

**ĐẠI HỌC HUẾ**  
**TRƯỜNG ĐẠI HỌC KHOA HỌC**  
**KHOA CÔNG NGHỆ THÔNG TIN**

**TS. NGUYỄN HỮU TÀI**

**GIÁO TRÌNH**  
**ĐỒ HỌA MÁY TÍNH**

(ĐÀO TẠO CỬ NHÂN CÔNG NGHỆ THÔNG TIN)

**NHÀ XUẤT BẢN ĐẠI HỌC HUẾ**  
Huế, 2017

FREE VERSION

---

Mã số sách: GT/ -2017

## LỜI NÓI ĐẦU

Giáo trình *Đồ họa Máy tính* nhằm mang đến cho người học là các sinh viên ngành Công nghệ Thông tin những kiến thức cơ bản và chuyên sâu trong lĩnh vực đồ họa máy tính, rèn luyện và phát triển kỹ năng thực hành thực nghiệm, kỹ năng lập trình cho lĩnh vực đồ họa máy tính. Nội dung của giáo trình với thời lượng giảng dạy 3 tín chỉ gồm có 6 chương, 2 phục lục và 7 bài thực nghiệm được trình bày hướng dẫn chi tiết nhằm từng bước phát triển kỹ năng lập trình đồ họa, hiểu sâu và đánh giá chính xác các lý thuyết và giải thuật đồ họa. Bố cục của giáo trình gồm:

### Chương 1: Các yếu tố cơ sở của đồ họa

Trình bày các khái niệm cơ bản về thiết bị đồ họa và điểm ảnh (Pixel). Giới thiệu và trình bày chi tiết các giải thuật dựng các đường cơ bản như: Đoạn thẳng, đường tròn, ellipse. Hướng dẫn chi tiết các bước để tạo ứng dụng khung phục vụ cho việc thực hành thực nghiệm thông qua “**Bài thực nghiệm số 1**”, để từ đó dần dắt làm quen và trang bị từng bước các kiến thức lập trình đồ họa trên windows với VC++ và MFC.

### Chương 2: Các hệ màu và cơ chế tổ chức bộ nhớ màn hình

Trình bày đôi nét về cấu trúc màn hình màu. Tính chất giao thoa ánh sáng và nguyên lý tạo điểm màu trên màn hình hay máy in. Giới thiệu sơ bộ về các hệ màu RGB, CMY, HSV. Tìm hiểu về cơ chế tổ chức bộ nhớ màn hình, cách tính địa chỉ để truy xuất thông tin điểm ảnh thông qua mode đồ họa căn bản 13H, và chuẩn hiển thị đồ họa cao cấp Vesa.

### Chương 3: Các phép xén hình và tô màu

Giới thiệu phạm vi và ứng dụng của bài toán xén hình. Trình bày chi tiết các giải thuật xén hình căn bản như: Xén đoạn thẳng vào hình chữ nhật, xén đa giác vào hình chữ nhật. Giới thiệu bài toán tô màu và ứng dụng. Trình bày chi tiết 2 giải thuật tô màu gồm: Giải thuật vết dầu loang (Flood fill), và giải thuật tô đa giác theo dòng quét (Scan-line). Tìm hiểu sâu hơn về vấn đề xử lý đồ họa của hệ thống thông qua “**Bài thực nghiệm số 2**” để xử

lý bài toán tô màu theo giải thuật vết dầu loang.

#### Chương 4: Các phép biến đổi hình học

Trình bày lý thuyết biến đổi hình học affine với căn bản là các phép tính toán ma trận. Hệ tọa độ thuận nhất và lợi ích của nó trên mô hình xử lý máy tính. Một số ví dụ hướng dẫn thực hiện các bước phân tích bài toán biến hình phức tạp về thành tổng hợp của những phép biến hình cơ bản, dựa trên việc tính tích các ma trận. Phân tích bài toán quan sát vật thể trong không gian 3 chiều và sự mô phỏng thế giới thực.

#### Chương 5: Mô hình WireFrame

Trình bày chi tiết về mô hình Wireframe và cách thức tổ chức lưu trữ thông tin. Hướng dẫn xây dựng một ứng dụng mô phỏng việc quan sát vật thể 3 chiều trong không gian theo mô hình Wireframe, trong đó áp dụng kết hợp kiến thức của chương 4 và chương 5, thông qua “**Bài thực nghiệm số 3**”.

#### Chương 6: Mô hình các mặt đa giác và vấn đề khử mặt khuất

Giới thiệu mô hình các mặt đa giác, ưu và nhược điểm, cùng cách thức tổ chức lưu trữ thông tin. Giới thiệu bài toán khử mặt khuất và trình bày chi tiết các giải thuật sắp xếp theo độ sâu, giải thuật chọn lọc mặt sau, giải thuật vùng đệm độ sâu. “**Bài thực nghiệm số 4**” giúp phát triển ứng dụng 3DViewer mô phỏng việc quan sát vật thể trong không gian 3 chiều trên máy tính, trong đó vấn đề khử mặt khuất được xử lý bởi giải thuật chọn lọc mặt sau. Và “**Bài thực nghiệm số 5**” phát triển một bản nâng cấp của ứng dụng 3DViewer, trong đó vấn đề khử mặt khuất được xử lý bởi giải thuật vùng đệm độ sâu.

Cuối cùng, hệ thống 2 phụ lục nhằm giúp sinh viên có thể tìm hiểu và nghiên cứu sâu hơn một số vấn đề mà trong khuôn khổ thời gian có hạn của học phần không cho phép tìm hiểu sâu. Nội dung cơ bản của các phụ lục bao gồm:

#### Phụ lục 1: Các phương pháp dựng đường cong và mặt cong

Trình bày chi tiết các phương pháp tạo đường cong và mặt cong hiệu quả trên mô hình máy tính.

## Phụ lục 2: Các mô hình chiếu sáng

Trình bày chi tiết việc tính toán mô phỏng các loại hình chiếu sáng lên vật thể 3 chiều nhằm tăng tính trung thực của hình ảnh mô phỏng, tạo cho hình ảnh mô phỏng được trung thực và “đẹp hơn”. Đề xuất các giải pháp công nghệ xử lý kết hợp nhằm giải quyết vấn đề tốc độ thực thi cho bài toán xử lý mô phỏng hình ảnh 3 chiều trên máy tính. “*Bài thực nghiệm số 6*” và “*Bài thực nghiệm số 7*” giúp phát triển và hoàn thiện hơn nữa ứng dụng 3DView, để ứng dụng có thể mô phỏng trung thực hơn các đối tượng 3 chiều. Qua đó, giúp sinh viên có được kiến thức sâu hơn và kỹ năng vững vàng về vấn đề xử lý mô phỏng đối tượng 3 chiều trên máy tính.

Với mong muốn tạo điều kiện tốt nhất để sinh viên có thể dễ dàng lĩnh hội kiến thức lý thuyết, phát triển năng lực thực nghiệm và kỹ năng giải quyết vấn đề nói chung hay kỹ năng lập trình nói riêng, tác giả đã cố gắng để trình bày các vấn đề thuộc lĩnh vực đồ họa máy tính một cách chi tiết mạch lạc và chuẩn xác nhất có thể, các kỹ thuật xử lý luôn sát với công nghệ trong thực tiễn. Hy vọng rằng cuốn sách sẽ mang lại nhiều bổ ích cho sinh viên cũng như bạn đọc. Tác giả cũng mong nhận được nhiều đóng góp ý kiến của quý đồng nghiệp cùng bạn đọc để cuốn sách được hoàn thiện hơn trong lần tái bản sau.

Tác giả

TS. Nguyễn Hữu Tài

FREE VERSION



## MỤC LỤC

Chương 1 .....	1
1. Các khái niệm cơ bản.....	1
1.1. Thiết bị đồ họa và điểm ảnh.....	1
1.2. Điểm và đoạn thẳng trong mặt phẳng .....	2
2. Các giải thuật vẽ đoạn thẳng .....	3
2.1. Vẽ đoạn thẳng dựa vào phương trình.....	3
2.2. Vẽ đoạn thẳng dựa vào giải thuật Bresenham .....	6
2.3. Môi trường thực nghiệm và các bước thiết lập cơ bản .....	13
2.4. So sánh đánh giá hai giải thuật dựng đường thẳng .....	20
3. Các giải thuật vẽ đường tròn.....	23
3.1. Giải thuật vẽ đường tròn MidPoint.....	24
3.2. Giải thuật vẽ đường tròn Bresenham .....	29
3.3. So sánh đánh giá hai giải thuật dựng đường tròn .....	32
4. Giải thuật vẽ Ellipse.....	33
4.1. Giải thuật Bresenham cho vẽ hình Ellipse .....	35
4.2. Tóm tắt giải thuật Bresenham cho vẽ Ellipse .....	38
4.3. Cài đặt giải thuật .....	38
5. Bài tập cuối chương .....	44
Chương 2 .....	46
1. Đôi nét về cấu trúc màn hình màu .....	46
2. Các hệ màu.....	47
2.1. Hệ RGB.....	48
2.2. Hệ màu CMY .....	49
2.3. Hệ màu HSV .....	50
3. Cơ chế tổ chức bộ nhớ màn hình .....	54
3.1. Cơ chế hoạt động của chế độ màn hình độ phân giải $320 \times 200$ với 256 màu.....	55
3.2. Cơ chế hoạt động của màn hình theo chuẩn Vesa .....	56

4. Kỹ thuật thực hiện vẽ đồ họa ở hậu trường (Off-screen Rendering).....	58
<b>Chương 3 .....</b>	<b>63</b>
1. Trường hợp hình F là một tập hữu hạn điểm .....	64
2. Trường hợp xén một đoạn thẳng vào một vùng hình chữ nhật trong không gian 2 chiều.....	64
2.1. Khi cạnh của hình chữ nhật song song với trục tọa độ .....	64
2.2. Khi 1 cạnh của hình chữ nhật tạo với trục hoành một góc $\alpha$ .....	77
3. Clipping một đa giác vào trong một vùng hình chữ nhật .....	77
3.1. Giải thuật Sutherland-Hodgman .....	77
3.2. Cài đặt giải thuật .....	79
3.3. Nhược điểm của giải thuật Sutherland-Hodgman và hướng xử lý khắc phục .....	82
4. Một số giải thuật tô màu .....	83
4.1. Giải thuật vết dầu loang .....	83
4.2. Giải thuật tô màu đa giác theo dòng quét (Scan-line Algorithm).....	104
5. Bài tập cuối chương .....	115
<b>Chương 4 .....</b>	<b>117</b>
1. Các phép biến đổi hình học hai chiều (Affine 2D) .....	118
1.1. Phép tịnh tiến .....	118
1.2. Phép đồng dạng.....	118
1.3. Phép đối xứng .....	119
1.4. Phép quay quanh gốc tọa độ .....	120
1.5. Phép biến dạng (Twist Transformation) .....	120
1.6. Tọa độ thuần nhất (Homogeneous Coordinates) .....	120
1.7. Tổng hợp các phép biến đổi Affine .....	120
1.8. Phép quay quanh điểm bất kỳ .....	122
1.9. Các ví dụ minh họa .....	123

1.10. Biến đổi hệ trục tọa độ (hay biến đổi ngược) .....	126
1.11. Cài đặt .....	127
2. Các phép biến đổi Affine 3D .....	128
2.1. Các hệ trục tọa độ .....	128
2.2. Các công thức biến đổi .....	129
3. Các phép chiếu vật thể trong không gian lên mặt phẳng .....	132
3.1. Phép chiếu phối cảnh (Perspective) .....	132
3.2. Phép chiếu song song.....	133
4. Quan sát vật thể 3 chiều và quay hệ quan sát .....	133
4.1. Biến đổi từ hệ trục cục bộ sang hệ trục người quan sát ...	134
4.2. Phép chiếu phối cảnh .....	140
4.3. Phép chiếu song song.....	141
4.4. Cài đặt .....	142
5. Bài tập cuối chương .....	142
<b>Chương 5 .....</b>	<b>144</b>
1. Mô hình Wireframe .....	145
2. Vẽ hình dựa trên dữ liệu kiểu WireFrame với các phép chiếu	147
2.1. Phép chiếu trực giao đơn giản.....	147
2.2. Phép chiếu phối cảnh đơn giản .....	147
2.3. Cài đặt thực nghiệm cho mô hình wireframe.....	148
3. Bài tập cuối chương .....	160
<b>Chương 6 .....</b>	<b>162</b>
1. Mô tả đối tượng 3 chiều bằng mô hình các mặt đa giác .....	162
2. Xây dựng cấu trúc dữ liệu cho mô hình các mặt đa giác .....	164
3. Các phương pháp khử mặt khuất .....	167
3.1. Giải thuật người thợ sơn với chiến lược sắp xếp theo chiều sâu (Depth-Sorting).....	167
3.2. Giải thuật chọn lọc mặt sau (Back-Face Detection) .....	170
3.3. Cài đặt minh họa cho giải thuật chọn lọc mặt sau .....	172

3.4. Giải thuật vùng đệm độ sâu (Z-Buffer).....	190
3.5. Cài đặt minh họa cho giải thuật “vùng đệm độ sâu” .....	193
4. Bài tập cuối chương .....	203
Chương 7 .....	204
1. Nguồn sáng xung quanh .....	204
2. Nguồn sáng định hướng .....	204
2.1. Khái niệm.....	204
2.2. Tính toán mô phỏng.....	205
2.3. Cài đặt giải thuật .....	207
3. Nguồn sáng điểm .....	209
4. Mô hình bóng Phong.....	209
4.1. Khái niệm.....	209
4.2. Tính toán mô phỏng.....	211
4.3. Cài đặt giải thuật .....	216
5. 4. BÀI TẬP CUỐI CHƯƠNG.....	228
Chương 8 .....	229
1. Đường cong Bezier và mặt cong Bezier .....	230
1.1. Giải thuật De Casteljau .....	231
1.2. Dạng Bernstein của các đường cong và mặt cong Bezier	232
1.3. Dạng biểu diễn ma trận của đường Bezier.....	233
1.4. Các tính chất của đường cong Bezier .....	234
1.5. Đánh giá đường cong Bezier và sự khác biệt của đường cong Spline.....	237
2. Đường cong Spline và B-Spline .....	239
TÀI LIỆU THAM KHẢO.....	242

FREE VERSION

## DANH SÁCH HÌNH VẼ

Hình 1.1. Giao diện đồ họa windows 8 thể hiện trên màn hình của hãng Dell.....	1
Hình 1.2. Minh họa việc hiển thị hình ảnh đồ họa trên thiết bị .....	2
Hình 1.3. Ảnh minh họa một đoạn thẳng từ A(5,4) đến B(10,7).....	5
Hình 1.4. Minh họa việc chọn lựa điểm P hay Q dựa vào các tham số .....	7
Hình 1.5. Minh họa đoạn thẳng được vẽ trên thiết bị đồ họa. Các Pixel vuông màu đỏ là hình ảnh thể hiện của đoạn thẳng AB trên màn hình.....	12
Hình 1.6. Các bước tạo một project phục vụ cho quá trình thực nghiệm.....	14
Hình 1.7. Giao diện MFC Application Wizard giúp chọn lựa kiểu ứng dụng .....	15
Hình 1.8. Hình ảnh của một ứng dụng dạng dialog based làm khuôn mẫu xây dựng các ứng dụng thực nghiệm đồ họa máy tính .....	15
Hình 1.9. Thiết kế giao diện chương trình LineDemo.....	16
Hình 1.10. Menu ngữ cảnh trong quá trình tạo biến nhận dữ liệu từ edit control .....	16
Hình 1.11. Đặt tên và xác định các thông số cho biến.....	17
Hình 1.12. Các bước để thêm một hàm xử lý vào lớp CLineDemoDlg .....	17
Hình 1.13. Xác định tên hàm và các tham số .....	18
Hình 1.14. Kết quả thực thi chương trình với hình ảnh biểu diễn cho một đoạn thẳng AB được tính toán theo giải thuật Bresenham .....	20
Hình 1.15. Đồ thị toán học của đường tròn tâm O bán kính R .....	23
Hình 1.16. Minh họa việc chọn lựa điểm P hay Q.....	24
Hình 1.17. Minh họa hàm $f_{circle}$ .....	25
Hình 1.18. Minh họa việc hiển thị các điểm ảnh của đường tròn với các kích cỡ khác nhau .....	28

Hình 1.19. Hình Ellipse với các cung AC và BC .....	33
Hình 1.20. Minh họa kỹ thuật tô ellipse theo dòng quét.....	41
Hình 1.21. Hình ảnh thực nghiệm dựng đường ellipse và hình ellipse bởi giải thuật Bresenham .....	44
Hình 2.1. Màu sắc và sự giao thoa.....	46
Hình 2.2. Hai loại cấu trúc màn hình màu: (a) CRT, (b) LCD .....	47
Hình 2.3. Ảnh biểu diễn của một mũi tên màu trắng, và một chữ E trên máy tính được phóng lớn tương ứng với hai loại màn hình CRT và LCD.....	47
Hình 2.4. Biểu đồ thể hiện các sắc độ trong không gian màu chuẩn CIE 1931 .....	48
Hình 2.5. Không gian màu trong chế độ 24-bit .....	49
Hình 2.6. Hệ màu CMYK chuyên dùng trong in ấn .....	49
Hình 2.7. Hệ màu HSV biểu diễn trong chế độ số thực (từ 0 đến 1) 50	50
Hình 2.8. Hệ màu HSV biểu diễn trong chế độ lượng hóa nguyên ..	51
Hình 2.9. Minh họa việc chuyển đổi qua lại giữa 2 hệ màu HSV và RGB .....	51
Hình 2.10. Minh họa hệ màu HSL.....	54
Hình 2.11. Minh họa hệ màu Lab .....	54
Hình 2.12. Một số mode màn hình cùng thông tin chi tiết về độ phân giải và số màu có thể hiển thị, số bit phân phối cho 3 thành phần màu RGB của một điểm ảnh.....	57
Hình 2.13. Minh họa tình huống tiêu cực khi thực hiện đồ họa trực tiếp trên vùng bộ nhớ dành riêng cho màn hình với các ứng dụng đồ họa đòi hỏi nhiều thời gian xử lý. Người sử dụng có thể quan sát thấy một chuỗi các hình ảnh đang trong quá trình xây dựng, thay vì chỉ một hình ảnh hoàn thiện như mong muốn.....	59
Hình 3.1. Minh họa kỹ thuật Clipping trong phần mềm AutoCad ...	63
Hình 3.2. Minh họa việc xén đoạn thẳng AB vào hình chữ nhật.....	64
Hình 3.3. Minh họa các tình huống có thể xảy ra khi xén đoạn thẳng vào hình chữ nhật.....	69

Hình 3.4. Phân bổ mã vùng dựa theo vị trí tương ứng với hình chữ nhật.....	71
Hình 3.5. Minh họa tình huống xử lý phức tạp nhất đối với thuật toán Liang-Barsky.....	76
Hình 3.6. Minh họa bài toán xén đa giác vào hình chữ nhật .....	77
Hình 3.7. Minh họa kết quả các bước của giải thuật .....	78
Hình 3.8. Hình ảnh thực nghiệm giải thuật Sutherland-Hodgman. Với đầu vào là đa giác lớn (viền màu đỏ) chúng ta thu được đa giác nhỏ (viền màu xanh blue).....	83
Hình 3.9. Minh họa tình huống còn sai sót của giải thuật .....	83
Hình 3.10. Ảnh gốc và bản sửa đổi của nó, thu được sau khi tiến hành tô màu vùng nền đen thành nền xanh theo giải thuật vết dầu loang .....	84
Hình 3.11. Giao diện cho ứng dụng minh họa bài toán tô màu sử dụng giải thuật vết dầu loang .....	86
Hình 3.12. Hình ảnh trước khi, trong khi, và sau khi một vùng ảnh trên cửa sổ được tô màu theo giải thuật vết dầu loang.....	90
Hình 3.13. Chi phí thời gian với các hàm thao tác điểm ảnh SetPixel và GetPixel trên Device Context.....	91
Hình 3.14. Giao diện chương trình nâng cấp với nút “Fast Flood Fill” .....	94
Hình 3.15. Chi phí thời gian với các hàm thao tác điểm ảnh SetPixel và GetPixel trên Memory Device Context.....	96
Hình 3.16. Nâng cấp giao diện với nút “The best Flood Fill” .....	100
Hình 3.17. Kết quả tô màu và chi phí thời gian thực hiện hàm <b>MyBestFloodFill</b> , thông qua việc truy xuất trực tiếp vào bộ nhớ của ảnh DIB để lấy thông tin hay thiết lập thông tin của điểm ảnh .....	103
Hình 3.18. Minh họa một số tình huống của dòng quét trong giải thuật Scan-line.....	104
Hình 3.19. Minh họa hình ảnh đa giác trước (a) và sau khi được tô (b) .....	104
Hình 3.20. Hình ảnh phóng lớn mô tả quá trình quét (scan) theo hàng	

để tiến hành tô màu đa giác theo thời gian.....	105
Hình 3.21. Minh họa bài toán “mê cung” .....	105
Hình 3.22. Minh họa cơ chế nội suy giao điểm trong giải thuật tô đa giác theo dòng quét (Scanline Algorithm) .....	107
Hình 3.23. Kết quả tô đa giác có 28 đỉnh .....	115
Hình 3.24. Kết quả tô đa giác có 561 đỉnh, trong đó phần nhiều là các cạnh với độ dài rất bé.....	115
Hình 4.1. Hình ảnh thu được từ các góc quan sát khác nhau của cùng một đối tượng. Việc thay đổi góc quan sát được thực hiện thông qua các phép biến đổi hình học 3 chiều .....	117
Hình 4.2. Minh họa phép biến đổi đồng dạng cho một tam giác....	119
Hình 4.3. Hình vẽ minh họa phép quay quanh điểm M.....	124
Hình 4.4. Minh họa phép biến đổi thuận và biến đổi nghịch.....	126
Hình 4.5. Hệ tọa độ 3 chiều trực tiếp và gián tiếp .....	128
Hình 4.6. Hệ tọa độ Đè-cát và hệ tọa độ cầu .....	129
Hình 4.7. Minh họa phép chiếu phối cảnh .....	132
Hình 4.8. Minh họa bài toán quan sát vật thể trong không gian 3 chiều .....	134
Hình 4.9. Tịnh tiến hệ trục OXYZ thành O'X <sub>1</sub> Y <sub>1</sub> Z <sub>1</sub> .....	135
Hình 4.10. Quay hệ trục O'X <sub>1</sub> Y <sub>1</sub> Z <sub>1</sub> thành O'X <sub>2</sub> Y <sub>2</sub> Z <sub>2</sub> .....	136
Hình 4.11. Quay hệ trục O'X <sub>2</sub> Y <sub>2</sub> Z <sub>2</sub> thành O'X <sub>3</sub> Y <sub>3</sub> Z <sub>3</sub> .....	138
Hình 4.12. Đảo chiều trục X của hệ trục O'X <sub>3</sub> Y <sub>3</sub> Z <sub>3</sub> để thu được hệ trục quan sát O'X'Y'Z' .....	139
Hình 4.13. Phép chiếu phối cảnh trong bài toán quan sát vật thể 3 chiều .....	140
Hình 4.14. Minh họa tính chất của phép chiếu phối cảnh .....	141
Hình 5.1. Minh họa công đoạn số hóa đối tượng 3 chiều .....	145
Hình 5.2. Mô hình wireframe cho một nhân vật trong game.....	146
Hình 5.3. Cách bố trí một phép chiếu phối cảnh đơn giản .....	148
Hình 5.4. Giao diện chương trình WireFrameDemo .....	149

