxX);
            y2 = random(MaxY);
            // Phat sinh ngau nhien mau ve duong thang
            color = random(MaxColors);
            // Goi ham dat mau ve duong thang
            setcolor(color);
            // Goi ham do hoa thuc hien viec ve duong thang
            line(x1, y1, x2, y2);
        } while (!kbhit()); // Vong lap ket thuc khi nguoi dung nhan phim bat ki
}

// Ham chinh cua chuong trinh
void main (void)
{

        randomize(); // Khoi tao bo phat sinh so ngau nhien
        InitGraph(); // Khoi tao che do do hoa

        // Cac thao do hoa
        LineDemo();
        getch();

        closegraph(); // Ngung lam viec tren che do do hoa
}
```

## 15. HỆ THỐNG TỌA ĐỘ

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **getmaxx**<br>**getmaxy** | Trả về chiều rộng (theo x) và chiều cao (theo y) tối đa của chế độ màn hình đang dùng. |
| **getx**<br>**gety** | Trả về tọa độ hiện hành của con trỏ. |
| **getviewsettings** | Lấy thông tin về viewport hiện hành. Thông tin này được lưu trong kiểu **struct viewporttype {int left, top, right, bottom, clip}** |
| **cleardevice** | Xóa toàn bộ màn hình đồ họa bằng màu nền và đưa con trỏ về vị trí (0, 0) của màn hình. |
| **clearviewport** | Xóa toàn bộ **viewport** và đưa con trỏ về vị trí (0,0) của **viewport.** |
| **setviewport** | Thiết lập **viewport** cho các thao tác đồ họa (viết, vẽ, tô, ...). |

## 16. VẼ ĐIỂM, ĐƯỜNG, VÙNG

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **putpixel** | Vẽ ra màn hình một **pixel** tại một vị trí xác định. |
| **getpixel** | Trả về màu của một **pixel** tại một vị trí cho trước. |
| **moveto** | Di chuyển vị trí con trỏ hiện hành. |
| **line** | Vẽ một đoạn thẳng. |
| **circle**<br>**ellipse** | Vẽ đường tròn, ellipse. |
| **rectangle**<br>**drawpoly** | Vẽ hình chữ nhật, đa giác. |
| **getlinesettings**<br>**setlinestyle** | Trả về/thiết lập kiểu (**style**), mẫu tô (**pattern**), bề rộng (**thickness**) được dùng cho việc vẽ các đường nói chung (đường thẳng, đa giác, hình chữ nhật...). Với hàm **getlinesettings,** các thông tin này được trả về trong một biến có cấu trúc là **struct linesettingstype.** |
| **getcolor**<br>**setcolor** | Trả về / đặt màu vẽ hiện hành. Màu vẽ này dùng cho việc vẽ các điểm (**pixel**), đường, ... |
| **setwritemode** | Đặt cách vẽ các đường mới là **COPY_PUT** hay **XOR_PUT**. Hàm này thường chỉ làm việc với **line, lineto, linerel, rectangle, drawpoly.** |
| **Tham khảo** | **moverel, lineto, linerel, arc, sector, pieslice, setaspectratio, getaspectratio, getmaxcolor, getbkcolor, setbkcolor, getpalette, setallpalette, getdefaultpalette, setrgbpalette, getpalettesize** |

**Một số điểm cần lưu ý**

- Các lệnh **setfillstyle** chỉ ảnh hưởng đến các đường trong các hàm **line, linerel, lineto, rectangle, drawpoly, ...**
- Nếu dùng chế độ vẽ là **XOR_PUT**, thì việc vẽ hai lần liên tiếp sẽ cho kết quả là vẽ và xóa đối tượng đó.

**Chương trình minh họa**

*// Ham minh hoa cac kieu duong thang (linestyle)*