Hình 5.5. Menu ngữ cảnh cho phép tạo một class cho project .....	149
Hình 5.6. Tạo một MFC Class .....	150
Hình 5.7. Thiết lập tham số cho lớp CWireFrame .....	150
Hình 5.8. Thực hiện Class Wizard với lớp CWireFrameDemoDlg để thêm các sự kiện.....	157
Hình 5.9. Thêm các hàm xử lý sự kiện chuột trên cửa sổ chính của chương trình.....	157
Hình 5.10. Một số góc quan sát đối tượng được thiết lập thông qua các thao tác rê chuột.....	159
Hình 5.11. Đối tượng được thể hiện với các kích cỡ khác nhau.....	160
Hình 6.1. Hình ảnh của một số đối tượng 3 chiều thể hiện theo mô hình các mặt đa giác.....	162
Hình 6.2. Minh họa việc số hóa thông tin vật thể 3 chiều theo mô hình các mặt đa giác.....	163
Hình 6.3. Minh họa đối tượng theo mô hình các mặt đa giác: (a) Một nhân vật game theo mô hình các mặt đa giác cùng với phép ánh xạ hình ảnh bì mặt vật liệu lên các đa giác; (b) Một con Hổ với các mặt đa giác chưa tô màu; (c) Con Hổ với các đa giác được tô màu bằng phương pháp ánh xạ hình ảnh bì mặt vật liệu lên các đa giác .....	164
Hình 6.4. Minh họa sai lệch của giải thuật sắp xếp theo độ sâu khi hai mặt phẳng ở trong trạng thái cắt nhau.....	168
Hình 6.5. Minh họa sai lệch của giải thuật sắp xếp theo độ sâu khi hai mặt đa giác ở trong trạng thái chồng lên nhau .....	168
Hình 6.6. Minh họa phép kiểm tra phần kéo dài trên trục Z .....	169
Hình 6.7. Hình ảnh 2 mặt đa giác đan chéo vào nhau .....	170
Hình 6.8. Minh họa cho mô hình chọn lọc mặt sau .....	171
Hình 6.9. Kết quả thực nghiệm xử lý với hình cầu cấu tạo bởi 450 mặt đa giác theo giải thuật chọn lọc mặt sau: (a) Hình thể hiện các mặt quan sát được; (b) Hình thể hiện các mặt quan sát được và vector pháp tuyến của mỗi bì mặt đa giác (có xử lý vân đề chiếu sáng).....	172
Hình 6.10. Tạo lớp CObject_3D .....	173

Hình 6.11. Hình ảnh thực nghiệm cài đặt giải thuật chọn lọc mặt sau	183
Hình 6.12. Hình ảnh thực nghiệm cài đặt giải thuật chọn lọc mặt sau, với các đối tượng hình cầu, khủng long (Dinasaur) từ các file dữ liệu mô tả	188
Hình 6.13. Hình minh họa cách xác định tích hữu hướng của hai vector và cách áp dụng	190
Hình 6.14. Minh họa hình chiếu của 2 mặt phẳng lên mặt phẳng chiếu và phần chồng lấp (overlap) giữa chúng	191
Hình 6.15. Minh họa cơ chế nội suy giao điểm trong giải thuật tô đa giác theo dòng quét (Scanline Algorithm) có tính đến nội suy độ sâu của tạo ảnh, để có thể áp dụng vào giải thuật vùng đệm độ sâu. Ở đây $z_1$ chính là giá trị độ sâu của điểm $P_1$ , và $P_1'$ là tạo ảnh của $P_1$	193
Hình 6.16. Mô hình trực thăng (Hughes 500) được xử lý mặt khuất theo giải thuật vùng đệm độ sâu, có xử lý thêm vấn đề chiếu sáng nhằm tạo ra hình ảnh trung thực	202
Hình 6.17. Mô hình trực thăng (Hughes 500) với một số mặt được lược bỏ để có thể quan sát được phần bên trong của đối tượng. Xử lý mặt khuất theo giải thuật vùng đệm độ sâu	203
Hình 7.1. Sự khuếch tán của ánh sáng trên các bề mặt	204
Hình 7.2. Sự phản xạ của ánh sáng trên các bề mặt	205
Hình 7.3. Mô phỏng hiện tượng phản chiếu trên bề mặt đối tượng với phần sáng trắng được đánh dấu bởi vòng màu đỏ	206
Hình 7.4. Các hình cầu được số hóa theo mô hình các mặt đa giác với số mặt lần lượt là (a) 200 mặt, (b) 450 mặt, (c) 16.200 mặt	210
Hình 7.5. Minh họa kết quả xử lý tô bóng (a) Tô bóng thường, (b) Tô bóng theo giải thuật Phong	211
Hình 7.6. Vector pháp tuyến của các điểm trên một mặt cong	211
Hình 7.7. Minh họa vector pháp tuyến tại các đỉnh của đa giác	212
Hình 7.8. Minh họa các bước nội suy vector pháp tuyến cho từng điểm trên mặt đa giác	213

Hình 7.9. Hình ảnh thực nghiệm minh họa hình cầu 50 mặt không sử dụng phương pháp tô bóng cong (với chỉ một vector pháp tuyến cho mỗi bề mặt).....	213
Hình 7.10. Hình ảnh thực nghiệm minh họa hình cầu 50 mặt sử dụng phương pháp tô bóng Phong, với các vector pháp tuyến tại các đỉnh của bề mặt (hay đa giác) không cùng phương với vector pháp tuyến bề mặt mà có xu hướng ngả ra bên ngoài như mặt cong .....	214
Hình 7.11. Minh họa như Hình 7.10, nhưng được giản lược bớt một số mặt nhằm giúp quan sát rõ hơn các vector pháp tuyến tại đỉnh và bề mặt. Hình ảnh cho thấy các vector pháp tuyến tại đỉnh của bề mặt có xu hướng nghiêng ra bốn phía bên ngoài như của một mặt cong .....	215
Hình 7.12. Minh họa đối tượng với các vector nút.....	215
Hình 7.13. Kết quả thực nghiệm cài đặt với giải thuật z-buffer kết hợp phương pháp tô bóng Phong (so sánh với Hình 6.9 để thấy sự khác biệt) .....	227
Hình 7.14. Kết quả thực nghiệm so sánh giữa phương pháp tô bóng thường (a) và tô bóng Phong trên cùng một hình cầu lõm màu xám có 800 mặt đa giác .....	228
Hình 8.1. Thay đổi chất lượng hình ảnh với hàm ánh xạ có dạng đường cong Bezier bậc 3, cung cấp khả năng điều chỉnh hình dạng và độ cong của hàm ánh xạ một cách uyển chuyển và đơn giản .....	229
Hình 8.2. Minh họa việc nội suy đường cong Bezier .....	232
Hình 8.3. Đường cong Bezier bậc 3 được vẽ bởi chương trình Paint của Microsoft. Các điểm tròn (màu đỏ) chính là các điểm kiểm soát của nó.....	233
Hình 8.4. Một đường cong Spline được vẽ bởi chương trình AutoCad .....	239
Hình 8.5. Đường cong Multi-Spline với các điểm điều khiển (hay vector tiếp tuyến) giúp điều khiển độ cong.....	239

FREE VERSION



## **DANH SÁCH BẢNG BIỂU**

Bảng 3.1. Bảng quy tắc đánh mã .....	70
Bảng 4.1. Bảng ma trận của các phép biến đổi cơ bản trong không gian 2 chiều theo hệ tọa độ thuần nhất .....	121
Bảng 5.1. Danh sách thông tin lưu trữ theo mô hình WireFrame của chiếc ghế.....	145
Bảng 6.1. Bảng danh sách thông tin các đỉnh của đa giác theo mô hình các mặt đa giác .....	164

FREE VERSION



## DANH MỤC THUẬT NGỮ VÀ CHỮ VIẾT TẮT

Ký hiệu và chữ viết tắt	Giải nghĩa
Affine 2D	Không gian Affine 2 chiều
Affine geometry	Hình học Affine
Back-Face Detection	Giải thuật chọn lọc mặt sau
Bezier/Bézier	Tên của giải thuật phát sinh đường cong hay mặt cong trong lĩnh vực đồ họa
Bresenham	Tên của một số giải thuật dựng hình (đoạn thẳng, đường tròn, đường ellipse) trong giáo trình này
B-spline	Dạng tổng quát hóa của đường cong hay mặt cong Bezier
CMY	Không gian màu, được xây dựng trên 3 màu cơ sở Cyan (màu lục lam), Magenta (màu đỏ tươi), Yellow (màu vàng)
Cohen-Sutherland	Tên giải thuật xén đoạn thẳng vào hình chữ nhật
DC (Device context)	Ngữ cảnh (hay bối cảnh) thiết bị đồ họa, là một cấu trúc định nghĩa một tập các đối tượng đồ họa và các thuộc tính liên quan của chúng, có 4 loại DC khác nhau trong MFC là: Display, Printer, Memory (or compatible), và Information
Depth-Sorting	Giải thuật người thợ sơn hay sắp xếp theo chiều sâu
DIB	Ảnh bitmap không phụ thuộc thiết bị <i>Device-Independent Bitmap</i>
Flood Fill	Chỉ quá trình tô màu theo giải thuật vết dầu loang
Homogeneous Coordinates	Tọa độ thuần nhất
HSV	Không gian màu, được xây dựng trên 3 thành phần cơ sở là H (Hue, sắc màu), S (Saturation, độ bão hòa) và

	V (Value, thể hiện độ sáng)
Liang-Barsky	Tên một giải thuật xén đoạn thẳng vào hình chữ nhật
MFC	Thư viện chứa các lớp C++ dùng để bao bọc các hàm API của hệ điều hành Windows <i>Microsoft Foundation Class Library</i>
MidPoint	Tên của một giải thuật dựng đường tròn
Pixel	Điểm ảnh
Polygon mesh model	Mô hình các mặt đa giác lưu trữ thông tin của đối tượng trong không gian 3 chiều
Phong	Tên của một mô hình xử lý ánh sáng, giúp cho đối tượng 3 chiều có hình dáng cong có được hình ảnh bóng sáng sát với thực tế hơn các mô hình tạo bóng sáng khác trong lĩnh vực đồ họa
RGB	Không gian màu, được xây dựng trên 3 màu cơ sở Red, Green và Blue
ScanLine	Tên một thuật toán tô đa giác theo phương pháp quét dòng (scan line) trong giáo trình này
Sutherland-Hodgman	Tên một giải thuật xén đa giác vào hình chữ nhật
VESA	Một tổ chức tiêu chuẩn kỹ thuật cho chuẩn hiển thị trên máy tính <i>Video Electronics Standards Association</i>
WireFrame	Mô hình khung dây lưu trữ thông tin về hình dáng (bộ khung) của đối tượng trong không gian 3 chiều
Z-Buffer	Giải thuật vùng đệm độ sâu

## Chương 1

# CÁC YẾU TỐ CƠ SỞ CỦA ĐỒ HỌA

Trong chương này sẽ trình bày các khái niệm về điểm ảnh, tọa độ điểm ảnh và ma trận điểm ảnh trên thiết bị đồ họa. Trình bày các giải thuật giúp dựng hình một cách hiệu quả đối với các đối tượng cơ bản như đoạn thẳng, hình tròn, hình ellipse.

### 1. CÁC KHÁI NIỆM CƠ BẢN

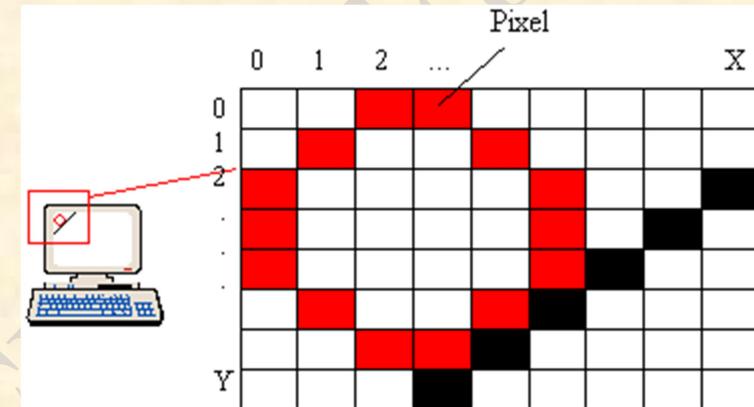
#### 1.1. Thiết bị đồ họa và điểm ảnh

Thiết bị đồ họa được hiểu là những phương tiện giúp chúng ta thể hiện được các hình ảnh thông qua sự điều khiển của máy tính. Từ đó, chúng ta có thể liệt kê một số thiết bị quen thuộc như màn hình máy tính, máy in, máy vẽ,... Hình 1.1 cho thấy khả năng hiển thị sinh động hình vẽ, chữ, hình ảnh thu được từ camera trên một màn hình máy tính của hãng Dell. Để có thể điều khiển được quá trình hiển thị thông tin (hình vẽ, chữ viết, hình ảnh,...) trên thiết bị đồ họa, chúng ta cần hiểu được tính chất cấu tạo của chúng. Trong chương này, chúng ta cần tìm hiểu một số khái niệm cơ bản liên quan đến quá trình dựng hình.



Hình 1.1. Giao diện đồ họa windows 8  
thể hiện trên màn hình của hãng Dell

Mỗi thiết bị đồ họa có một mặt phẳng (hai chiều) được phân chia thành các dòng (rows) và các cột (columns). Giao của các dòng và các cột tạo nên các điểm ảnh, thuật ngữ tiếng Anh là Pixel. Kích thước của điểm ảnh phụ thuộc vào diện tích của bề mặt hiển thị và số điểm ảnh tối đa mà thiết bị điều khiển và hiện được trên bề mặt đó. Độ phân giải của thiết bị màn hình thường được biểu diễn bởi khả năng phân chia với số cột và số dòng cực đại. Ví dụ, màn hình LCD Full HD sẽ cho khả năng phân chia được 1920 cột và 1080 dòng, từ đó tạo nên hơn 2 triệu điểm ảnh. Các cột và các dòng được đánh chỉ số bắt đầu từ 0 tại vị trí góc trên bên trái như minh họa trong Hình 1.2. Từ đó, mỗi điểm ảnh được định danh thông qua một cặp chỉ số (x,y), trong đó x và y lần lượt là chỉ số cột và chỉ số dòng tạo nên điểm ảnh, cặp chỉ số này còn được gọi là tọa độ điểm ảnh trên thiết bị đồ họa. Để thấy rằng, tọa độ điểm ảnh trên thiết bị đồ họa luôn luôn phải là một cặp số nguyên dương hoặc bằng không. Các cặp giá trị tọa độ thực (không nguyên) hoặc âm không được chấp nhận, vì nó không giúp hệ thống xác định được điểm ảnh cần điều khiển.



Hình 1.2. Minh họa việc hiển thị hình ảnh đồ họa trên thiết bị

## 1.2. Điểm và đoạn thẳng trong mặt phẳng

Về mặt toán học, một đoạn thẳng bao gồm một tập vô hạn các điểm trong mặt phẳng với cặp tọa độ thực và không có kích thước (hay kích thước vô cùng bé). Khái niệm này có nhiều khác biệt với khái niệm Pixel trên thiết bị đồ họa mà người học cần nắm vững trước khi bắt đầu tìm hiểu bài toán dựng hình trong lĩnh vực đồ họa máy tính. Từ Hình 1.2

chúng ta có thể hiểu rằng quá trình dựng hình trên thiết bị đồ họa chính là quá trình xác định một tập các điểm ảnh (pixel) sao cho chúng có thể thể hiện được hình ảnh mà chúng ta mong muốn ở mức tốt nhất (tối ưu nhất) có thể. Ví dụ, đoạn thẳng màu đen được thể hiện bằng một tập 6 pixel liên tiếp nhau như minh họa trong Hình 1.2, mỗi pixel có một kích thước cụ thể phụ thuộc vào kích thước và độ phân giải của thiết bị.

## 2. CÁC GIẢI THUẬT VẼ ĐOẠN THẲNG

Phương trình tổng quát của một đường thẳng được viết dưới dạng:

$$y = ax + b$$

Trong đó:

- $a$  là hệ số góc hay còn gọi là độ dốc, nó phản ánh mối tương quan giữa 2 biến số  $x$  và  $y$ .
- $b$  là khoảng chẵn trên trực hoành.

Phương trình đường thẳng đi qua 2 điểm  $A(x_a, y_a)$  và  $B(x_b, y_b)$  được viết dưới dạng:

$$\frac{y - y_a}{y_b - y_a} = \frac{x - x_a}{x_b - x_a} \quad (1.1)$$

Trong đó  $x_a \neq x_b$  và  $y_a \neq y_b$ .

(Khi  $x_a = x_b$  thì phương trình là  $x = x_a$ , còn khi  $y_a = y_b$  thì phương trình là  $y = y_a$ )

Đặt  $\Delta x = x_b - x_a$  và  $\Delta y = y_b - y_a$  thì (1.1) trở thành

$$\begin{aligned} y &= \frac{\Delta y}{\Delta x} x - \frac{\Delta y}{\Delta x} x_a + y_a \\ \Leftrightarrow y &= ax + b \text{ với } \begin{cases} a = \frac{\Delta y}{\Delta x} \\ b = -ax_a + y_a \end{cases} \end{aligned} \quad (1.2)$$

### 2.1. Vẽ đoạn thẳng dựa vào phương trình

Khi biết phương trình của một đường, chúng ta hoàn toàn có thể vẽ được đường biểu diễn nhờ vào các tính toán trên phương trình. Ở đây,

đường mà chúng ta cần biểu diễn là một đoạn thẳng AB với A( $x_a, y_a$ ) và B( $x_b, y_b$ ). Phương trình biểu diễn được cho bởi (1.2) với

$$x \in [x_a, x_b], y \in [y_a, y_b]$$

**Quy trình dựng hình có thể tóm tắt như sau:**

- Nếu  $|\Delta y| \leq |\Delta x|$ , nghĩa là biến số x biến thiên nhanh hơn biến số y, lúc này để đảm bảo tính liên tục của các điểm vẽ chúng ta cho biến số x thay đổi tuần tự và tính biến số y qua phương trình. Cụ thể như sau:

Cho x nhận các giá trị nguyên lần lượt từ  $x_a$  đến  $x_b$ , với mỗi giá trị x chúng ta thực hiện:

- Tính  $y = ax + b$  thông qua phương trình.
- Vẽ điểm  $(x, \text{round}(y))$ .

Ở đây điểm trên đoạn thẳng có tọa độ là  $(x, y)$ . Song chúng ta không thể vẽ điểm đó bởi giá trị y là một giá trị thực trong khi các hệ thống biểu diễn đồ họa chỉ có hữu hạn điểm và mỗi điểm có tọa độ nguyên. Từ đó chúng ta buộc phải minh họa cho điểm  $(x, y)$  thuộc đoạn thẳng thực bởi một điểm trên hệ thống thiết bị đồ họa gần với nó nhất, đó chính là điểm có tọa độ cột là x và dòng là giá trị làm tròn về số nguyên của y.

- Ngược lại, nghĩa là biến số y biến thiên nhanh hơn biến số x, lúc này để đảm bảo tính liên tục của các điểm vẽ chúng ta cho biến số y thay đổi tuần tự và tính biến số x qua phương trình. Cụ thể như sau:

Cho y nhận các giá trị nguyên lần lượt từ  $y_a$  đến  $y_b$ , với mỗi giá trị y chúng ta thực hiện:

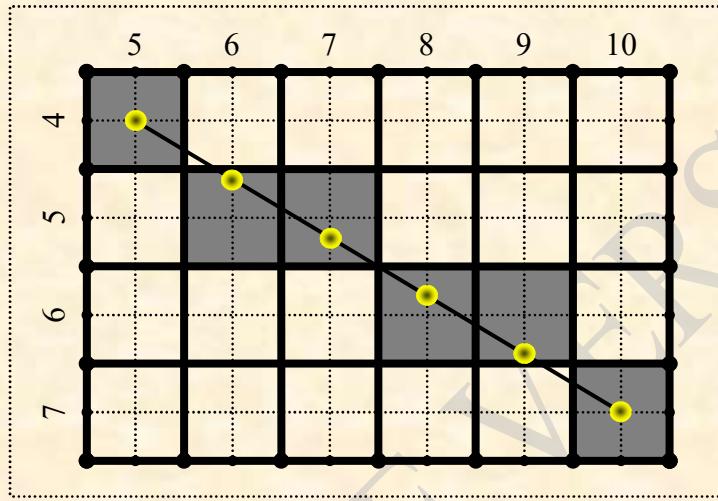
- Tính  $x = \frac{y-b}{a}$  (hay  $x = \frac{\Delta x}{\Delta y}y - \frac{\Delta x}{\Delta y}y_a + x_a$ )
- Vẽ điểm  $(\text{round}(x), y)$

**Ví dụ:** Cho A(5, 4) đến B(10, 7) để vẽ đoạn thẳng AB chúng ta thực hiện các bước sau:

Tính:  $\Delta x = x_b - x_a = 10 - 5 = 5; \Delta y = y_b - y_a = 7 - 4 = 3$

$$\begin{cases} a = \frac{\Delta y}{\Delta x} = \frac{3}{5} \\ b = -ax_a + y_a = 1 \end{cases}$$

- Vì  $|\Delta y| \leq |\Delta x|$ , nên chúng ta cho x nhận các giá trị nguyên lần lượt từ  $x_a$  đến  $x_b$ , với mỗi giá trị x chúng ta cần thực hiện:
  - Tính  $y = ax + b$  thông qua phương trình.
  - Vẽ điểm  $(x, \text{round}(y))$ .



Hình 1.3. Ảnh minh họa một đoạn thẳng từ A(5,4) đến B(10,7)

Cụ thể như sau:

Khi  $x = x_a = 5$ :  $y = ax + b = 4$ ; Vẽ điểm  $(5,4)$

Khi  $x = 6$ :  $y = 23/5 = 4.6$ ; Vẽ điểm  $(6,5)$

Khi  $x = 7$ :  $y = 26/5 = 5.2$ ; Vẽ điểm  $(7,5)$

Khi  $x = 8$ :  $y = 29/5 = 5.8$ ; Vẽ điểm  $(8,6)$

Khi  $x = 9$ :  $y = 32/5 = 6.4$ ; Vẽ điểm  $(9,6)$

Khi  $x = 10$ :  $y = 7$ ; Vẽ điểm  $(10,7)$

Kết quả chúng ta có hình vẽ đoạn thẳng AB có thể minh họa như trong Hình 1.3.

## 2.2. Vẽ đoạn thẳng dựa vào giải thuật Bresenham

Mục 2.1 đã đưa ra quy trình để vẽ một đoạn thẳng AB bất kỳ trên thiết bị đồ họa. Tuy nhiên, phương pháp tính toán còn chưa thật sự hiệu quả. Cụ thể, tại mỗi bước lặp để tìm ra được tọa độ của một điểm vẽ, chúng ta cần phải tính 1 phép nhân và 1 phép cộng trên trường số thực, cùng với một phép tính làm tròn (round) số thực về số nguyên. Cũng với cách tiếp cận trên, song giải thuật Bresenham hướng tới một sự phân tích bài toán sâu sắc hơn để đi đến một quy trình ít tính toán hơn.

Giả thiết đầu tiên mà giải thuật Bresenham đặt ra là hệ số góc của đoạn thẳng  $a \in [0, 1]$ , các trường hợp còn lại của hệ số góc như  $a \in (1, +\infty)$ ;  $a \in [-1, 0]$ ; hay  $a \in (-\infty; -1)$  có thể được quy về trường hợp đoạn thẳng có hệ số góc  $a \in [0, 1]$  thông qua các phép lấy đối xứng, quy trình xử lý cụ thể đối với các đoạn thẳng có hệ số góc  $a \notin [0, 1]$  sẽ được bàn thảo và hướng dẫn tại mục 2.2.3.

Từ giả thiết đặt ra là hệ số góc của đoạn thẳng  $a \in [0, 1]$ , chúng ta có thể suy ra rằng, trên toàn bộ đoạn thẳng tham số x luôn luôn biến thiên nhanh hơn tham số y. Từ đó, đưa đến quy trình: Cho x nhận các giá trị nguyên lần lượt từ  $x_a$  đến  $x_b$ , với mỗi giá trị x chúng ta cần phải tìm ra một giá trị y nguyên để  $(x, y)$  chính là tọa độ của điểm cần minh họa trên thiết bị. Và điểm mấu chốt ở đây là việc tìm ra giá trị y phải thông qua ít phép tính toán hơn quy trình đã trình bày ở mục 2.1.

Giả thiết với hai điểm đầu mút  $A(x_a, y_a)$  và  $B(x_b, y_b)$  có tọa độ nguyên và  $x_a < x_b$  (nếu cần thì hoán đổi hai đầu mút A và B để thỏa mãn giải thiết  $x_a < x_b$ ). Rõ ràng, điểm ảnh đầu tiên cần biểu diễn trên thiết bị chính là điểm A có tọa độ  $(x_a, y_a)$ . Nếu gọi điểm ảnh được lựa chọn đầu tiên trong quy trình là  $(x_0, y_0)$  thì:

$$(x_0, y_0) = (x_a, y_a)$$

Theo lập luận quy nạp:

- Giả thiết rằng đến bước thứ i chúng ta đã chọn được điểm ảnh thứ i, hay nói cách khác là điểm ảnh được chọn ở bước thứ i với tên gọi là  $(x_i, y_i)$  đã được xác định giá trị.

- Vậy đến bước tiếp theo, bước thứ  $i+1$ , chúng ta sẽ chọn điểm ảnh nào? Nói cách khác là điểm ảnh được chọn ở bước thứ  $(i+1)$  với tên gọi  $(x_{i+1}, y_{i+1})$  sẽ được xác định các giá trị ra sao?

**Chú ý:**  $x_i, y_i$  là tên gọi của tọa độ điểm ảnh thứ  $i$ , ví dụ  $(x_0, y_0)$  là tên gọi của điểm ảnh được lựa chọn đầu tiên ( $i = 0$ ) và có giá trị  $(x_a, y_a)$

Để trả lời câu hỏi này chúng ta cần dựa vào một số lập luận sau đây:

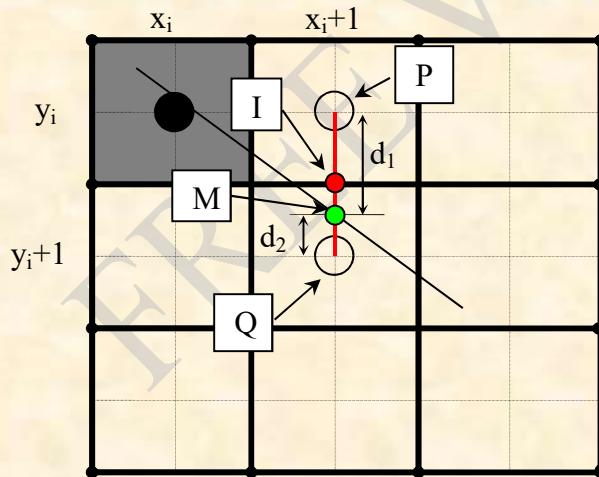
Như đã trình bày thì điểm ảnh chọn thứ  $i+1$  sẽ phải có hoành độ  $x$  bằng hoành độ của điểm ảnh được lựa chọn trước đó cộng thêm 1:

$$\text{Hay } x_{i+1} = x_i + 1$$

Gọi  $M$  là điểm thuộc  $AB$  sao cho  $x_M = x_{i+1} = x_i + 1$

$$\text{Thì: } y_M = ax_M + b = a(x_i + 1) + b = (ax_i + b) + a$$

Vậy điểm tiếp theo thuộc đoạn thẳng mà chúng ta cần tìm điểm ảnh minh họa trên thiết bị là  $M(x_i + 1, (ax_i + b) + a)$ . Câu hỏi đặt ra là chúng ta sẽ chọn điểm nào trong 2 điểm ảnh  $P(x_i + 1, y_i)$  và  $Q(x_i + 1, y_i + 1)$  để minh họa cho  $M$  trên thiết bị đồ họa (xem Hình 1.4).



Hình 1.4. Minh họa việc chọn lựa điểm  $P$  hay  $Q$  dựa vào các tham số

Để trả lời câu hỏi này chúng ta cần xem xét một biểu thức trung gian:

$$\text{Đặt } d_1 = (y_M - y_P) \text{ và } d_2 = (y_Q - y_M)$$

Xét biểu thức:

$$d_1 - d_2 = (y_M - y_P) - (y_Q - y_M) = 2y_M - (y_P + y_Q) = 2 \left[ y_M - \frac{y_P + y_Q}{2} \right]$$

Nếu gọi I là trung điểm của QP thì:  $d_1 - d_2 = 2[y_M - y_I]$

Rõ ràng là:

- Nếu  $d_1 - d_2 < 0$  dẫn đến  $y_M < y_I$ , suy ra P gần điểm M hơn Q, vậy chúng ta sẽ chọn điểm ảnh P làm điểm minh họa cho điểm M thuộc đường thẳng trên thiết bị đồ họa.
- Nếu  $d_1 - d_2 > 0$  dẫn đến  $y_M > y_I$ , suy ra Q gần điểm M hơn P, vậy chúng ta sẽ chọn điểm ảnh Q làm điểm minh họa cho M trên thiết bị đồ họa.
- Nếu  $d_1 - d_2 = 0$  dẫn đến  $y_M = y_I$ , suy ra khả năng lựa chọn P và Q là như nhau, song chúng ta phải quyết định chọn một điểm ảnh. Trong tình huống này giải thuật quy định chọn điểm Q.

Vậy để tìm được điểm minh họa tiếp theo chúng ta cần xét dấu của biểu thức  $d_1 - d_2$ . Tuy nhiên, chúng ta thấy biểu thức  $d_1 - d_2$  còn khá phức tạp, và phải thực hiện tính toán trên trường số thực do trong đó có xuất hiện phép chia:

$$\begin{aligned} y_M &= (ax_i + b) + a = (ax_i + (-ax_a + y_a)) + a \\ &= \frac{\Delta y}{\Delta x} x_i - \frac{\Delta y}{\Delta x} x_a + y_a + \frac{\Delta y}{\Delta x} \end{aligned} \tag{1.3}$$

Để tránh tính biểu thức  $d_1 - d_2$  trên trường số thực, người ta hướng tới một biểu thức tương đương về dấu đó là:

$$P_i = \Delta x(d_1 - d_2)$$

Việc đưa  $\Delta x$  vào nhằm loại bỏ mẫu số trong biểu thức  $d_1 - d_2$ , từ đó thu được biểu thức  $P_i$  tính trên trường số nguyên. Thật vậy:

$$\begin{aligned} P_i &= \Delta x(d_1 - d_2) = \Delta x(2y_M - (y_P + y_Q)) = \Delta x(2y_M - (y_i + y_i + 1)) \\ &= \Delta x(2y_M - 2y_i - 1) = 2\Delta x y_M - 2\Delta x y_i - \Delta x \end{aligned}$$

Thay  $y_M$  bởi giá trị ở (1.3) chúng ta được:

$$\begin{aligned} P_i &= 2\Delta yx_i - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - 2\Delta xy_i - \Delta x \\ &= 2\Delta yx_i - 2\Delta xy_i - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - \Delta x \end{aligned} \quad (1.4)$$

Chúng ta thấy, biểu thức  $P_i$  được xác lập từ tọa độ của điểm chọn thứ i là  $(x_i, y_i)$ . Vậy  $P_{i+1}$  sẽ được xác lập từ điểm chọn thứ  $i+1$  là  $(x_{i+1}, y_{i+1})$  như sau:

$$P_{i+1} = 2\Delta yx_{i+1} - 2\Delta xy_{i+1} - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - \Delta x \quad (1.5)$$

Vì dấu của  $P_i$  và dấu của  $(d_1 - d_2)$  là tương đương nên có thể tóm tắt quy tắc chọn điểm ảnh tiếp theo như sau:

- Nếu  $P_i < 0$ : Thì chọn điểm ảnh P làm điểm minh họa cho M trên thiết bị đồ họa. Hay nói cách khác là điểm ảnh chọn thứ  $i+1$  là  $(x_{i+1}, y_{i+1})$  sẽ có giá trị bằng P. Nghĩa là  $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$ , từ đó thay vào công thức (1.5), chúng ta có:

$$P_{i+1} = 2\Delta y(x_i + 1) - 2\Delta xy_i - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - \Delta x = P_i + 2\Delta y$$

- Nếu  $P_i \geq 0$ : Thì chọn điểm Q là điểm minh họa cho M trên thiết bị đồ họa. Hay nói cách khác là điểm chọn thứ  $i+1$  là  $(x_{i+1}, y_{i+1})$  sẽ có giá trị bằng Q. Nghĩa là:  $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + 1)$ , từ đó thay vào công thức (1.5) chúng ta có:

$$\begin{aligned} P_{i+1} &= 2\Delta y(x_i + 1) - 2\Delta x(y_i + 1) - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - \Delta x \\ &= P_i + 2\Delta y - 2\Delta x \end{aligned}$$

Khi  $i = 0$ , ta có  $(x_0, y_0) = (x_a, y_a)$  thay vào (1.4) chúng ta có:

$$P_0 = 2\Delta yx_0 - 2\Delta yx_a + 2\Delta xy_a + 2\Delta y - 2\Delta xy_0 - \Delta x = 2\Delta y - \Delta x$$

Từ đây, chúng ta thấy được quy trình chọn ra các điểm trên thiết bị để minh họa cho đoạn thẳng AB theo giải thuật Bresenham như sau:

- Điểm chọn đầu tiên ( $i = 0$ ) là  $(x_0, y_0) = (x_a, y_a)$  và giá trị  $P_0 = 2\Delta y - \Delta x$ .

- Dựa vào giá trị của  $P_0$  là âm hay dương mà chúng ta lại chọn được điểm tiếp theo  $(x_1, y_1)$  và tính được giá trị  $P_1$ .
- Dựa vào giá trị của  $P_1$  là âm hay dương mà chúng ta lại chọn được điểm tiếp theo  $(x_2, y_2)$  và tính được giá trị  $P_2$ .
- Cứ như vậy, chúng ta tìm ra được tập các điểm trên thiết bị đồ họa để minh họa cho đoạn thẳng AB.

### 2.2.1. Tóm tắt giải thuật Bresenham

**Đầu vào:** Tọa độ  $(x_a, y_a)$  và  $(x_b, y_b)$  của đoạn thẳng AB thỏa mãn giả thiết hệ số góc thuộc đoạn  $[0, 1]$  và  $x_a < x_b$ .

**Đầu ra:** Vẽ các điểm ảnh nhằm thể hiện hình ảnh đoạn thẳng AB trên mặt phẳng thiết bị đồ họa.

❖ **Bước 1:** (Bước khởi động, tính toán các giá trị ban đầu)

$$\Delta x = x_b - x_a; \Delta y = y_b - y_a;$$

$$\text{Const1} = 2\Delta y; \text{Const2} = 2\Delta y - 2\Delta x$$

$$P_0 = 2\Delta y - \Delta x; (x_0, y_0) = (x_a, y_a)$$

Vẽ điểm  $(x_0, y_0)$

❖ **Bước 2:** (Bước lặp, thực hiện tính các giá trị điểm ảnh)

Với mỗi giá trị  $i$  ( $i = 0, 1, 2, \dots$ ) chúng ta xét dấu  $P_i$

➢ Nếu  $P_i < 0$ : Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$$

$$P_{i+1} = P_i + \text{Const1}$$

➢ Ngược lại (tức  $P_i \geq 0$ ): Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i + 1)$$

$$P_{i+1} = P_i + \text{Const2}$$

Vẽ điểm  $(x_{i+1}, y_{i+1})$  vừa tìm được

❖ **Bước 3:** (Xác định điều kiện lặp)

Lặp lại bước 2 với những giá trị  $i$  tiếp theo, cho đến khi điểm tìm được trùng với B, nghĩa là  $x_{i+1} = x_b$  thì giải thuật kết thúc.

### 2.2.2. Ví dụ

Cho đoạn thẳng AB với A(5,6) và B(10,10). Sử dụng giải thuật Bresenham, chúng ta có thể tìm được các điểm ảnh cần vẽ để biểu diễn đoạn AB trên màn hình như sau:

#### ❖ Bước 1:

$$\Delta x = 10 - 5 = 5; \quad \Delta y = 10 - 6 = 4;$$

$$\text{Const1} = 2\Delta y = 8; \quad \text{Const2} = 2\Delta y - 2\Delta x = 8 - 10 = -2;$$

$$P_0 = 2\Delta y - \Delta x = 8 - 5 = 3; \quad (x_0, y_0) = (x_a, y_a) = (5, 6)$$

Vẽ điểm  $(x_0, y_0)$

#### ❖ Bước 2:

$i = 0:$

Ta có  $P_0 = 3 \geq 0$  nên:

$$(x_1, y_1) = (x_0 + 1, y_0 + 1) = (6, 7)$$

$$P_1 = P_0 + \text{Const2} = 3 + (-2) = 1$$

Vẽ điểm  $(x_1, y_1) = (6, 7)$

$i = 1:$

Ta có  $P_1 = 1 \geq 0$  nên:

$$(x_2, y_2) = (x_1 + 1, y_1 + 1) = (7, 8)$$

$$P_2 = P_1 + \text{Const2} = 1 + (-2) = -1$$

Vẽ điểm  $(x_2, y_2) = (7, 8)$

$i = 2:$

Ta có  $P_2 = -1 < 0$  nên:

$$(x_3, y_3) = (x_2 + 1, y_2) = (8, 8)$$

$$P_3 = P_2 + \text{Const1} = -1 + 8 = 7$$

Vẽ điểm  $(x_3, y_3) = (8, 8)$

$i = 3:$

Ta có  $P_3 = 7 \geq 0$  nên:

$$(x_4, y_4) = (x_3 + 1, y_3 + 1) = (9, 9)$$

$$P_4 = P_3 + \text{Const2} = 7 + (-2) = 5$$

Vẽ điểm  $(x_4, y_4) = (9, 9)$

$i = 4$ :

Ta có  $P_4 = 5 \geq 0$  nên:

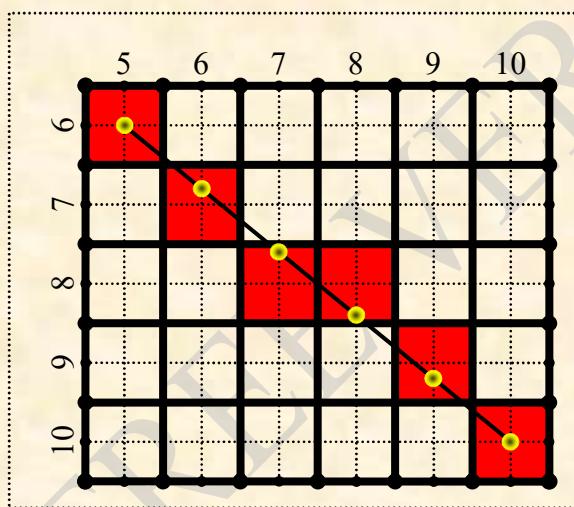
$$(x_5, y_5) = (x_4 + 1, y_4 + 1) = (10, 10)$$

$$P_5 = P_4 + \text{Const2} = 5 + (-2) = 3$$

Vẽ điểm  $(x_5, y_5) = (10, 10)$

Vì  $x_5 = x_b = 10$ , nên kết thúc vòng lặp và cũng là kết thúc giải thuật.

Hình vẽ minh họa (xem Hình 1.5)



Hình 1.5. Minh họa đoạn thẳng được vẽ trên thiết bị đồ họa. Các Pixel vuông màu đỏ là hình ảnh thể hiện của đoạn thẳng AB trên màn hình

### 2.2.3. Hướng dẫn cho các trường hợp hệ số góc ngoài đoạn [0, 1]

Giả sử cho  $A(0, 50)$ ,  $B(100, 10)$ , để dựng đoạn thẳng AB chúng ta cần tiến hành một số phân tích:

$$\Delta x = x_b - x_a = 100 - 0 = 100$$

$$\Delta y = y_b - y_a = 10 - 50 = -40$$

Suy ra hệ số góc  $a = \Delta y / \Delta x = -0.4 \in [-1, 0]$

Lúc này, ta cần lấy đối xứng của AB qua trục OX để được CD với C(0, -50) D(100, -10), nên xét trên đoạn thẳng CD chúng ta có:

$$\Delta x = x_d - x_c = 100 - 0 = 100$$

$$\Delta y = y_d - y_c = (-10) - (-50) = 40$$

Suy ra hệ số góc  $a = \Delta y / \Delta x = 0.4 \in [1, 0]$  thỏa mãn điều kiện của giải thuật Bresenham. Từ đó, chúng ta có thể áp dụng giải thuật Bresenham để tính toán ra các điểm ảnh cần vẽ trên CD, nhưng chúng ta sẽ không vẽ nó (vì mục đích chúng ta là dựng hình AB) mà lại lấy đối xứng qua trục OX (*tức đối xứng ngược lại với lúc đầu*) rồi mới vẽ, thì lúc này các điểm ảnh vẽ ra sẽ là hình ảnh của đoạn thẳng AB. Như thế, CD chỉ đóng vai trò trung gian nhằm áp dụng được giải thuật còn kết quả sau cùng chúng ta thu được vẫn là hình ảnh minh họa cho đoạn AB.

Áp dụng tương tự:

- Trường hợp hệ số góc  $a \in (1, +\infty)$ , lúc này chúng ta cần lấy đối xứng qua đường phân giác của góc phần tư thứ nhất để hệ số góc được quy về  $[0, 1]$ .
- Trường hợp hệ số góc  $a \in (-\infty, -1)$ , lúc này chúng ta cần lấy 2 lần đối xứng. Đối xứng qua OX rồi tiếp đến đối xứng qua đường phân giác.

Xét điểm M(x,y). Đối xứng qua OX chúng ta được tọa độ mới là (x, -y). Tiếp đến lấy đối xứng qua tia phân giác góc phần tư thứ nhất chúng ta được (-y, x). Hay nói cụ thể hơn là với một đoạn thẳng đầu vào có tọa độ là A(x<sub>a</sub>, y<sub>a</sub>) và B(x<sub>b</sub>, y<sub>b</sub>) thì đoạn thẳng trung gian của nó sẽ là CD với C(-y<sub>a</sub>, x<sub>a</sub>) và D(-y<sub>b</sub>, x<sub>b</sub>).

### 2.3. Môi trường thực nghiệm và các bước thiết lập cơ bản

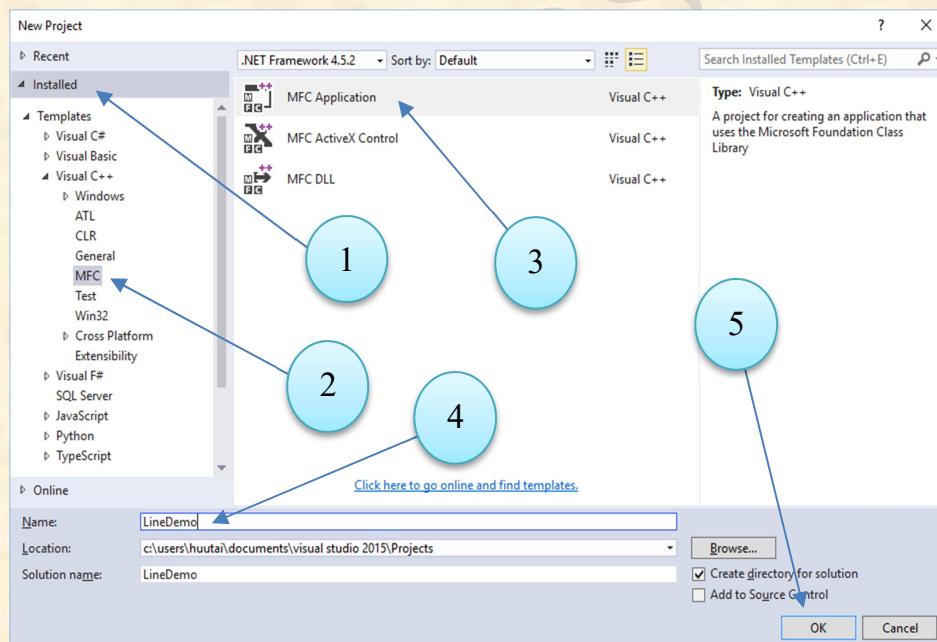
Chúng tôi đề xuất sinh viên thực hành thực nghiệm trên môi trường Microsoft Visual C++ sử dụng thư viện MFC (*Microsoft Foundation Class Library*) để có thể dễ dàng tương tác sâu với thiết bị đồ họa và mang lại năng lực thực thi mạnh mẽ. Đồng thời, nó cũng phù hợp với nền tảng kiến thức lập trình hướng đối tượng và kỹ năng lập trình với ngôn ngữ C++ mà sinh viên đã được trang bị trước khi đến với môn học này.

Tuy nhiên, sinh viên cũng cần phải trang bị và trau dồi thêm khả năng lập trình xử lý sự kiện trên giao diện đồ họa của Windows và một số kiến thức cơ bản về MFC.