```
void LineStyleDemo(void)
{
    int x1 = 50, y1 = 50, y2 = MaxY-y1, Step = 30;
    struct linesettingstype LineInfo;

    cleardevice();
    // Luu thong tin ve duong cu
    getlinesettings(&LineInfo);
    setcolor(GREEN);
    // Ve duong voi dinh dang mac dinh, do rong la 1 pixel
    for (int style=SOLID_LINE; style<=USERBIT_LINE; style++)
    {
        setlinestyle(style, 1, NORM_WIDTH);
        line(x1, y1, x1, y2);
        x1 += Step;
    }
    setcolor(YELLOW);
    // Dinh dang duong voi kieu duong ngau nhien, do rong la 3
    for (style=SOLID_LINE; style<=USERBIT_LINE; style++)
    {
        // Chon kieu duong can ve
        setlinestyle(style, 1, THICK_WIDTH);
        x1  += Step;
        line(x1, y1, x1, y2);
    }
    // Tra lai thong tin ve duong
    setlinestyle(LineInfo.linestyle,LineInfo.pattern, LineInfo.thickness);
    getch();
}
```

## 17. TÔ MÀU VÙNG

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **fillellipse** | Vẽ và tô màu ellipse. |
| **fillpoly** | Vẽ và tô màu một đa giác. |
| **floodfill** | Tô màu một vùng được bao quanh với một biên cho trước. |
| **bar** | Tô màu một vùng hình chữ nhật |
| **getfillsettings**<br>**setfillstyle** | Trả về / thiết lập mẫu tô (pattern) và màu tô dùng cho việc tô màu các đối tượng. Đối với hàm **getfillsettings**, thông tin về mẫu tô và màu tô được trả về trong một biến có cấu trúc **struct fillsettingstype** |
| **getfillpattern**<br>**setfillpattern** | Các hàm này được dùng với mẫu tô do người dùng định nghĩa. Hàm **setfillpattern** được dùng để định nghĩa một mẫu tô từ người dùng, hàm **getfillpattern** được dùng để trả về mẫu tô của người dùng đã được định nghĩa từ hàm **setfillpattern**. |
| **Tham khao** | **bar3D, pieslice** |

**Một số điểm cần lưu ý**

- Trừ hàm **bar**, các hàm như **fillellipse, fillpoly** đều vẽ đường biên của đối tượng trước (thông tin về định dạng dùng từ các hàm vẽ đường) sau đó khi tô phần ruột bên trong mới sử dụng các thông tin về màu tô và mẫu tô.
- Tuy nhiên với hàm **bar3D**, hàm này sẽ vẽ đường biên khối hộp ba chiều trước, sau đó mới tô phần ruột bên trong.

**Chương trình minh họa**

```
// Ham minh hoa cac kieu mau to (fill pattern)
void FillAreaDemo(void)
{
    int x = MaxX/2, y = MaxY/2;
    int R=MaxY/2;
    struct fillsettingstype FillInfo;
    // Luu thong tin ve cach to mau lai
    getfillsettings(&FillInfo);
    clearviewport();
    for(int pattern=SOLID_FILL; pattern<USER_FILL;pattern++)
    {
        // Chon kieu mau to
        setfillstyle(pattern, pattern);
        // Ve moi cung mot mau va mot mau to khac nhau
        pieslice(x, y, (pattern-1)*36, pattern*36, R);
    }

    // Tra lai thong tin to mau nhu cu
    setfillstyle(FillInfo.pattern, FillInfo.color);
    getch();
}
```

## 18. CÁC HÀM VẼ CHỮ

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **outtext**<br>**outtextxy** | Hiển thị một chuỗi kí tự tại một vị trí xác định trên màn hình. |
| **settextstyle** | Thiết lập font chữ, hướng của chữ, và kích thước chữ dùng cho việc hiển thị chuỗi kí tự của hàm **outtext, outtextxy.** |
| **settextjustify** | Thiết lập chế độ canh chỉnh **(text justification**) cho việc hiển thị chuỗi kí tự của hàm **outtext, outtextxy.** |
| **gettextsettings** | Trả về các thông tin về các tính chất của việc hiển thị chuỗi kí tự ra màn hình. Các thông tin bao gồm font chữ , hướng chữ, kích thước chữ, việc canh chỉnh, ... Thông tin này được trả về trong một biến có cấu trúc **struct textsettingstype.** |
| **setusercharsize** | Định kích thước của kí tự. |
| **textwidth**<br>**textheight** | Trả về kích thước theo chiều ngang **(width**) và chiều cao **(height)** tính bằng **pixel** của một chuỗi cho trước. |
| **Tham khao** | **sprintf, installuserfont, ...** |

**Một số điểm cần lưu ý**

- Người ta thường dùng hàm **sprintf** trong khi chuẩn bị một chuỗi cho hàm **outtextxy**. Ví dụ : để in chuỗi **"**Chuc mung sinh nhat lan thu **<n>** cua ban **<Ten>"** với **n** là một số nguyên và **Ten** là một chuỗi, ta dùng kết hợp hai lệnh sau :

  **sprintf**(szMsg, "Chuc mung sinh nhat lan thu **%d** cua ban **%s**", n, Ten);

  **outtextxy**(100, 100, szMsg).

**Chương trình minh họa**