### **Bài thực nghiệm số 1:**

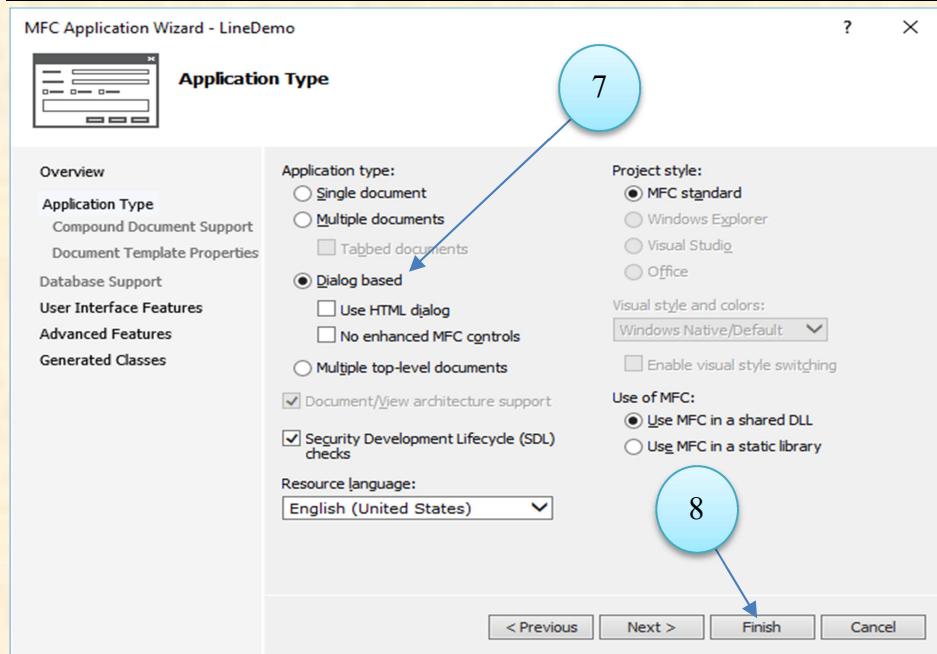
Dưới đây là các bước hướng dẫn để sinh viên có thể tiếp cận với quy trình xây dựng một ứng dụng đồ họa cơ bản, ứng dụng này bước đầu đặt nền tảng cho quá trình thực hành thực nghiệm, là quá trình quan trọng giúp người học có thể kiểm tra tính đúng đắn, tính thực tiễn của các lý thuyết đã được học thông qua kết quả thực nghiệm và các phân tích đánh giá, mang lại hiểu biết sâu sắc và đa chiều trên thực tiễn.

- ❖ Bước 1: Tạo một dự án (project) mới trong Microsoft Visual Studio sử dụng ngôn ngữ Visual C++ và thư viện MFC:



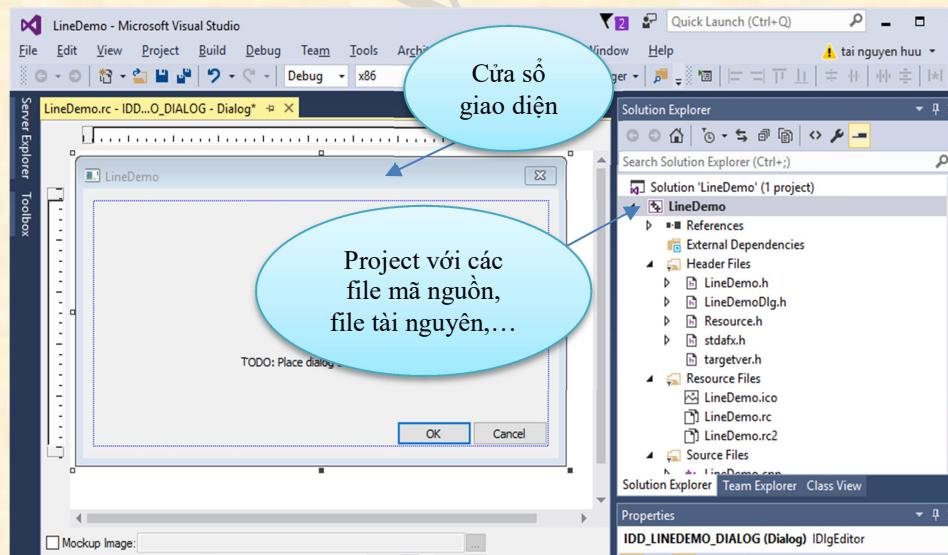
Hình 1.6. Các bước tạo một project phục vụ cho quá trình thực nghiệm

Khi hộp thoại MFC Application Wizard xuất hiện, cần bấm nút Next để chuyển đến mục Application Type. Tiếp đến click chọn mục “Dialog based” như hình dưới đây, rồi cuối cùng bấm nút Finish để kết thúc quá trình tạo khung ứng dụng.



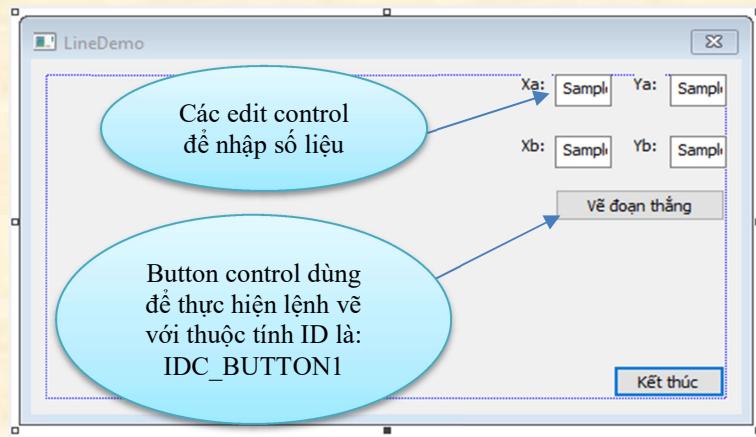
Hình 1.7. Giao diện MFC Application Wizard giúp chọn lựa kiểu ứng dụng

Kết thúc quá trình trên, Microsoft sẽ tạo ra một khung ứng dụng (template / bản mẫu) với các thành phần và giao diện như hình dưới đây:



Hình 1.8. Hình ảnh của một ứng dụng dạng dialog based làm khuôn mẫu xây dựng các ứng dụng thực nghiệm đồ họa máy tính

❖ Bước 2: Thiết kế lại giao diện

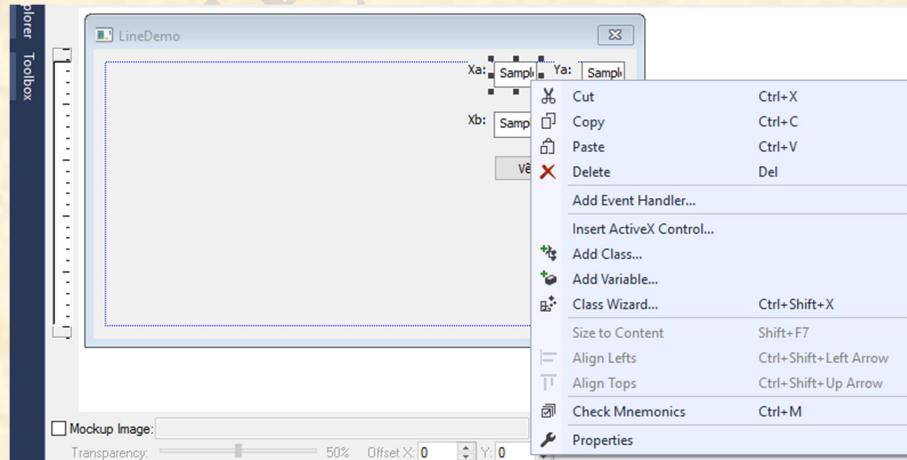


Hình 1.9. Thiết kế giao diện chương trình LineDemo

Tùy thuộc vào yêu cầu của bài toán mà chúng ta có những giải pháp thiết kế giao diện khác nhau. Hình 1.9 là thiết kế đơn giản cho bài toán dựng đoạn thẳng AB với các tọa độ được nhập vào thông qua các hộp nhập liệu (edit control).

❖ Bước 3: Tạo các biến nhận dữ liệu từ các edit control

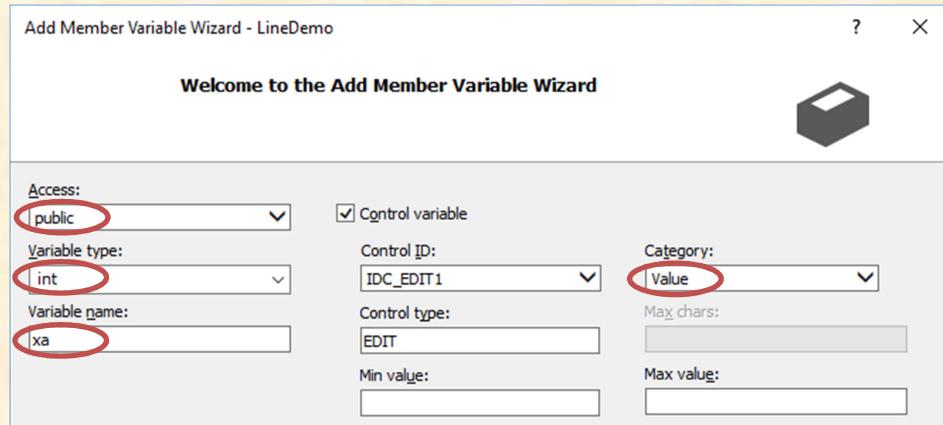
Click chuột phải vào edit control cần tạo biến nhận dữ liệu, tiếp đến chọn mục “Add variable...” trong menu ngữ cảnh.



Hình 1.10. Menu ngữ cảnh trong quá trình tạo biến nhận dữ liệu từ edit control

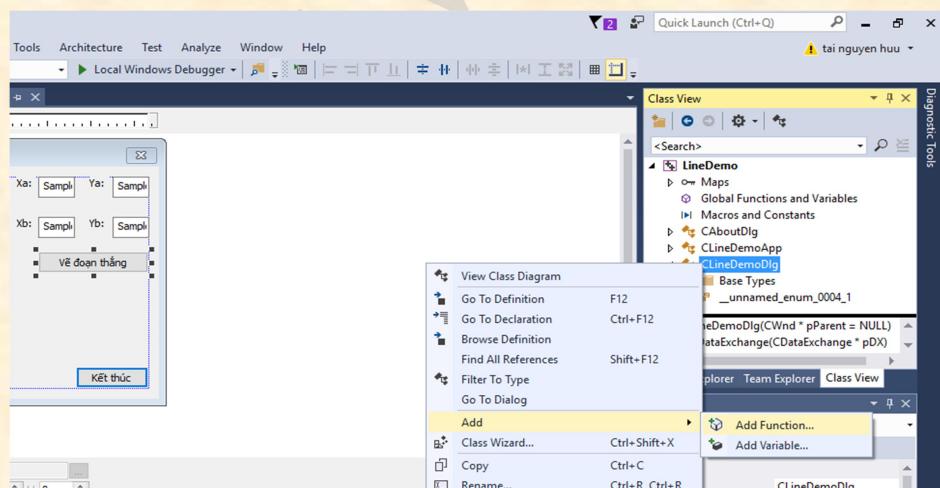
Sau đó, cần thiết lập thuộc tính *Category* là *Value*, kế đến là tên biến và các thông số cơ bản khác như trong *Hình 1.11*.

Thực hiện tương tự với các edit control còn lại để tạo các biến nhận dữ liệu tọa độ của đoạn thẳng AB. Các biến cần tạo gồm: xa, ya, xb, yb kiểu số nguyên.



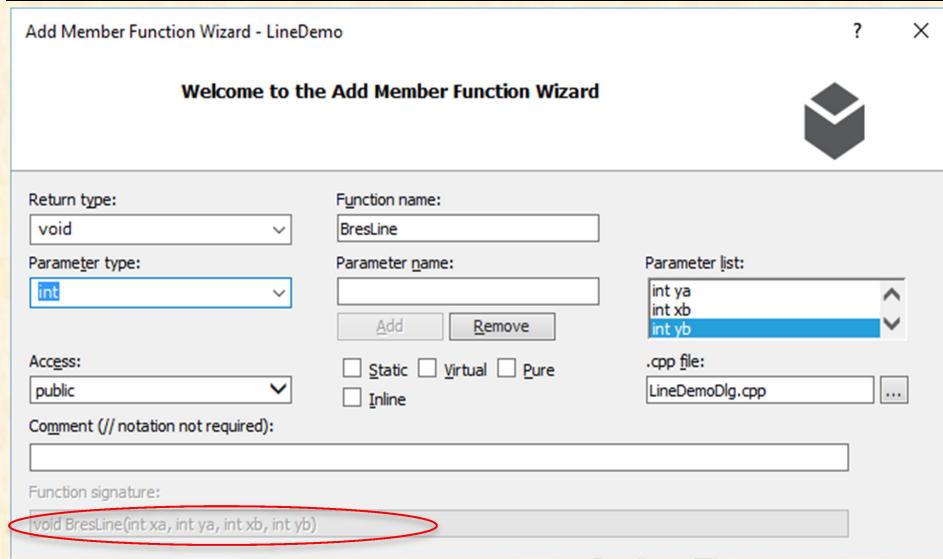
*Hình 1.11. Đặt tên và xác định các thông số cho biến*

- ❖ Bước 4: Thêm hàm BresLine vào lớp CLineDemoDlg với chức năng tính toán theo giải thuật Bresenham và vẽ hình lên cửa sổ giao diện.



*Hình 1.12. Các bước để thêm một hàm xử lý vào lớp CLineDemoDlg*

Đầu tiên chúng ta click vào tab “Class View”, tiếp đến click chuột phải vào mục CLineDemoDlg, rồi chọn mục Add→Add Function...



Hình 1.13. Xác định tên hàm và các tham số

Sau cùng, cần xác định tên hàm, kiểu trả về của hàm và danh sách các tham số như trong Hình 1.13. Sau khi hoàn thành cần bấm nút **Finish** để chuyển sang công đoạn viết mã lệnh cho hàm.

❖ Bước 5: Viết mã lệnh thực thi cho hàm **BresLine**:

```
void CLineDemoDlg::BresLine(int xa, int ya, int xb, int yb)
{
    /* Giải thích đầu vào thỏa mãn hai điều kiện của giải thuật
    Bresenham Là hệ số góc a thuộc [0, 1] và xa < xb. Từ đó, chúng ta
    chỉ cần thực hiện theo đúng các bước đã được nêu ra trong giải
    thuật.

    Để giải quyết với đầu vào tổng quát, sinh viên cần tham khảo
    thêm phần hướng dẫn xử lý cho các tình huống hệ số góc a ngoài
    đoạn [0, 1]. Phần này được xem như một yêu cầu nâng cấp mã Lệnh mà
    sinh viên cần thực hiện */

    /* Lấy con trỏ quản lý đối tượng Device Context của cửa sổ giao
    diện chương trình để có thể tiến hành vẽ các điểm ảnh trên cửa sổ
    */
}

CDC *p_DC = this->GetDC();
COLORREF Color = RGB(255, 0, 0); //Màu đỏ

//Thực hiện Bước 0:
```

```

int Dx = xb - xa, Dy = yb - ya;
int P = 2 * Dy - Dx, Const1 = 2 * Dy, Const2 = 2 * Dy - 2 * Dx;
int x = xa, y = ya;

p_DC->SetPixel(x, y, Color);

// Thực hiện Bước 2 (bước Lặp):

while (x < xb) // Điều kiện dừng (Bước 3)
{
    x++;
    if (P < 0)
    {
        P += Const1;
    }
    else
    {
        y++;
        P += Const2;
    }
    p_DC->SetPixel(x, y, Color);
}
}

```

- ❖ Bước 6: Bấm DoubleClick vào nút “Vẽ đoạn thẳng” (tức nút có ID là IDC\_BUTTON1) để chương trình MS Visual Studio tự động thêm vào một hàm đáp ứng sự kiện khi người sử dụng click (hay bấm chọn) vào nút “Vẽ đoạn thẳng” có dạng:

```

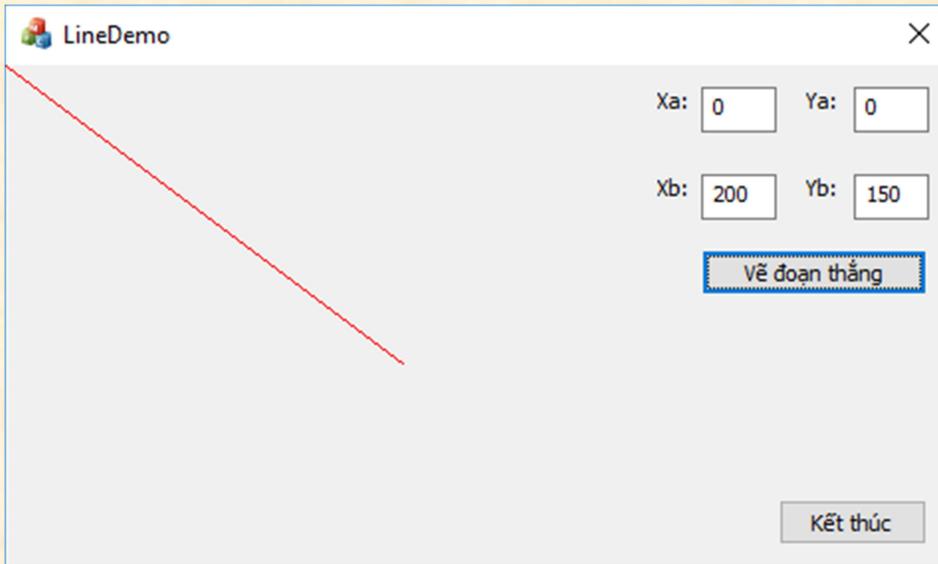
void CLineDemoDlg::OnBnClickedButton1()
{}
```

Tiếp đến chúng ta cần viết mã lệnh cho hàm này như sau:

```

void CLineDemoDlg::OnBnClickedButton1()
{
    /* Gọi lệnh thực thi việc cập nhật dữ liệu vào các biến xa, ya,
    xb, yb */
    UpdateData(true);
    // Gọi hàm vẽ đoạn thẳng theo giải thuật Bresenham đã cài đặt
    BresLine(xa, ya, xb, yb);
}
```

- ❖ Bước 7: Biên dịch và thực thi chương trình, nhập tọa độ của đoạn thẳng AB vào các edit control, rồi click nút “Vẽ đoạn thẳng” để thấy được hình ảnh biểu diễn cho đoạn thẳng AB được hiển thị trên cửa sổ như hình dưới đây:



Hình 1.14. Kết quả thực thi chương trình với hình ảnh biểu diễn cho một đoạn thẳng AB được tính toán theo giải thuật Bresenham

#### 2.4. So sánh đánh giá hai giải thuật dựng đường thẳng

Sau đây, chúng ta sẽ tiến hành so sánh và đánh giá hai giải thuật dựng đoạn thẳng trên một số tiêu chí quan trọng:

##### 1. Yêu cầu về phần cứng thực thi tính toán

Rõ ràng rằng, hai giải thuật có hai yêu cầu khác nhau về phần cứng thực thi tính toán. Cụ thể:

- Giải thuật vẽ đoạn thẳng dựa vào phương trình: Yêu cầu tính toán với kiểu dữ liệu số thực, vì vậy một hệ thống đồ họa nếu sử dụng giải thuật này sẽ cần có vi xử lý hỗ trợ tính toán số thực, yêu cầu này không phải khi nào cũng được đáp ứng. Thực tế cho thấy, rất nhiều hệ thống thiết bị trong công nghiệp cho đến giải trí và gia dụng có màn hình hiển thị đồ họa cấp thấp (độ phân giải thấp) với nhiệm vụ hiển thị các thông tin đơn

giản), được xây dựng trên các vi xử lý số nguyên 8-bit, 16-bit, hay 32-bit nhằm đáp ứng yêu cầu về giá thành. Do đó, các hệ thống này không thể đáp ứng được yêu cầu của giải thuật, hoặc chỉ đáp ứng miếng cưỡng theo một cách phức tạp hơn đó là xây dựng các modul phần mềm hỗ trợ xử lý số thực dựa trên các lệnh tính toán số nguyên. Ngay cả vi xử lý máy tính 16 bit của hãng Intel có mã hiệu là 8086 được thiết kế trong giai đoạn 1976 - 1978, và trang bị trên các máy IBM PC, có kiến trúc bộ xử lý số học và logic (Arithmetic Logic Unit, hay ALU) chỉ thực hiện các tính toán trên trường số nguyên. Vì vậy, các tính toán số thực trên máy tính giai đoạn đó phải được thực hiện thông qua giải pháp phần mềm, mất nhiều thời gian thực thi.

- Giải thuật Bresenham: Chỉ yêu cầu các tính toán căn bản trên trường số nguyên gồm: So sánh, cộng, trừ, và phép dịch bit để thực hiện phép nhân 2. Những yêu cầu tính toán này là hết sức căn bản và có thể thực hiện được trên hầu hết các hệ thống vi xử lý.

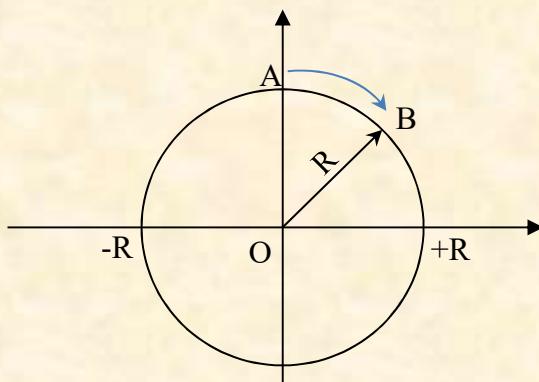
## 2. Số lệnh máy cần thực hiện để tính toán tọa độ các điểm ảnh

Giải thuật vẽ đoạn thẳng dựa vào phương trình	Giải thuật vẽ đoạn thẳng Bresenham
<i>Tính toán trong phần khởi động</i>	
$\Delta x = x_b - x_a$	$\Delta x = x_b - x_a$
$\Delta y = y_b - y_a$	$\Delta y = y_b - y_a$
$a = \Delta y / \Delta x$	$Const1 = 2\Delta y$
$b = -a * x_a + y_a$	$Const2 = 2\Delta y - 2\Delta x$
$x = x_a$	$P = 2\Delta y - \Delta x$
$y = y_a$	$x = x_a$
	$y = y_a$
<b>Nhận xét:</b> Số lệnh máy cần thực hiện ở giai đoạn khởi động của cả hai giải thuật không khác nhau đáng kể. Mặt khác, do bước khởi động chỉ thực hiện một lần, nên góp phần không đáng kể vào chi phí thời gian tính toán của toàn bộ quy trình thực thi của giải thuật.	

Tính toán trong vòng lặp để thu được tọa độ một điểm ảnh minh họa	
$x = x + 1$ $y = \text{int}(a.x + b + 0.5)$	$x = x + 1$ Nếu $P < 0$ : $P = P + \text{Const1}$ Ngược lại: $y = y + 1$ $P = P + \text{Const2}$
<ul style="list-style-type: none"> <li>❖ Tổng cộng:</li> <li>- 3 lệnh cộng số thực</li> <li>- 1 lệnh nhân số thực</li> <li>- 1 lệnh lấy phần nguyên của số thực</li> </ul>	<ul style="list-style-type: none"> <li>❖ Tổng cộng trong tình huống xấu nhất:</li> <li>- 3 lệnh cộng số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul> <ul style="list-style-type: none"> <li>❖ Tổng cộng trong tình huống tốt nhất:</li> <li>- 2 lệnh cộng số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul>
<p><b>Nhân xét:</b> Ở đây, chúng ta chỉ so sánh mang tính tương đối, vì 2 giải thuật có yêu cầu nền tảng tính toán cùng các lệnh máy khác nhau, nên sẽ rất khác nhau khi thực hiện trên các nền tảng kiến trúc vi xử lý khác nhau. Nhưng nhìn chung, giải thuật dựa vào phương trình ngoài yêu cầu phải thực hiện trên dữ liệu số thực, nó còn nhiều hơn một lệnh nhân số thực, vì thế khi đoạn thẳng càng dài, kéo theo số lần lặp càng lớn, thì hiệu quả cải thiện về thời gian tính toán của giải thuật Bresenham càng cao.</p>	

Từ những so sánh và đánh giá trên cho thấy, chúng ta nên lựa chọn giải thuật Bresenham để cài đặt trên các hệ thống đồ họa để mang lại hiệu quả thực thi tốt nhất, còn giải thuật dựng đường tròn dựa vào phương trình chỉ mang tính tham khảo và so sánh.

### 3. CÁC GIẢI THUẬT VẼ ĐƯỜNG TRÒN



Hình 1.15. Đồ thị toán học của đường tròn tâm O bán kính R

Phương trình đường tròn tâm O (góc tọa độ) bán kính R (nguyên) là:

$$X^2 + Y^2 = R^2$$

Trong mục này, chúng ta chỉ cần tìm phương pháp vẽ đường tròn có tâm tại gốc tọa độ. Bởi vì, nếu chúng ta vẽ được đường tròn có tâm tại gốc, thì bằng cách thêm vào phép tịnh tiến chúng ta sẽ thu được đường tròn có tâm ở vị trí bất kỳ.

Dễ thấy, để vẽ được đường tròn tâm gốc tọa độ chúng ta chỉ cần tìm phương pháp vẽ cung một phần tam AB như trên Hình 1.15, kết hợp với các phép lấy đối xứng chúng ta sẽ có các phần còn lại của đường tròn.

Với cung AB thì rõ ràng độ dốc của nó thuộc đoạn  $[-1, 0]$ . Điều này chúng ta có thể dễ dàng thấy qua góc của tiếp tuyến với cung AB hay qua đạo hàm phương trình biểu diễn cung AB.

Vì cung AB có độ dốc trong khoảng  $[-1, 0]$ , nên chúng ta suy ra rằng trên toàn bộ cung AB khi biến số x tăng thì biến số y giảm, và tốc độ thay đổi của y chậm hơn của x. Từ đây, chúng ta có thể đề ra một quy trình dựng cung AB nhằm đảm bảo tính liên tục (hay nối tiếp, liền kề) của các pixel được chọn, cụ thể:

- Cho biến số x nhận lần lượt các giá trị nguyên từ  $x_a$  đến  $x_b$ . Với mỗi giá trị x, chúng ta thực hiện:
  - Tìm giá trị y nguyên tương ứng để pixel có tọa độ  $(x, y)$  sẽ là điểm

gần nhất với điểm có tọa độ thực ( $x, y_{circle}$ ) vốn thuộc đường tròn về mặt toán học.

- Vẽ pixel ( $x, y$ ) tìm được và các điểm đối xứng của nó để có được đường tròn.

Trong mục này, chúng ta sẽ đi tìm hiểu 2 giải thuật cho phép dựng đường tròn (*thực chất là dựng cung AB và các đối xứng của nó*) một cách hiệu quả về mặt tốc độ tính toán.

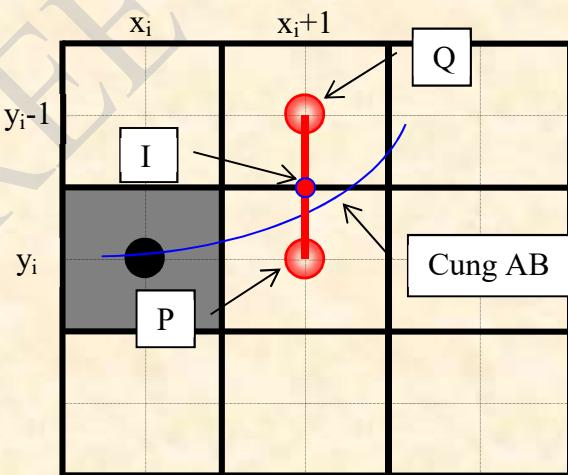
### 3.1. Giải thuật vẽ đường tròn MidPoint

Giải thuật MidPoint hay còn gọi là giải thuật xét điểm giữa.

- Điểm ảnh (hay Pixel) đầu tiên được chọn để dựng cung AB sẽ chính là điểm A(0,R), nghĩa là điểm chọn đầu tiên của quy trình dựng hình với tên gọi  $(x_0, y_0)$  sẽ có tọa độ  $(0, R)$ . Nói cách khác, chúng ta có:

$$(x_0, y_0) = (0, R)$$

- Giả sử ở bước thứ i, chúng ta đã xác định được giá trị điểm ảnh cần vẽ của bước này là  $(x_i, y_i)$ , hay nói cách khác là điểm vẽ thứ i là  $(x_i, y_i)$  đã được xác định giá trị. Câu hỏi đặt ra là đến bước thứ  $i+1$ , chúng ta sẽ chọn điểm vẽ thứ  $(i+1)$  như thế nào? Hay nói theo một cách khác là điểm chọn thứ  $(i+1)$  với tên gọi là  $(x_{i+1}, y_{i+1})$  có giá trị bằng bao nhiêu?



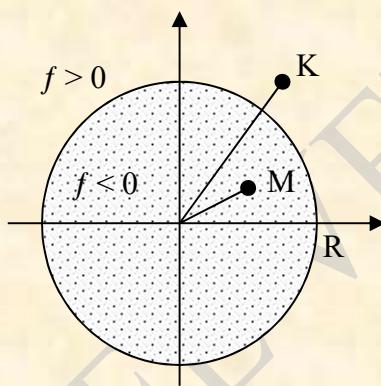
Hình 1.16. Minh họa việc chọn lựa điểm P hay Q dựa vào các tham số khi dựng cung AB trên màn hình

Vì điểm ảnh chọn lựa tiếp theo của quy trình dựng hình phải được thực hiện trên cơ sở tuân thủ quy tắc đã đề ra ở phần đầu mục 3, nên hoành độ  $x$  của nó phải tăng một giá trị so với giá trị của điểm chọn trước đó, hay nói cách khác là:

$$x_{i+1} = x_i + 1$$

Mặt khác, trên cung AB (xem Hình 1.16) khi  $x$  tăng thì  $y$  giảm, và tốc độ thay đổi của  $y$  chậm hơn của  $x$ , nên với giá trị  $x$  tăng 1 đơn vị (hay một cột) thì giá trị  $y$  sẽ thay đổi một lượng  $\Delta y$  với ràng buộc  $-1 \leq \Delta y \leq 0$ . Mà điểm ảnh chọn bước trước là  $(x_i, y_i)$  nên điểm ảnh chọn ở bước tiếp theo chỉ có thể là một trong hai điểm ảnh  $P(x_i+1, y_i)$  và  $Q(x_i+1, y_i-1)$ .

Để quyết định được điểm chọn là  $P$  hay  $Q$ , chúng ta hướng đến một biểu thức mà dấu của nó cho phép chúng ta ra quyết định chọn điểm nào.



Hình 1.17. Minh họa hàm  $f_{circle}$

Trước hết, chúng ta xét hàm (xem Hình 1.17):

$$f_{circle}(x, y) = x^2 + y^2 - R^2$$

Với một điểm  $M(x, y)$  thì rõ ràng chúng ta có:

- $f_{circle}(M) = f(x, y) = x^2 + y^2 - R^2 < 0$  khi và chỉ khi điểm  $M$  nằm trong đường tròn (tâm  $O$  bán kính  $R$ ).
- $f_{circle}(M) = f(x, y) = x^2 + y^2 - R^2 > 0$  khi và chỉ khi điểm  $M$  nằm ngoài đường tròn (tâm  $O$  bán kính  $R$ ).
- $f_{circle}(M) = f(x, y) = x^2 + y^2 - R^2 = 0$  khi và chỉ khi điểm  $M$  nằm trên đường tròn.

Từ kết quả trên, nếu chúng ta gọi I là trung điểm của đoạn thẳng PQ thì tọa độ của I là  $(x_i + 1, y_i - 0.5)$  và:

$$\begin{aligned} \text{Đặt } P_i &= f_{\text{circle}}(I) = f_{\text{circle}}(x_i + 1, y_i - 0.5) \\ &= (x_i + 1)^2 + (y_i - 0.5)^2 - R^2 \end{aligned} \quad (1.6)$$

- Khi  $P_i = f_{\text{circle}}(I) < 0$ : Thì điểm I nằm trong đường tròn tâm O bán kính R, vì thế điểm P sẽ gần với đường tròn hơn điểm Q. Suy ra điểm P sẽ được chọn làm điểm ảnh biểu diễn cho cung AB ở bước thứ i+1.
- Khi  $P_i = f_{\text{circle}}(I) > 0$ : Thì điểm I nằm ngoài đường tròn, vì thế điểm Q sẽ gần với đường tròn hơn điểm P. Suy ra điểm Q sẽ được chọn làm điểm ảnh biểu diễn cho cung AB ở bước thứ i+1.
- Khi  $P_i = f_{\text{circle}}(I) = 0$ : Thì điểm I nằm trên đường tròn, suy ra khả năng lựa chọn P và Q là như nhau, song chúng ta phải quyết định chọn một điểm. Trong tình huống này giải thuật quy định chọn điểm Q.

Từ đây chúng ta thấy, dấu của biểu thức  $P_i$  có thể được sử dụng như một hàm ra quyết định cho việc lựa chọn điểm ảnh tiếp theo trong quá trình dựng hình.

Để giải thuật được đơn giản người ta tối ưu hóa việc tính  $P_i$  theo công thức truy hồi, cụ thể:

$$P_{i+1} = f(x_{i+1} + 1, y_{i+1} - 0.5) = (x_{i+1} + 1)^2 + (y_{i+1} - 0.5)^2 - R^2 \quad (1.7)$$

Dấu của  $P_i$  sẽ quyết định giá trị  $P_{i+1}$  như sau:

- Nếu  $P_i < 0$ : Thì điểm chọn tiếp theo là  $P(x_i + 1, y_i)$ , điều đó có nghĩa là  $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$ . Thay vào (1.7) chúng ta thu được:

$$P_{i+1} = (x_i + 1 + 1)^2 + (y_i - 0.5)^2 - R^2 = P_i + 2x_i + 3$$

- Nếu  $P_i \geq 0$ : Thì điểm chọn tiếp theo là  $Q(x_i + 1, y_i - 1)$ , điều đó có nghĩa là  $(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$ . Thay vào (1.7) chúng ta được:

$$P_{i+1} = (x_i + 1 + 1)^2 + (y_i - 1 - 0.5)^2 - R^2 = P_i + 2(x_i - y_i) + 5$$

Đầu tiên, chúng ta chọn điểm A(0,R), nghĩa là  $(x_0, y_0) = (0, R)$ , thay vào (1.6) chúng ta có:

$$P_0 = 1 - R + \frac{1}{4} = \frac{5}{4} - R$$

Từ đây, quy trình dựng hình có thể được thực hiện theo trình tự sau:

- Tính  $P_0$ , vẽ điểm  $(x_0, y_0) = (0, R)$ .
- Dựa vào dấu của  $P_0$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_1, y_1)$  và giá trị  $P_1$ .
- Dựa vào dấu của  $P_1$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_2, y_2)$  và giá trị  $P_2$ .
- Quá trình trên được lặp đi lặp lại cho đến khi chúng ta vẽ được điểm nguyên gần nhất với B.
- ❖ Một điểm đáng chú ý ở đây là: Các giá trị  $P$  tiếp theo có được bằng cách cộng giá trị  $P$  trước đó với một lượng nguyên  $2x_i + 3$  hoặc  $2(x_i - y_i) + 5$  tùy theo dấu của  $P$ . Song, nếu giá trị  $P$  khởi đầu là  $P_0 = 1 - R + \frac{1}{4}$  là một giá trị thực sẽ làm cho việc tính các giá trị  $P$  tiếp theo cũng phải xử lý trên trường số thực. Một điều dễ thấy là nếu chúng ta thay đổi giá trị  $P_0$  khởi đầu thành  $1 - R$  thì dấu của  $P_0$  và các  $P_i$  có được sau đó không hề thay đổi (mặc dù giá trị có bị giảm một lượng 0.25), dẫn đến kết quả chọn lựa các điểm ảnh của giải thuật không hề bị thay đổi, nhưng các tính toán giá trị  $P$  chỉ cần thực hiện trên trường số nguyên. Sự thay đổi này giúp giảm độ phức tạp tính toán nếu xét trên khía cạnh thực hiện bằng phần mềm (software) hay giảm yêu cầu về kiến trúc phần cứng nếu xét trên khía cạnh thực hiện bằng phần cứng (Hardware).

### 3.1.1. Tóm tắt giải thuật vẽ đường tròn MidPoint

- ❖ **Bước 1:** (Bước khởi động, tính toán các giá trị ban đầu)

$$P_0 = 1 - R; (x_0, y_0) = (0, R)$$

Vẽ điểm  $(x_0, y_0)$

❖ **Bước 2:** (Bước lặp, thực hiện tính các giá trị điểm ảnh)

Với mỗi giá trị  $i$  ( $i=0,1,2,\dots$ ) chúng ta xét dấu  $P_i$ .

➤ Nếu  $P_i < 0$ : Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$$

$$P_{i+1} = P_i + 2x_i + 3$$

➤ Ngược lại (tức  $P_i \geq 0$ ): Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$$

$$P_{i+1} = P_i + 2(x_i - y_i) + 5$$

Vẽ điểm  $(x_{i+1}, y_{i+1})$  vừa tìm được.

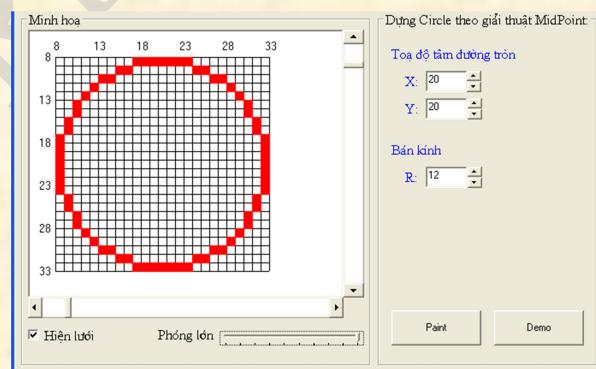
❖ **Bước 3:** (Xác định điều kiện lặp)

Lặp lại bước 2 với những giá trị  $i$  tiếp theo, cho đến khi chúng ta vẽ được điểm nguyên gần nhất với B, nghĩa là:

$$x_{i+1} = \text{round}(x_b) = \text{Round}\left(\frac{R}{\sqrt{2}}\right) \text{ thì giải thuật kết thúc.}$$

### 3.1.2. Cài đặt

- ❖ Sinh viên cần xây dựng hàm vẽ đường tròn theo giải thuật đã trình bày trên. Trên cơ sở đó, mở rộng chương trình LineDemo với chức năng mới là vẽ đường tròn.
- ❖ Nâng cấp chương trình với khả năng vẽ phóng lớn, với thiết kế giao diện như hình dưới đây:



Hình 1.18. Minh họa việc hiển thị các điểm ảnh của đường tròn với các kích cỡ khác nhau

### 3.2. Giải thuật vẽ đường tròn Bresenham

Lập luận tương tự giải thuật trên, song không dùng hàm  $f_{circle}$  mà dùng biểu thức  $(d_1 - d_2)$ . Giải thuật được trình bày chi tiết như sau:

- Điểm ảnh đầu tiên được chọn để vẽ sẽ là điểm A(0, R), nghĩa là:

$$(x_0, y_0) = (0, R)$$

- Giả sử đến bước thứ i chúng ta đã chọn được điểm ảnh  $(x_i, y_i)$  để vẽ. Câu hỏi đặt ra là đến bước thứ  $i+1$  chúng ta sẽ chọn điểm ảnh có tên gọi  $(x_{i+1}, y_{i+1})$  với giá trị bằng bao nhiêu?

Vì điểm ảnh tiếp theo sẽ chọn theo quy tắc đã nêu ở phần đầu mục 3, nên hoành độ x của điểm ảnh chọn ở bước sau sẽ tăng một giá trị so với điểm ảnh được lựa chọn ở bước trước, hay nói cách khác là:

$$x_{i+1} = x_i + 1$$

Đồng thời, trên cung AB khi x tăng thì y giảm và tốc độ thay đổi của y chậm hơn của x, nên rõ ràng chúng ta thấy rằng với giá trị x tăng 1 thì giá trị y sẽ thay đổi một lượng  $\Delta y$  với điều kiện ràng buộc là  $-1 \leq \Delta y \leq 0$ . Mà điểm ảnh chọn bước trước là  $(x_i, y_i)$  nên điểm ảnh chọn tiếp theo chỉ có thể là một trong hai điểm P( $x_i + 1, y_i$ ) và Q( $x_i + 1, y_i - 1$ ).

Để quyết định được điểm ảnh cần chọn là P hay Q, chúng ta hướng đến một biểu thức mà dấu của nó cho phép chúng ta ra quyết định chọn điểm nào.

$$\text{Đặt: } d_1 = y_P^2 - y^2 \text{ và } d_2 = y^2 - y_Q^2$$

giá trị y ở đây chính là tung độ của cung AB ứng với hoành độ  $x = x_i + 1$

$$\begin{aligned} \text{Đặt: } P_i &= d_1 - d_2 = y_P^2 + y_Q^2 - 2y^2 = y_i^2 + (y_i - 1)^2 - 2(R^2 - x_{i+1}^2) \\ &= y_i^2 + (y_i - 1)^2 - 2(R^2 - (x_i + 1)^2) = y_i^2 + (y_i - 1)^2 - 2R^2 + 2(x_i + 1)^2 \end{aligned} \quad (1.8)$$

Dấu của biểu thức  $P_i$  cho phép xác định điểm ảnh được chọn tiếp theo là P hay Q.

- Khi  $P_i < 0$ : Thì điểm ảnh P sẽ gần với đường tròn hơn điểm ảnh Q, do đó chúng ta sẽ chọn điểm ảnh P làm điểm biểu diễn (vẽ).

- Khi  $P_i > 0$ : Thì điểm ảnh Q sẽ gần với đường tròn hơn P, do đó chúng ta sẽ chọn điểm ảnh Q làm điểm biếu diễn.
- Khi  $P_i = 0$ : Thì khoảng cách từ P và Q đến đường tròn đều bằng nhau, nên chúng ta có thể chọn P hay Q đều được. Trong tình huống này giải thuật quy ước chọn điểm ảnh Q làm điểm biếu diễn.

Từ đây, chúng ta thấy có thể dựa vào dấu của biểu thức  $P_i$  để ra quyết định chọn điểm tiếp theo.

Để giải thuật được đơn giản người ta tối ưu hóa việc tính  $P_i$  theo công thức truy hồi:

$$P_{i+1} = y_{i+1}^2 + (y_{i+1} - 1)^2 - 2R^2 + 2(x_{i+1} + 1)^2 \quad (1.9)$$

Dấu của  $P_i$  sẽ quyết định giá trị  $P_{i+1}$  cụ thể như sau:

- Nếu  $P_i < 0$ : Thì điểm ảnh chọn tiếp theo là P( $x_i + 1, y_i$ ), nghĩa là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$$

Thay vào (1.9) chúng ta được:

$$\begin{aligned} P_{i+1} &= y_i^2 + (y_i - 1)^2 - 2R^2 + 2((x_i + 1) + 1)^2 = P_i + 2(2(x_i + 1) + 1) \\ &= P_i + 4x_i + 6 \end{aligned}$$

- Nếu  $P_i \geq 0$ : Thì điểm ảnh chọn tiếp theo là Q( $x_i + 1, y_i - 1$ ), nghĩa là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$$

Thay vào (1.9) chúng ta được:

$$\begin{aligned} P_{i+1} &= (y_i - 1)^2 + (y_i - 2)^2 - 2R^2 + 2((x_i + 1) + 1)^2 \\ &= P_i + (-4y_i + 4) + 2(2(x_i + 1) + 1) = P_i + 4(x_i - y_i) + 10 \end{aligned}$$

Đầu tiên, chúng ta chọn điểm ảnh A(0,R), nghĩa là  $(x_0, y_0) = (0, R)$ , thay vào (1.8) chúng ta có:

$$\begin{aligned} P_0 &= y_0^2 + (y_0 - 1)^2 - 2R^2 + 2(x_0 + 1)^2 = R^2 + (R - 1)^2 - 2R^2 + 2 \\ &= R^2 + R^2 - 2R + 1 - 2R^2 + 2 = 3 - 2R \end{aligned}$$

Vậy quy trình vẽ được thực hiện như sau:

- Tính  $P_0$ , vẽ điểm ảnh  $(x_0, y_0) = (0, R)$ .
- Dựa vào dấu của  $P_0$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_1, y_1)$  và giá trị  $P_1$ .
- Dựa vào dấu của  $P_1$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_2, y_2)$  và giá trị  $P_2$ .
- Quá trình trên được lặp đi lặp lại cho đến khi chúng ta vẽ được điểm ảnh nguyên gần nhất với B.

### 3.2.1. Tóm tắt giải thuật vẽ đường tròn Bresenham

❖ **Bước 1:** (Bước khởi động, tính toán các giá trị ban đầu)

$$P_0 = 3 - 2R; (x_0, y_0) = (0, R)$$

Vẽ điểm  $(x_0, y_0)$ .

❖ **Bước 2:** (Bước lặp, thực hiện tính các giá trị điểm ảnh)

Với mỗi giá trị  $i$  ( $i=0, 1, 2, \dots$ ) chúng ta xét dấu  $P_i$ .

➤ Nếu  $P_i < 0$ : Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$$

$$P_{i+1} = P_i + 4x_i + 6$$

➤ Ngược lại (tức  $P_i \geq 0$ ): Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$$

$$P_{i+1} = P_i + 4(x_i - y_i) + 10$$

Vẽ điểm  $(x_{i+1}, y_{i+1})$  vừa tìm được.

❖ **Bước 3:** (Xác định điều kiện lặp)

Lặp lại bước 2 với những giá trị  $i$  tiếp theo, cho đến khi chúng ta vẽ được điểm nguyên gần nhất với B. Điều này có nghĩa là khi  $x_{i+1} = \text{Round}(x_b)$  thì giải thuật kết thúc.