```
// Ham minh hoa cac ham ve chu
void TextDemo(void)
{
    // horizontal text justification settings
    char *szHJust[] = { "LEFT_TEXT",
                        "CENTER_TEXT",
                        "RIGHT_TEXT"
                         };
    // vertical text justification settings
    char *szVJust[] = { "BOTTOM_TEXT",
                                        "CENTER_TEXT",
                                        "TOP_TEXT"
                                  };
    int x = MaxX/2, y = MaxY/2;
    char Msg[80];

    settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
    for (int hj=LEFT_TEXT; hj<=RIGHT_TEXT; hj++)
        for (int vj=BOTTOM_TEXT; vj<=TOP_TEXT; vj++)
        {
            cleardevice();
            // Thiet lap su canh chinh
            settextjustify(hj, vj);
            // In chuoi thong bao ung voi che do canh chinh
            sprintf(msg, "%s %s", szHJust[hj], szVJust[vj]);
            // Ve duong chu thap de thay duoc tac dung su canh chinh
            line(x-100, y, x+100, y);
            line(x, y-100, x, y+100);
            // Ve chu ra
            outtextxy(x, y, msg);
            getch();
        }
}
```

## 19. CÁC HÀM VỀ ẢNH BITMAP

**Nhóm các hàm và ý nghĩa sử dụng**

| Tên hàm | Ý nghĩa |
|---|---|
| **getimage** | Lưu một vùng ảnh trên màn hình vào bộ nhớ. |

| Tên hàm | Ý nghĩa |
|---|---|
| **putimage** | Đưa một ảnh trong bộ nhớ được lưu từ hàm **getimage** ra màn hình. |
| **imagesize** | Trả về kích thước (tính bằng bytes) dùng để lưu trữ một vùng ảnh trên màn hình. Hàm này dùng kèm với hàm **getimage** . |
| **Tham khao** | **setwritemode** |

**Một số điểm cần lưu ý**

- Hàm **getimage** chỉ lưu được các ảnh có kích thước không lớn hơn **64Kb**
- Hàm **putimage** lúc hiển thị một ảnh đã được lưu trong bộ nhớ khi gọi hàm **getimage** ra màn hình chỉ cần 2 tham số là tọa độ góc trên bên trái **(left, top)** thay vì phải cần 4 tham số như lúc lưu một vùng ảnh trong hàm **getimage** .

# TÀI LIỆU THAM KHẢO


**72.** Francis S. Hill. Computer Graphics. Macmillan Publishing Company, NewYork, 1990, 754 tr.

**73.** James D.Foley, Andries Van Dam, Feiner, John Hughes. Introduction to Computer Graphics. Addision Wesley, NewYork, 1995, 559 tr.

**74.** James D.Foley, Andries Van Dam, Feiner, John Hughes. Computer Graphics - Principle and Practice. Addision Wesley, NewYork, 1996, 1175 tr.

**75.** Dương Anh Đức, Lê Đình Duy. Giáo trình Đồ họa máy tính. Khoa Công nghệ thông tin, Trường Đại học Khoa học Tự nhiên (lưu hành nội bộ), 1996, 237 tr.

**76.** Hoàng Kiếm, Dương Anh Đức, Lê Đình Duy, Vũ Hải Quân. Giáo trình Cơ sở Đồ họa Máy Tính, NXB Giáo dục, 2000.

**77.** Donald Hearn, M.Pauline Baker. Computer Graphics, C version. Prentice Hall International Inc, Upper Saddle River, New Jersey, 1997, 652tr.

# MỤC LỤC