### 3.2.2. Cài đặt

Sinh viên cần cài đặt một hàm vẽ đường tròn theo giải thuật Bresenham và chương trình sử dụng hàm để vẽ các đường tròn ngẫu nhiên.

### 3.3. So sánh đánh giá hai giải thuật dựng đường tròn

Sau đây, chúng ta sẽ tiến hành so sánh và đánh giá hai giải thuật dựng đường tròn trên một số tiêu chí quan trọng:

#### 1. Yêu cầu về phần cứng thực thi tính toán

Rõ ràng rằng cả hai giải thuật MidPoint và Bresenham đều chỉ yêu cầu tính toán trên trường số nguyên, vì vậy yêu cầu về phần cứng thực thi tính toán của cả hai giải thuật là như nhau, hay không có sự khác biệt.

#### 2. Số lệnh máy cần thực hiện để tính toán được tọa độ điểm ảnh

Giải thuật vẽ đường tròn MidPoint	Giải thuật vẽ đường tròn Bresenham
<i>Tính toán trong phần khởi động</i>	
$P = 1 - R$ $x = 0$ $y = R$	$P = 3 - 2R$ $x = 0$ $y = R$
<i>Nhân xét:</i> Số lệnh máy cần thực hiện ở giai đoạn khởi động của cả hai giải thuật không khác nhau đáng kể. Mặt khác, do bước khởi động chỉ thực hiện một lần, nên góp phần không đáng kể vào chi phí thời gian tính toán của toàn bộ quy trình thực thi của giải thuật.	
<i>Tính toán trong vòng lặp để thu được tọa độ một điểm ảnh minh họa</i>	
$x = x + 1$ Nếu $P < 0$ : $P = P + 2x + 3$ Ngược lại: $y = y - 1$ $P = P + 2(x - y) + 5$	$x = x + 1$ Nếu $P < 0$ : $P = P + 4x + 6$ Ngược lại: $y = y - 1$ $P = P + 4(x - y) + 10$
❖ Tổng cộng trong tình huống xấu nhất: - 3 lệnh cộng số nguyên - 2 lệnh trừ số nguyên	❖ Tổng cộng trong tình huống xấu nhất: - 3 lệnh cộng số nguyên - 2 lệnh trừ số nguyên

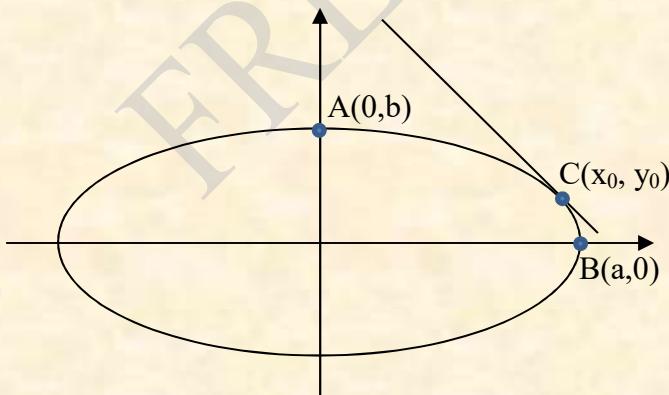
<ul style="list-style-type: none"> <li>- 1 lệnh nhân số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul> <p>❖ Tổng cộng trong tình huống tốt nhất:</p> <ul style="list-style-type: none"> <li>- 3 lệnh cộng số nguyên</li> <li>- 1 lệnh nhân số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul>	<ul style="list-style-type: none"> <li>- 1 lệnh nhân số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul> <p>❖ Tổng cộng trong tình huống tốt nhất:</p> <ul style="list-style-type: none"> <li>- 3 lệnh cộng số nguyên</li> <li>- 1 lệnh nhân số nguyên</li> <li>- 1 lệnh chuyển hướng thực thi (hay lệnh nhảy)</li> </ul>
<p><b>Nhận xét:</b> Dễ thấy rằng, cả hai giải thuật là giống nhau về các loại lệnh và số lệnh cần thực thi.</p>	

Qua kết quả so sánh trên có thể giúp chúng ta đi đến kết luận rằng cả hai giải thuật là tương đương nhau trên các tiêu chí đánh giá. Vì vậy, chúng ta có thể chọn lựa một trong hai giải thuật trên để cài đặt trên các hệ thống đồ họa, và chúng mang lại hiệu quả thực thi ngang nhau.

#### 4. GIẢI THUẬT VẼ ELLIPSE

Phương trình chính tắc của Ellipse có dạng:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \quad (1.10)$$



Hình 1.19. Hình Ellipse với các cung AC và BC

Để dựng được ellipse, chúng ta chỉ cần tìm cách dựng cung AB, các phần còn lại dễ dàng có được bằng cách lấy đối xứng.

Tư tưởng chung để dựng một đường bất kỳ là cần phải xác định ra các miền mà trên phạm vi toàn miền luôn xác định được một biến số biến thiên nhanh hơn biến số còn lại.

Rõ ràng, trên cung AB thì độ dốc giảm liên tục từ điểm A (độ dốc bằng 0) đến B (độ dốc tiến đến  $-\infty$ ). Xét về tốc độ biến thiên của 2 biến số thì:

- Tốc độ biến thiên của biến số X giảm dần từ A đến B.
- Tốc độ biến thiên của biến số Y tăng dần từ A đến B.

Rõ ràng, trên cung AB phải có một điểm mà tại đó tốc độ biến thiên của X và Y bằng nhau (song, x tăng thì y giảm), đó chính là điểm có độ dốc bằng -1.

Gọi C( $x_0, y_0$ ) là điểm nằm trên cung AB của ellipse mà tiếp tuyến tại đó có độ dốc bằng -1. Khi đó, tiếp tuyến d của ellipse sẽ có dạng:

$$\frac{x \cdot x_0}{a^2} + \frac{y \cdot y_0}{b^2} = 1$$

Mặt khác, về mặt lý thuyết thì độ dốc của d sẽ bằng đạo hàm của cung AB tại tiếp điểm C( $x_0, y_0$ ). Từ đó, chúng ta có:

$$\text{Đạo hàm của cung } AB \text{ tại điểm } C = -\frac{x_0 b^2}{y_0 a^2} = -1$$

$$\text{Suy ra: } y_0^2 = \frac{b^4}{a^4} x_0^2 \quad (1.11)$$

Đồng thời, do C thuộc ellipse nên chúng ta có:

$$\frac{x_0^2}{a^2} + \frac{y_0^2}{b^2} = 1 \quad (1.12)$$

Từ (1.11) và (1.12) chúng ta suy ra:

$$x_0 = \frac{a^2}{\sqrt{a^2 + b^2}} \text{ và } y_0 = \frac{b^2}{\sqrt{a^2 + b^2}}$$

Chúng ta sẽ trình bày giả thuật để vẽ cung AC (Đi từ A đến C theo chiều kim đồng hồ). Cung CB được thực hiện một cách tương tự khi chúng ta đổi vai trò của x và y. Các phần còn lại của ellipse có được bằng cách lấy đối xứng.

Trên cung AC độ dốc nằm trong đoạn  $[0, -1]$ , nghĩa là x tăng thì y giảm và tốc độ biến thiên của x lớn hơn của y. Vậy, tư tưởng của giải thuật dựng cung AC sẽ là: Cho tham số x biến thiên tuần tự từ  $x_a$  đến  $x_c$  với các giá trị nguyên, và với mỗi giá trị x chúng ta tìm một giá trị y nguyên gần nhất với giá trị y thực của ellipse.

Phản tiếp theo dưới đây sẽ trình bày chi tiết giải thuật Bresenham áp dụng cho dựng ellipse.

#### 4.1. Giải thuật Bresenham cho vẽ hình Ellipse

Rõ ràng, điểm ảnh đầu tiên được chọn để dựng cung AC sẽ là điểm A(0, b), nghĩa là:

$$(x_0, y_0) = (0, b)$$

Giả sử đến bước thứ i chúng ta đã chọn được điểm ảnh  $(x_i, y_i)$  để vẽ. Câu hỏi đặt ra là đến bước thứ  $i+1$ , chúng ta sẽ chọn điểm ảnh có tên gọi  $(x_{i+1}, y_{i+1})$  với giá trị bằng bao nhiêu?

Vì điểm tiếp theo sẽ có hoành độ x tăng một giá trị so với giá trị của điểm chọn trước, hay nói cách khác là:

$$x_{i+1} = x_i + 1$$

Đồng thời, vì trên cung AC khi x tăng thì y giảm và tốc độ thay đổi của y chậm hơn của x, nên rõ ràng chúng ta thấy là với giá trị x tăng 1 thì giá trị y thay đổi một lượng  $\Delta y$  với ràng buộc  $-1 \leq \Delta y \leq 0$ . Mà điểm chọn bước trước là  $(x_i, y_i)$  nên điểm chọn tiếp theo  $(x_{i+1}, y_{i+1})$  chỉ có thể là một trong hai điểm P( $x_i + 1, y_i$ ) và Q( $x_i + 1, y_i - 1$ ).

Để quyết định được điểm chọn là P hay Q, chúng ta hướng đến một biểu thức mà dấu của nó cho phép chúng ta ra quyết định chọn điểm nào.

$$\text{Đặt: } d_1 = y_P^2 - y^2 \text{ và } d_2 = y^2 - y_Q^2$$

(giá trị y ở biểu thức trên là tung độ của cung AC ứng với hoành độ đang xét  $x_{i+1}$ ).

$$\begin{aligned}
 \text{Đặt } P_i &= (d_1 - d_2)a^2 = [y_P^2 + y_Q^2 - 2y^2].a^2 \\
 &= [y_i^2 + (y_i - 1)^2 - 2(\frac{b^2(a^2 - x_{i+1}^2)}{a^2})].a^2 \\
 &= [y_i^2 + (y_i - 1)^2 - \frac{2b^2(a^2 - (x_i + 1)^2)}{a^2}].a^2 \\
 &= y_i^2.a^2 + (y_i - 1)^2.a^2 - 2b^2[a^2 - (x_i + 1)^2] \\
 &= y_i^2.a^2 + (y_i - 1)^2.a^2 - 2b^2.a^2 + 2b^2.(x_i + 1)^2
 \end{aligned} \tag{1.13}$$

(lượng  $a^2$  được đưa vào nhằm mục đích khử mẫu của  $(d_1 - d_2)$ , song không làm cho  $P_i$  và  $(d_1 - d_2)$  khác dấu).

Dấu của biểu thức  $P_i$  cho phép chúng ta xác định điểm ảnh tiếp theo là P hay Q. Cụ thể:

- Khi  $P_i < 0$ : Thì điểm P sẽ sát với cung AC hơn điểm Q, do đó chúng ta sẽ chọn điểm P làm điểm biểu diễn (vẽ).
- Khi  $P_i > 0$ : Thì điểm Q sẽ sát với cung AC hơn điểm P, do đó chúng ta sẽ chọn điểm Q làm điểm biểu diễn.
- Khi  $P_i = 0$ : Thì khoảng cách từ P và Q đến cung AC đều bằng nhau, nên chúng ta có thể chọn P hay Q đều được. Trong tình huống này giải thuật quy ước chọn điểm Q làm điểm biểu diễn.

Vậy từ đây, chúng ta thấy có thể dựa vào dấu của biểu thức  $P_i$  để ra quyết định chọn điểm tiếp theo.

Để giải thuật được đơn giản người ta tối ưu hóa việc tính  $P_i$  theo công thức truy hồi:

$$P_{i+1} = y_{i+1}^2.a^2 + (y_{i+1} - 1)^2.a^2 - 2b^2.a^2 + 2b^2.(x_{i+1} + 1)^2 \tag{1.14}$$

Dấu của  $P_i$  sẽ quyết định giá trị  $P_{i+1}$  cụ thể như sau:

- Nếu  $P_i < 0$ : Thì điểm chọn tiếp theo là  $P(x_i + 1, y_i)$ , nghĩa là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i).$$

Thay vào (1.14) chúng ta được:

$$\begin{aligned} P_{i+1} &= y_i^2 \cdot a^2 + (y_i - 1)^2 \cdot a^2 - 2b^2 \cdot a^2 + 2b^2 \cdot [(x_i + 1) + 1]^2 \\ &= P_i + 2b^2 \cdot [2(x_i + 1) + 1] \\ &= P_i + 2b^2(2x_i + 3) \end{aligned}$$

- Nếu  $P_i \geq 0$ : Thì điểm chọn tiếp theo là  $Q(x_i + 1, y_i - 1)$ , nghĩa là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$$

Thay vào (1.14) chúng ta được:

$$\begin{aligned} P_{i+1} &= (y_i - 1)^2 \cdot a^2 + (y_i - 2)^2 \cdot a^2 - 2b^2 \cdot a^2 + 2b^2 \cdot [(x_i + 1) + 1]^2 \\ &= P_i + a^2(-4y_i + 4) + 2b^2[2(x_i + 1) + 1] \\ &= P_i + 4a^2(1 - y_i) + 2b^2(2x_i + 3) \\ &= P_i + 2b^2(2x_i + 3) + 4a^2(1 - y_i) \end{aligned}$$

Đầu tiên, chúng ta chọn điểm  $A(0, b)$ , nghĩa là  $(x_0, y_0) = (0, b)$ , thay vào (1.13) chúng ta có:

$$\begin{aligned} P_0 &= y_0^2 \cdot a^2 + (y_0 - 1)^2 \cdot a^2 - 2b^2 \cdot a^2 + 2b^2 \cdot (x_0 + 1)^2 \\ &= b^2 \cdot a^2 + (b - 1)^2 \cdot a^2 - 2a^2 \cdot b^2 + 2b^2 \\ &= b^2 \cdot a^2 + a^2 \cdot b^2 - 2a^2 \cdot b + a^2 - 2a^2 \cdot b^2 + 2b^2 = -2a^2 \cdot b + a^2 + 2b^2 \\ &= a^2(1 - 2b) + 2b^2 \end{aligned}$$

Vậy quy trình vẽ được thực hiện như sau:

- Tính  $P_0$ , vẽ điểm  $(x_0, y_0) = (0, b)$ .
- Dựa vào dấu của  $P_0$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_1, y_1)$  và giá trị  $P_1$ .
- Dựa vào dấu của  $P_1$  chúng ta lại chọn được điểm vẽ tiếp theo  $(x_2, y_2)$  và giá trị  $P_2$ .
- Quá trình trên được lặp đi lặp lại cho đến khi chúng ta vẽ được điểm nguyên gần nhất với C.

## 4.2. Tóm tắt giải thuật Bresenham cho vẽ Ellipse

❖ **Bước 1:** (Bước khởi động, tính toán các giá trị ban đầu)

$$P_0 = a^2(1 - 2b) + 2b^2; (x_0, y_0) = (0, b)$$

Vẽ điểm  $(x_0, y_0)$ .

❖ **Bước 2:** (Bước lặp, thực hiện tính các giá trị điểm ảnh)

Với mỗi giá trị  $i$  ( $i = 0, 1, 2, \dots$ ) chúng ta xét dấu  $P_i$

➤ Nếu  $P_i < 0$ : Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i)$$

$$P_{i+1} = P_i + 2b^2(2x_i + 3)$$

➤ Ngược lại (tức  $P_i \geq 0$ ): Thì xác định điểm tiếp theo là:

$$(x_{i+1}, y_{i+1}) = (x_i + 1, y_i - 1)$$

$$P_{i+1} = P_i + 2b^2(2x_i + 3) + 4a^2(1 - y_i)$$

Vẽ điểm  $(x_{i+1}, y_{i+1})$  vừa tìm được.

❖ **Bước 3:** (Xác định điều kiện lặp)

Lặp lại bước 2 với những giá trị  $i$  tiếp theo, cho đến khi chúng ta vẽ được điểm nguyên gần nhất với C, nghĩa là:

$$x_{i+1} = \text{Round}(x_C) = \text{Round}\left(\frac{a^2}{\sqrt{a^2 + b^2}}\right) \text{ thì giải thuật kết thúc.}$$

**Chú ý:** Tóm tắt giải thuật trên chỉ áp dụng cho đoạn AC. Để dựng đoạn BC, chúng ta cần có sự thay đổi vai trò của của x và y cũng như a và b. Cụ thể, để dựng được cung BC cần hoán đổi trong toàn bộ giải thuật: x thành y và y ngược lại thành x, a thành b và b ngược lại thành a.

## 4.3. Cài đặt giải thuật

Sau đây là phần cài đặt minh họa cho giải thuật vẽ ellipse gồm có:

(1) Hàm Bre\_Ellipse thực hiện chức năng vẽ đường ellipse (không tô màu phần bên trong). (2) Hàm FillEllipse thực hiện chức năng vẽ hình ellipse có tô màu phần bên trong hình.

### **1. Hàm Bre\_Ellipse**

**Đầu vào:** Con trỏ ngũ cành thiết bị pDC, tọa độ tâm của ellipse (X\_center, Y\_center), bán kính theo trục x là a và bán kính theo trục y là b, màu sắc của đường ellipse là PenColor.

**Đầu ra:** Hình ảnh thể hiện của ellipse trên ngũ cành thiết bị.

```
void CLineDlg::Bre_Ellipse(CDC * pDC, int X_center, int Y_center,
int a, int b, COLORREF PenColor)
{
    if ((a <= 0) || (b <= 0))
        return;

    int x, y;
    int P;
    int Const1, Const2;
    Const1 = int(double(a*a) / sqrt(double(a*a + b*b)) + 0.5);
    Const2 = int(double(b*b) / sqrt(double(a*a + b*b)) + 0.5);

    // Vẽ cung AC
    x = 0; y = b; P = a*a*(1 - 2 * b) - 2 * b*b;

    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(x + X_center, -y + Y_center, PenColor);

    while (x < Const1)
    {
        if (P < 0)
            P += 2 * b*b*(2 * x + 3);
        else
        {
            P += 2 * b*b*(2 * x + 3) + 4 * a*a*(1 - y);
            if (y > Const2)
                y--;
        }
        x++;
    }

    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(x + X_center, -y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, -y + Y_center, PenColor);
}
```

```
// Vẽ cung BC
y = 0; x = a; P = b*b*(1 - 2 * a) - 2 * a*a;

pDC->SetPixel(x + X_center, y + Y_center, PenColor);
pDC->SetPixel(-x + X_center, y + Y_center, PenColor);

while (y < Const2)
{
    if (P < 0)
        P += 2 * a*a*(2 * y + 3);
    else
    {
        P += 2 * a*a*(2 * y + 3) + 4 * b*b*(1 - x);
        if (x > Const1)
            x--;
    }
    y++;

    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(x + X_center, -y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, -y + Y_center, PenColor);
}
}
```

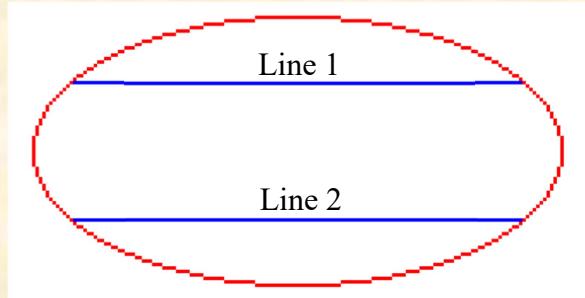
## 2. Hàm FillEllipse

Phần bên trong của ellipse được tiến hành tô theo kiểu quét dòng. Tại mỗi bước, chúng ta tính được điểm  $(x_i, y_i)$  theo giải thuật và xác định được thêm 3 điểm đối xứng của nó lần lượt là  $(-x_i, y_i), (x_i, -y_i), (-x_i, -y_i)$ . Từ đó, xác định được 2 dòng quét là:

Line 1: Từ điểm  $(-x_i, y_i)$  đến điểm  $(x_i, y_i)$ .

Line 2: Từ điểm  $(-x_i, -y_i)$  đến điểm  $(x_i, -y_i)$ .

Tiến hành tô màu cho Line 1 và Line 2 như Hình 1.20 dưới đây.



Hình 1.20. Minh họa kỹ thuật tô ellipse theo dòng quét

Tiến hành xây dựng hàm FillEllipse với thông tin vào ra:

**Đầu vào:** Con trỏ ngũ cẩm thiết bị pDC, tọa độ tâm của ellipse (X\_center, Y\_center), bán kính theo trục x là a và bán kính theo trục y là b, màu sắc của đường ellipse là PenColor và màu tô bên trong đường ellipse là BrushColor.

**Đầu ra:** Hình ảnh thể hiện của ellipse trên ngũ cẩm thiết bị.

```

void FillEllipse(CDC * pDC, int X_center, int Y_center, int a, int
b, COLORREF PenColor, COLORREF BrushColor)
{
    if ((a <= 0) || (b <= 0))
        return;

    CPen MyPen(PS_SOLID, 1, BrushColor);
    HGDIOBJ OldPen = pDC->SelectObject(MyPen);
    int x, y, Old_y;
    int P;
    int Const1, Const2;

    Const1 = int((double(a*a) / sqrt(double(a*a + b*b))) +0.5);
    Const2 = int((double(b*b) / sqrt(double(a*a + b*b))) +0.5);

    // Vẽ cung AC
    x = 0; y = b; P = a*a*(1 - 2 * b) - 2 * b*b;
    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
}

```

```

pDC->SetPixel(x + X_center, -y + Y_center, PenColor);
Old_y = y;

while (x < Const1)
{
    if (P < 0)
        P += 2 * b*b*(2 * x + 3);
    else
    {
        P += 2 * b*b*(2 * x + 3) + 4 * a*a*(1 - y);
        if (y > Const2)
            y--;
    }

    x++;

    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(x + X_center, -y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, -y + Y_center, PenColor);

    if (y != Old_y)
    {
        pDC->MoveTo(-x + 1 + X_center, y + Y_center);
        pDC->LineTo(x + X_center, y + Y_center);
        pDC->MoveTo(-x + 1 + X_center, -y + Y_center);
        pDC->LineTo(x + X_center, -y + Y_center);
        Old_y = y;
    }
}

// Vẽ cung BC

y = 0; x = a; P = b*b*(1 - 2 * a) - 2 * a*a;

pDC->SetPixel(x + X_center, y + Y_center, PenColor);
pDC->SetPixel(-x + X_center, y + Y_center, PenColor);

if (y != Old_y)
{
    pDC->MoveTo(-x + 1 + X_center, y + Y_center);
    pDC->LineTo(x + X_center, y + Y_center);
}

```

```
    old_y = y;
}

while (y < Const2)
{
    if (P < 0)
        P += 2 * a*a*(2 * y + 3);
    else
    {
        P += 2 * a*a*(2 * y + 3) + 4 * b*b*(1 - x);
        if (x > Const1)
            x--;
    }

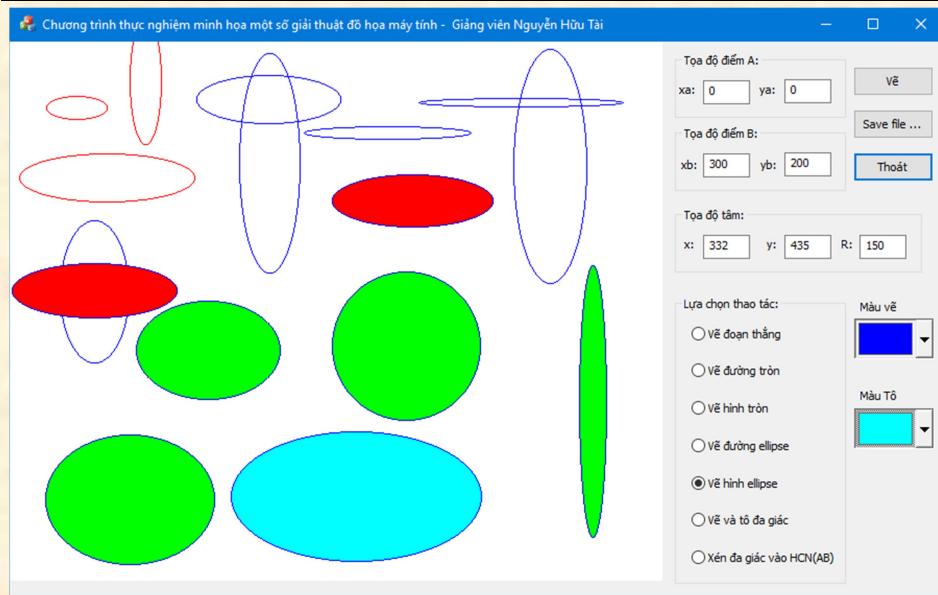
    y++;

    pDC->SetPixel(x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(x + X_center, -y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, y + Y_center, PenColor);
    pDC->SetPixel(-x + X_center, -y + Y_center, PenColor);

    if (x > 0)
    {
        pDC->MoveTo(-x + 1 + X_center, y + Y_center);
        pDC->LineTo(x + X_center, y + Y_center);
        pDC->MoveTo(-x + 1 + X_center, -y + Y_center);
        pDC->LineTo(x + X_center, -y + Y_center);
    }
}

pDC->SelectObject(OldPen);
}
```

Kết quả thực thi phần cài đặt trên được thể hiện qua Hình 1.21. dưới đây:



Hình 1.21. Hình ảnh thực nghiệm dựng đường ellipse và hình ellipse bởi giải thuật Bresenham

## 5. BÀI TẬP CUỐI CHƯƠNG

1. Cho điểm A(5, 7) và B(15, 15). Sử dụng giải thuật Bresenham để tìm tọa độ các điểm vẽ  $(x_i, y_i)$ .
2. Kế thừa dự án **LineDemo** trong **bài thực nghiệm số 1** đã được trình bày chi tiết tại **mục 2.3**, sinh viên cần xây dựng một hàm vẽ đoạn thẳng tổng quát cho phép vẽ đoạn thẳng AB trong mọi trường hợp hệ số góc.
3. Sử dụng giải thuật vẽ đường tròn Midpoint để tính giá trị các điểm vẽ  $(x_i, y_i)$  biết rằng  $R = 20$ .
4. Cài đặt một hàm vẽ đường tròn theo giải thuật MidPoint.
5. Cài đặt một hàm cho phép tô màu phần diện tích bên trong của đường tròn. Hàm có dạng **void FillCircle(int x, int y, int R, COLORREF FillColor)**.
6. Cài đặt hàm vẽ đường ellipse theo giải thuật Bresenham.

- 
- 7. Cài đặt một hàm tô màu phần bên trong của một Ellipse.
  - 8. Hãy xây dựng một giải thuật để dựng đường cong bậc 2 (Parabol) dạng tổng quát,  $y = ax^2 + bx + c$ , trên một đoạn xác định  $[x_1, x_2]$ .
  - 9. Viết chương trình vẽ đường Parabol.
  - 10. Xây dựng chương trình cho phép vẽ biểu đồ (Chart) từ số liệu đầu vào như trong chức năng vẽ đồ thị của chương trình Microsoft Excel.
  - 11. Hãy xây dựng một thư viện đồ họa riêng với các hàm vẽ các đường cơ bản do bạn tự viết dựa trên những kiến thức đã được lĩnh hội.

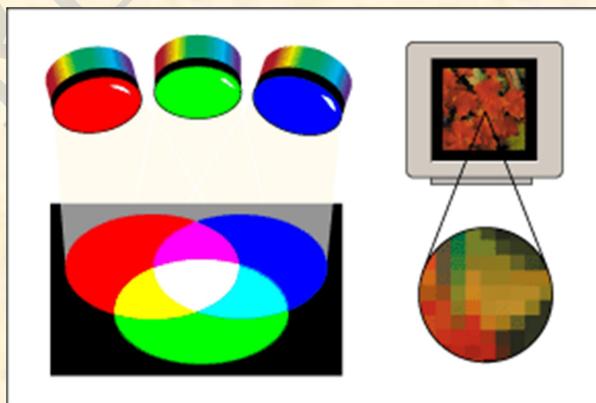
## Chương 2

# CÁC HỆ MÀU VÀ CƠ CHẾ TỔ CHỨC BỘ NHỚ MÀN HÌNH

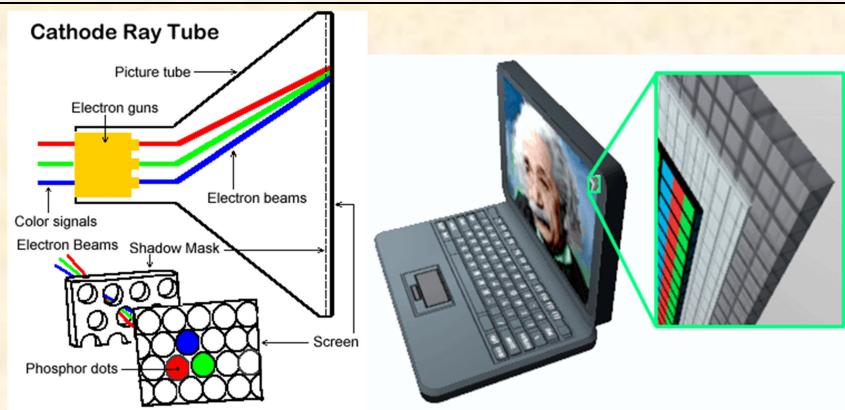
Giao thoa màu sắc là chìa khóa cho các kỹ thuật tái tạo hình ảnh màu trên các thiết bị đồ họa như màn hình, máy in. Trong chương này, chúng ta sẽ tìm hiểu qua về cấu trúc và cách thức tạo điểm ảnh màu trên màn hình, từ đó tìm hiểu về các hệ màu và giải quyết bài toán chuyển đổi giữa các hệ màu cũng như tính ứng dụng của mỗi hệ màu, và cuối cùng chúng ta sẽ tìm hiểu về cách thức tổ chức hoạt động của bộ nhớ màn hình thông qua một mode màn hình cụ thể và đơn giản đó là mode 13h.

### 1. ĐÔI NÉT VỀ CẤU TRÚC MÀN HÌNH MÀU

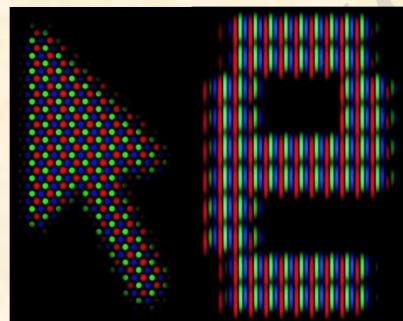
Màn hình màu như máy tính hay tivi về cơ bản được cấu tạo từ một ma trận các điểm ảnh có khả năng hiển thị với màu sắc thay đổi theo yêu cầu. Sở dĩ có được khả năng này là bởi tính chất giao thoa màu sắc của ánh sáng (xem Hình 2.1) và mỗi điểm ảnh được cấu tạo từ 3 điểm ảnh đơn sắc như Hình 2.2. Mắt người với đặc điểm độ nhạy kém, nên với thiết kế các điểm đơn sắc đủ bé và khoảng cách quan sát đủ xa thì mắt người sẽ không thể nhìn thấy hình ảnh 3 điểm đơn sắc riêng biệt mà chúng bị nhòa làm một và cho hình ảnh một màu tổng hợp theo nguyên lý giao thoa.



Hình 2.1. Màu sắc và sự giao thoa



Hình 2.2. Hai loại cấu trúc màn hình màu: (a) CRT, (b) LCD



Hình 2.3. Ảnh biểu diễn của một mũi tên màu trắng, và một chữ E trên máy tính được phóng lớn tương ứng với hai loại màn hình CRT và LCD

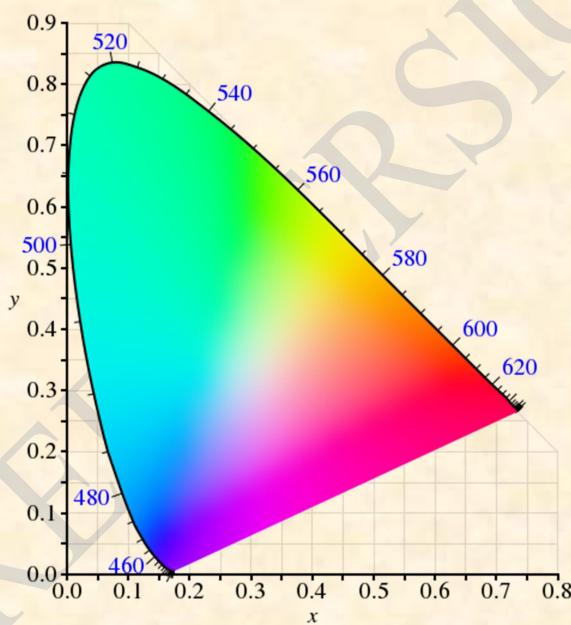
## 2. CÁC HỆ MÀU

Trên các thiết bị hiển thị như màn hình máy tính, màn hình tivi và phần lớn các thiết bị hiển thị màu thông dụng khác người ta chọn 3 màu Red, Green và Blue để biểu diễn tất cả các màu sắc khác nhau của hình ảnh, cũng như người họa sĩ chỉ dùng một số màu cơ bản, song các bức tranh được vẽ ra lại rất phong phú về màu sắc. Một câu hỏi đặt ra là tại sao lại chọn 3 màu trên mà không phải là một nhóm màu nào khác. Để trả lời câu hỏi này, chúng ta hãy tìm hiểu qua về cách tạo mắt người.

Mắt con người chúng ta cảm nhận màu sắc thông qua các tế bào võng mạc hình nón. Ba màu Red, Green và Blue được mắt con người cảm nhận rõ nhất, chúng có bước sóng dài lần lượt là 580nm, 545nm và 440nm. Sự hòa trộn của 3 bước sóng này sẽ cho chúng ta được những màu sắc khác

nhau. Năm 1931, Commission internationale de l'éclairage, được gọi tắt là CIE, đã định nghĩa mối liên hệ giữa màu sắc vật lý trong dải phổ điện từ mà mắt người nhìn thấy được (dựa trên bước sóng) với màu sắc được cảm nhận sinh lý trong trường thị giác người. Những ràng buộc toán học định nghĩa không gian màu là những công cụ thiết yếu cho việc quản lý màu sắc. Chúng cho phép chuyển đổi qua lại giữa các hình thức thể hiện màu sắc qua in ấn, màn hình, hay các thiết bị ghi hình. Trên cơ sở đó, hình thành nên chuẩn không gian màu RGB (tài liệu tham khảo số 6).

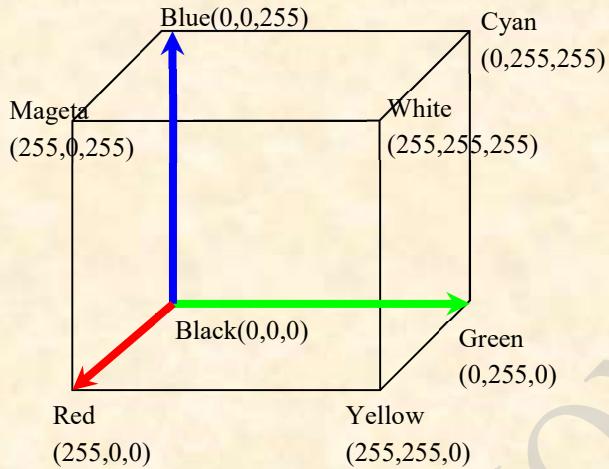
Hiện nay, không phải chỉ có hệ màu RGB mà còn có những hệ khác như: CMY, HSV, HSL, YCbCr, Lab,... Phản tiếp theo sẽ trình bày một số hệ màu cơ bản và thông dụng.



Hình 2.4. Biểu đồ thể hiện các sắc độ  
trong không gian màu chuẩn CIE 1931

## 2.1. Hệ RGB

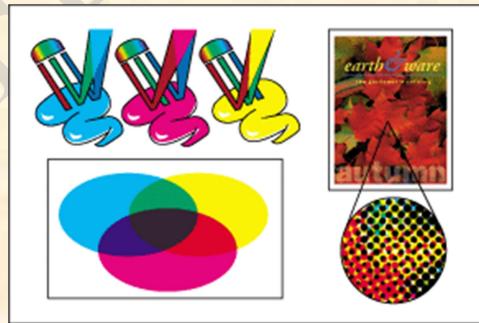
RGB là chữ viết tắt của 3 từ Red, Green và Blue. Hệ này có miền không gian giá trị là một khối 3 chiều. Mỗi màu xác định trên 3 thành phần là R, G, B. Sự giao nhau của các thành phần này sẽ tạo ra các màu sắc khác nhau tạo nên một không gian màu.



Hình 2.5. Không gian màu trong chế độ 24-bit

Cường độ của mỗi thành phần R,G,B được mã hóa trong các mức khác nhau. Có các mức mã hóa khá phổ biến là: Mã hóa 16, 64 và 256 mức. Hiện nay, mức mã hóa 256 mức là phổ biến (từ 0 đến 255). Nếu cường độ của mỗi thành phần được mã hóa trong 256 mức thì cần 8 bit để mã hóa, vậy một màu biểu diễn bởi 3 thành phần đơn sắc RGB sẽ lưu trữ bởi 24 bit, chế độ này thường được gọi là chế độ màu thực (True color - 24 bit), bởi vì nó có thể biểu diễn đến khoảng 16,7 triệu màu.

## 2.2. Hệ màu CMY



Hình 2.6. Hệ màu CMYK chuyên dùng trong in ấn

CMY là 3 chữ viết tắt từ: Cyan (màu lục lam), Magenta (màu đỏ tươi), Yellow (màu vàng). Công thức chuyển đổi từ hệ RGB sang hệ CMY như sau:

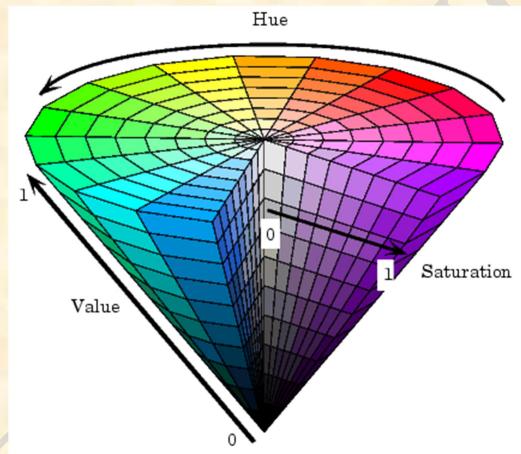
$$C = 1 - R$$

$$M = 1 - G$$

$$Y = 1 - B$$

Hệ màu này thường được sử dụng trong in ấn. Tuy nhiên, để tiết kiệm mực màu do chúng thường có giá thành đắt hơn mực đen, người ta dùng mực đen để hiển thị các điểm ảnh màu xám thay cho việc hòa trộn các màu CMY để tạo ra điểm màu xám, từ đó chúng ta có hệ màu CMYK để chỉ hệ màu CMY có sử dụng thêm màu đen trong các hệ thống in ấn.

### 2.3. Hệ màu HSV

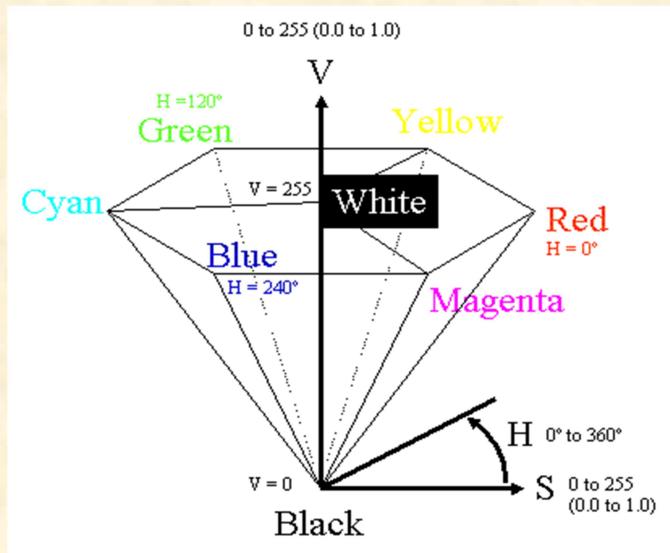


Hình 2.7. Hệ màu HSV biểu diễn trong ché độ số thực (từ 0 đến 1)

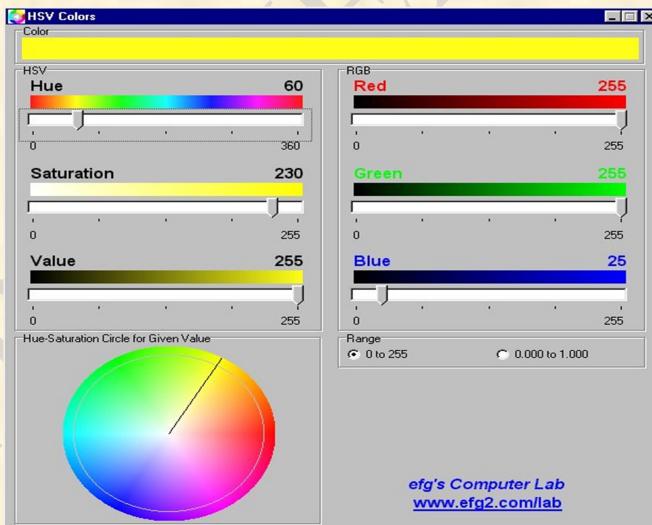
Hệ màu HSV được viết tắt từ 3 chữ sau: Hue (màu sắc), Saturation (sự bão hòa) và Value (giá trị, thể hiện mức độ sáng tối). Ba màu Red, Green và Blue trong hệ màu RGB có lẽ là 3 màu mà mắt con người cảm nhận ánh sáng chứ không phải là màu mà mắt con người cảm nhận màu sắc. Màu sắc (Hue, hay color) được đo bởi tần số ánh sáng, còn độ sáng được đo bởi cường độ. Màu càng sáng thì cường độ càng lớn. Hình 2.7 minh họa hệ màu HSV được biểu diễn trong không gian số thực với Hue có giá trị từ  $0 \rightarrow 360$ , trong khi S và V có giá trị trong khoảng  $0 \rightarrow 1$ .

Người ta cũng có thể áp dụng việc lượng tử hóa các giá trị S và V về không gian số nguyên với giá trị giao động trong khoảng  $0 \rightarrow 255$  như

Hình 2.8, trong đó: Màu đỏ hoàn toàn được biểu diễn là (0 độ, 255, 255), màu xanh (green) được biểu diễn là (120 độ, 255, 255).



Hình 2.8. Hệ màu HSV biểu diễn trong ché độ lượng hóa nguyên



Hình 2.9. Minh họa việc chuyển đổi qua lại giữa 2 hệ màu HSV và RGB

Trong nhiều tình huống áp dụng không đòi hỏi độ chính xác cao, thay vì biểu diễn thành phần Hue trong miền giá trị từ 0 → 360, người ta lượng tử hóa nó vào một thang giá trị nguyên từ 0 đến 255, đưa đến khả năng biểu diễn và lưu trữ giá trị Hue trong 1 byte.

Dưới đây là các hàm chuyển đổi giá trị điểm ảnh từ không gian màu RGB sang HSV và ngược lại. Trong đó, các thành phần R, G, B, H, S, V đều được biểu diễn dưới dạng số nguyên 1 byte với miền giá trị biến thiên từ 0 đến 255.

```

typedef struct RgbColor
{
    BYTE b;
    BYTE g;
    BYTE r;
};

typedef struct HsvColor
{
    BYTE h;
    BYTE s;
    BYTE v;
};

void RgbToHsv(RgbColor rgb, HsvColor &hsv)
{
    BYTE rgbMin, rgbMax;

    rgbMin = rgb.r < rgb.g ? (rgb.r < rgb.b ? rgb.r : rgb.b) : (rgb.g
    < rgb.b ? rgb.g : rgb.b);
    rgbMax = rgb.r > rgb.g ? (rgb.r > rgb.b ? rgb.r : rgb.b) : (rgb.g
    > rgb.b ? rgb.g : rgb.b);

    hsv.v = rgbMax;
    if (hsv.v == 0)
    {
        hsv.h = 0;
        hsv.s = 0;
        return;
    }

    hsv.s = 255 * long(rgbMax - rgbMin) / hsv.v;
    if (hsv.s == 0)
    {
        hsv.h = 0;
        return;
    }
}

```

```

if (rgbMax == rgb.r)
    hsv.h = 0 + 43 * (rgb.g - rgb.b) / (rgbMax - rgbMin);
else if (rgbMax == rgb.g)
    hsv.h = 85 + 43 * (rgb.b - rgb.r) / (rgbMax - rgbMin);
else
    hsv.h = 171 + 43 * (rgb.r - rgb.g) / (rgbMax - rgbMin);
}

void HsvToRgb(HsvColor hsv, RgbColor &rgb)
{
    BYTE region, remainder, p, q, t;

    if (hsv.s == 0)
    {
        rgb.r = hsv.v;
        rgb.g = hsv.v;
        rgb.b = hsv.v;
        return;
    }

    region = hsv.h / 43;
    remainder = (hsv.h - (region * 43)) * 6;

    p = (hsv.v * (255 - hsv.s)) >> 8;
    q = (hsv.v * (255 - ((hsv.s * remainder) >> 8))) >> 8;
    t = (hsv.v * (255 - ((hsv.s * (255 - remainder)) >> 8))) >> 8;

    switch (region)
    {
        case 0:
            rgb.r = hsv.v; rgb.g = t; rgb.b = p;
            break;
        case 1:
            rgb.r = q; rgb.g = hsv.v; rgb.b = p;
            break;
        case 2:
            rgb.r = p; rgb.g = hsv.v; rgb.b = t;
            break;
        case 3:
            rgb.r = p; rgb.g = q; rgb.b = hsv.v;
            break;
        case 4:
            rgb.r = t; rgb.g = p; rgb.b = hsv.v;
    }
}

```

```

        break;
    default:
        rgb.r = hsv.v; rgb.g = p; rgb.b = q;
        break;
    }
}

```

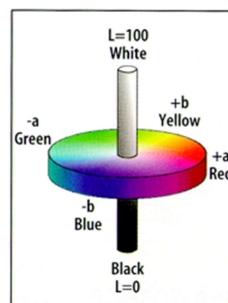
Các hệ màu HSL và HSI là một dạng biểu diễn khác của hệ màu HSV trong đó giá trị V được thay bởi I (Intensity: Cường độ) hay L (Lightness: Độ sáng). Công thức tính I hay L từ các giá trị R, G, và B được cho bởi:

$$I = (R + G + B)/3$$

$$L = (\min(R,G,B) + \max(R,G,B))/2$$



Hình 2.10. Minh họa hệ màu HSL



Hình 2.11. Minh họa hệ màu Lab

### 3. CƠ CHẾ TỔ CHỨC BỘ NHỚ MÀN HÌNH

Địa chỉ bộ nhớ logic dùng cho màn hình trong hệ điều hành MS-DOS được bắt đầu từ:

B800:0000 với chế độ text.

A000:0000 với các chế độ đồ họa.

Trong phần này chúng ta chỉ đề cập đến chế độ đồ họa.

Thông tin hiển thị trên màn hình được bố trí trong bộ nhớ bắt đầu từ địa chỉ logic A000:0000 đến A000:FFFF với kích thước là 64KB, song cách bố trí như thế nào lại phụ thuộc vào chế độ (mode) màn hình mà chúng ta chọn để hoạt động. Thông thường, một card điều khiển đồ họa có thể hoạt động trong nhiều mode khác nhau. Để ra chỉ thị cho card hoạt động theo một mode nào đó, chúng ta cần triệu gọi các hàm phục vụ đặt chế độ màn hình của Bios, và truyền tham số mode cho hàm.

Do có rất nhiều mode đồ họa khác nhau và mỗi mode lại có một cơ chế bố trí thông tin riêng. Trong phần này chỉ trình bày:

- Một mode khá thông dụng, được dùng chủ yếu trong các trò chơi (games) chạy trên hệ điều hành MS-DOS, đó là mode 13h (Hexa). Đây là mode với độ phân giải là  $320 \times 200$  với 256 màu, trong chế độ này một byte trong vùng nhớ màn hình lưu trữ thông tin của một điểm, hay nói cách khác là có sự tương ứng một-một giữa điểm ảnh và byte nhớ trong vùng nhớ màn hình. Mặt khác, kích thước bộ nhớ đủ để lưu trữ toàn bộ các điểm ảnh trên màn hình là  $320 \times 200 \times 1$  byte = 64.000 byte < 64 KB, kích thước này vừa đủ để nằm gọn trong một phân đoạn bộ nhớ (memory segmentation) A000. Vì thế, các thao tác xử lý điểm ảnh trong chế độ này khá đơn giản và nhanh chóng.
- Chuẩn Vesa (Video Electronics Standards Association): Hầu như tất cả các Card điều khiển màn hình thông dụng đều hỗ trợ mode này. Với mode bạn có thể đặt độ phân giải từ  $640 \times 400$ ,  $640 \times 480, \dots$  lên đến  $1024 \times 800$  hay cao hơn nữa tùy vào khả năng kỹ thuật của Card. Tương tự màu sắc có thể từ 16 màu, 256 màu, High color -16 bit hay True color - 24 bit hay 32 bit.

### 3.1. Cơ chế hoạt động của chế độ màn hình độ phân giải $320 \times 200$ với 256 màu

Trong chế độ này, màn hình được chia ra 320 cột và 200 hàng tạo nên  $320 \times 200$  điểm ảnh (pixel). Tuy độ phân giải thấp, song nó lại có ưu điểm là truy cập nhanh chóng, nhờ cơ chế bộ nhớ đơn giản. Pixel[cột 0, dòng 0] (ở góc trên bên trái màn hình) tương ứng với byte

nhớ có địa chỉ A000:0000, tương tự Pixel[cột 1, dòng 0] tương ứng với byte nhớ có địa chỉ A000:0001,...

Hay nói một cách tổng quát, pixel tại cột x dòng y tương ứng với byte nhớ [A000 : (y \* 320 + x)]

Lệnh sau cho phép vẽ màu có số 255 cho điểm có tọa độ (x, y):

Mem[\$A000 : (y \* 320 + x)] = 255

Câu lệnh trên cho phép vẽ ra màn hình 1 điểm có màu 255 tại vị trí (x, y). Tuy nhiên, màu có số 255 là màu như thế nào lại phụ thuộc vào thanh ghi màu số 255 lưu trữ các giá trị RGB mô tả cách hòa ra màu 255 như thế nào.

Để quyết định màu cho một thanh ghi màu, chúng ta cần đến các hàm phục vụ của Bios để đặt giá trị cho thanh ghi màu.

### 3.2. Cơ chế hoạt động của màn hình theo chuẩn Vesa

Video Electronics Standards Association (VESA) là một tổ chức tiêu chuẩn kỹ thuật cho chuẩn hiển thị trên máy tính. Chuyên phát triển các chuẩn giao diện kỹ thuật số cho các thiết bị hiển thị (màn hình, tivi,...). Hình 2.12 thể hiện các thông tin chi tiết về một số mode màn hình đồ họa máy tính theo chuẩn Vesa, công bố năm 1998. Các tài liệu công bố về các mode hiện đại hơn gần đây đòi hỏi người dùng phải trả phí thành viên mới được cung cấp.

VESA cung cấp nhiều mode đồ họa cao cấp, ví dụ mode 11Bh cho độ phân giải  $1280 \times 1024$ , với khả năng biểu diễn được khoảng 16.8 triệu màu do sử dụng 24 bit để biểu diễn thông tin một điểm ảnh trong không gian màu RGB. Bộ nhớ cần thiết để lưu trữ  $1280 \times 1024$  điểm ảnh (hay một trang màn hình) là:  **$1280 \times 1024 \times 3 \text{ byte} = 3.75 \text{ MB}$** .

Trong nền tảng hệ điều hành 16-bit MS-DOS, việc truy xuất bộ nhớ màn hình để thay đổi giá trị các điểm ảnh trên màn hình thông qua địa chỉ logic có segment mặc định A000 là khá phức tạp. Cụ thể, do hệ điều hành MS-DOS chỉ cung cấp không gian địa chỉ logic từ A000:0000 đến A000:FFFF với kích thước là 64 KB như là một cửa sổ để định vị truy xuất đến bộ nhớ RAM màn hình, trong khi kích thước bộ nhớ chúng ta cần truy xuất lên đến 3.75 MB. Để giải quyết vấn đề này, người ta chia

bộ nhớ RAM màn hình thành nhiều dải, thuật ngữ là band, mỗi band có kích thước 64 KB vừa đúng với kích thước của cửa sổ truy xuất A000:0000 đến A000:FFFF mà hệ điều hành cung cấp. Từ đó, để truy xuất thông tin của một điểm ảnh M(x,y) lưu trữ trong bộ nhớ RAM màn hình, chúng ta cần tính toán để biết được vùng nhớ của điểm ảnh M thuộc band nào và có địa chỉ offset trên band là bao nhiêu?

GRAPHICS				TEXT			
15-bit mode	7-bit mode	Resolution	Colors	15-bit mode	7-bit mode	Columns	Rows
number	number			number	number		
100h	-	640x400	256	108h	-	80	60
101h	-	640x480	256	109h	-	132	25
102h	6Ah	800x600	16	10Ah	-	132	43
103h	-	800x600	256	10Bh	-	132	50
104h	-	1024x768	16	10Ch	-	132	60
105h	-	1024x768	256				
106h	-	1280x1024	16				
107h	-	1280x1024	256				
10Dh	-	320x200	32K (1:5:5:5)				
10Eh	-	320x200	64K (5:6:5)				
10Fh	-	320x200	16.8M (8:8:8)				
110h	-	640x480	32K (1:5:5:5)				
111h	-	640x480	64K (5:6:5)				
112h	-	640x480	16.8M (8:8:8)				
113h	-	800x600	32K (1:5:5:5)				
114h	-	800x600	64K (5:6:5)				
115h	-	800x600	16.8M (8:8:8)				
116h	-	1024x768	32K (1:5:5:5)				
117h	-	1024x768	64K (5:6:5)				
118h	-	1024x768	16.8M (8:8:8)				
119h	-	1280x1024	32K (1:5:5:5)				
11Ah	-	1280x1024	64K (5:6:5)				
11Bh	-	1280x1024	16.8M (8:8:8)				
81FFh	Special Mode (see below for details)						

Hình 2.12. Một số mode màn hình cùng thông tin chi tiết về độ phân giải và số màu có thể hiển thị, số bit phân phối cho 3 thành phần màu RGB của một điểm ảnh

Các bước tính toán địa chỉ truy xuất cho một điểm ảnh M(x,y) gồm:

- Xác định số byte để lưu trữ một dòng ảnh, thường bằng số điểm ảnh trên một dòng nhân với số byte dữ liệu cho một điểm ảnh, kết quả phải được làm tròn lên số nguyên là bội số của DWORD (hay 4 byte):

$$\text{Byte\_per\_line} = \text{int}((1280 \times 3\text{byte} + 3) / 4) \times 4 = 3840$$

- Tính địa chỉ thực:

$$\text{RealAdd} = y \times \text{Byte\_per\_line} + x \times 3$$

- Tính số hiệu band bộ nhớ:

$$\text{Band\_Index} = \text{int}(\text{RealAdd} / 65536)$$

- Tính địa chỉ offset:

$$\text{Offset} = \text{RealAdd} - (\text{Band\_Index} \times 65536)$$

Khi đã biết được số band và địa chỉ offset, chúng ta cần tiến hành các bước sau để có thể truy xuất thông tin của điểm ảnh M trên bộ nhớ:

- Xác định giá trị Current\_Band, là band hiện thời đang được ánh xạ đến cửa sổ truy xuất bộ nhớ màn hình (hay ánh xạ đến vùng địa chỉ logic A000:0000 đến A000:FFFF).
- Nếu ( $\text{Band\_Index} \neq \text{Current\_Band}$ ): Gọi dịch vụ Bios (ngắt 10h hàm phục vụ số 4f05H) để ánh xạ band có số hiệu Band\_Index đến địa chỉ nhớ logic A000:0000.
- Truy xuất đến vùng dữ liệu bộ nhớ của điểm M thông qua địa chỉ logic A000: Offset.

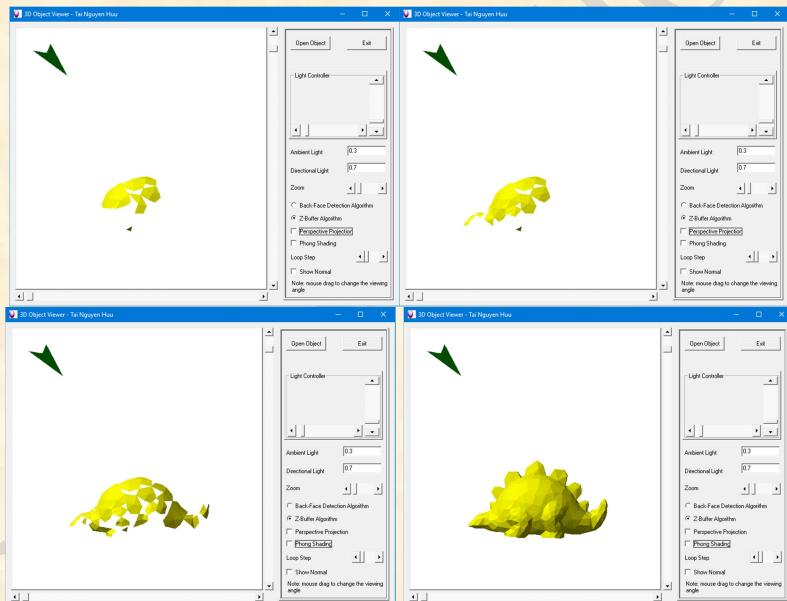
Chi tiết về cách thức hoạt động của các mode đồ họa theo chuẩn Vesa trong hệ điều hành MS-DOS được trình bày cụ thể trong *Tài liệu tham khảo số 5*.

#### **4. KỸ THUẬT THỰC HIỆN VẼ ĐỒ HỌA Ở HẬU TRƯỜNG (OFF-SCREEN RENDERING)**

Trong thực tế, rất nhiều bài toán ứng dụng đòi hỏi phải giải quyết tốt các vấn đề như: Tốc độ thực hiện các thao tác đồ họa, hình ảnh hiển thị một cách hợp lý (hay “chuyên nghiệp”) mà điển hình trong số này là các ứng dụng đồ họa trò chơi 2 hay 3 chiều và các ứng dụng mô phỏng thực tại ảo.

Vì vùng bộ nhớ ánh xạ đến màn hình, hay còn được biết đến như là VIDEO RAM, là tài nguyên dùng chung cho mọi chương trình đang chạy trên hệ thống máy tính có nhu cầu hiển thị thông tin lên màn hình. Đồng thời, cứ sau một chu kỳ, 1/60 giây hay 1/120 giây tùy theo thiết kế hay

thiết lập cấu hình của màn hình, thì hệ thống màn hình sẽ truy xuất vào vùng nhớ này để lấy thông tin và hiển thị thành hình ảnh tương ứng trên màn hình. Từ đó dễ thấy rằng, việc truy xuất và thực hiện nhiều thao tác đồ họa phức tạp trên vùng bộ nhớ đồ họa dành riêng cho màn hình sẽ gây ra nhiều trở ngại cũng như khó được đáp ứng về mặt tốc độ. Minh họa điển hình cho tình trạng tốc độ thực hiện trì trệ được đánh giá và mô tả chi tiết qua “Bài thực nghiệm số 2” (trang 86) khi dùng đối tượng quản lý ngữ cảnh thiết bị (Device-Context Object) để truy xuất đến vùng bộ nhớ màn hình thông qua các hàm SetPixel (vẽ điểm ảnh) và GetPixel (lấy giá trị màu của điểm ảnh) thực hiện bài toán tô màu theo giải thuật vết dầu loang cho một vùng ảnh thuộc màn hình.



Hình 2.13. Minh họa tình huống tiêu cực khi thực hiện đồ họa trực tiếp trên vùng bộ nhớ dành riêng cho màn hình với các ứng dụng đồ họa đòi hỏi nhiều thời gian xử lý. Người sử dụng có thể quan sát thấy một chuỗi các hình ảnh đang trong quá trình xây dựng, thay vì chỉ một hình ảnh hoàn thiện như mong muốn

Tốc độ thực hiện chậm sẽ dẫn đến trình trạng hình ảnh mong muốn được tạo ra và hiển thị từng phần theo thời gian trên màn hình như minh họa trong Hình 2.13. Điều này là không thể chấp nhận được trong nhiều

ứng dụng, điển hình nhất là các ứng dụng trò chơi dùng đồ họa mô phỏng 2 hay 3 chiều, người chơi không thể chấp nhận tình trạng từng phần hình ảnh được vẽ ra lần lượt theo thời gian và chậm chạp, thay vì các hình ảnh hoàn thiện được xuất hiện một cách nhanh chóng và đáp ứng yêu cầu về thời gian thực (real time).

Để đẩy nhanh tốc độ thực hiện và tránh tình trạng người sử dụng quan sát được các hình ảnh chưa hoàn thiện (đang trong quá trình hình thành), hầu hết các hệ thống đồ họa đều cung cấp một kỹ thuật đồ họa gián tiếp, còn được gọi là kỹ thuật Off-screen Rendering hay vẽ đồ họa ở hậu trường, khi hình ảnh đồ họa được tạo nên bởi các chi tiết đồ họa nói chung (như điểm ảnh, các đường,...) trên một vùng bộ nhớ riêng được cấp phát ngoài vùng bộ nhớ RAM ánh xạ đến màn hình (hay RAM video). Nếu chúng ta muốn xây dựng một hệ thống đồ họa có thể hỗ trợ kỹ thuật Off-screen, thì các hàm xử lý đồ họa cần phải được xây dựng mềm dẻo sao cho chúng có thể thực hiện được nhiệm vụ đồng thời trên cả vùng bộ nhớ RAM dành cho màn hình (cần tuân thủ các quy tắc giao tiếp với bộ nhớ RAM của màn hình, một dạng giao tiếp với thiết bị ngoại vi) và vùng bộ nhớ RAM thông thường được cấp phát làm Off-screen.

Trong môi trường lập trình Visual Studio C++, sử dụng MFC để xây dựng các chương trình (hay ứng dụng) chạy trên nền tảng Windows, tất cả các hoạt động đồ họa được thực hiện gián tiếp thông qua đối tượng quản lý ngữ cảnh thiết bị có tên gọi là Device-Context Object. *Thông tin chi tiết về Device-Context cần tham khảo trong tài liệu tham khảo số 10.* Các hàm đồ họa trên DC được xây dựng sao cho có thể hoạt động một cách độc lập với các thiết bị đồ họa (Màn hình, máy in, bộ nhớ,...) mà nó kết nối đến để tạo môi trường thuận lợi cho người lập trình. Kỹ thuật Off-screen có thể được thực hiện nhiều cách thông qua đối tượng DC, trong giáo trình này chúng ta sẽ xem xét đến 2 cách ở hai mức độ khác nhau tùy theo nhu cầu thực tiễn.

- ❖ Cách 1: Sử dụng lớp CMemDC (“Memory Device-Context” Class), là lớp dẫn xuất (hay kế thừa) từ lớp CDC, để tạo nên đối tượng dạng MemDC rồi thực hiện tất cả các thao tác đồ họa thông qua MemDC thay cho đối tượng DC như thông thường. Khi hoàn thành các tác vụ

vẽ đồ họa chúng ta sẽ có được hình ảnh hoàn thiện trong phần bộ nhớ gắn kết với MemDC nhưng hình ảnh này vẫn chưa xuất hiện trên màn hình, bước tiếp theo chúng ta cần gọi hàm hủy đối tượng có dạng ~CMemDC() để hệ thống tiến hành sao chép hình ảnh đã được tạo trong phần bộ nhớ cấp phát cho MemDC lên phần bộ nhớ màn hình để chúng xuất hiện trên màn hình và đồng thời đối tượng MemDC cũng được hủy. Minh họa cho cách thực hiện này được trình bày trong “**Bài thực nghiệm số 4**” (trang 172) phần nâng cấp.

Các bước xử lý Off-screen với đối tượng thuộc lớp MemDC:

1. Khởi tạo đối tượng thông qua hàm:

`CMemDC::CMemDC(CDC& dc, const CRect& rect)`

Với đầu vào:

+ dc: Device Context của cửa sổ nơi sẽ thể hiện hình ảnh hoàn thiện sau quá trình thực hiện các thao tác vẽ đồ họa.

+ rect: Xác định thông tin tọa độ của hình chữ nhật trên cửa sổ cần tạo Off-screen và cập nhật hình ảnh sau khi hoàn thành. Nếu chúng ta cần tạo Off-screen và cập nhật hình ảnh trên toàn bộ vùng bên trong cửa sổ (ClientArea) thì có thể dùng hàm GetClientRect(rect) để lấy thông tin, hoặc dùng một dạng khác của hàm khởi tạo đối tượng là `CMemDC(CDC& dc, CWnd* pWnd)`.

2. Gọi hàm `BOOL CMemDC::IsMemDC() const;` để biết quá trình cấp phát bộ nhớ và tạo MemDC có thành công hay không, nếu hàm không thành công thì các thao tác đồ họa với MemDC sau đó sẽ được thực hiện trên dc gốc mà chúng ta đã truyền vào ở giai đoạn khởi tạo đối tượng MemDC, nghĩa là nó sẽ thực hiện trên “screen DC” thay vì “Memory DC” như mong muốn.
3. Lấy dc của đối tượng MemDC thông qua hàm: `CDC& GetDC()`.
4. Thực hiện các thao tác vẽ đồ họa trên dc của MemDC.
5. Cập nhật hình ảnh đã hoàn thiện trên MemDC lên cửa sổ và sau đó hủy đối tượng thông qua hàm: `CMemDC::~CMemDC()`.

Rõ ràng, MemDC chỉ mang tính tạm thời, nó sẽ bị xóa ngay sau khi hình ảnh được cập nhật lên cửa sổ thông qua hàm hủy.

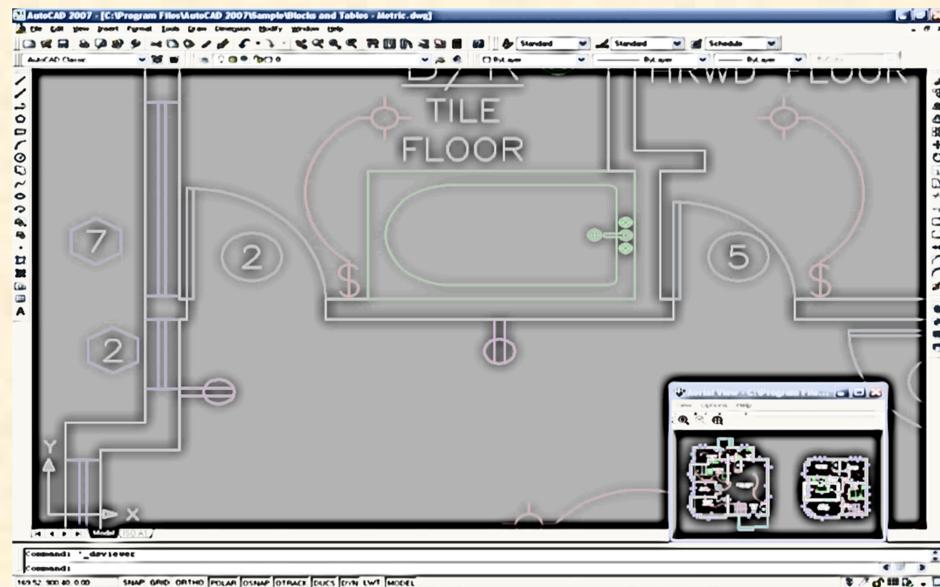
- ❖ Cách 2: Khởi tạo một DC mới và một đối tượng ảnh bitmap dạng DIB, là một định dạng ảnh bitmap mà ứng dụng có thể truy cập (write/read) một cách trực tiếp vào vùng bộ nhớ cấp phát cho DIB, rồi gắn DIB vào DC thông qua hàm SelectObject là chúng ta đã có một DC dạng MemDC.

Các thao tác vẽ đồ họa với DC đã được liên kết với DIB sẽ được thực thi trên vùng nhớ được cấp phát cho DIB chứ không phải trên vùng nhớ RAM liên kết với màn hình. Mặt khác, chúng ta cũng có thể truy cập trực tiếp vào vùng bộ nhớ cấp phát cho DIB để thực hiện những xử lý đồ họa chuyên biệt chứ không cần thông qua các hàm đồ họa được xây dựng sẵn (như SetPixel, GetPixel, LineTo,...) của Device Context, bằng cách này chúng ta có thể đẩy nhanh các thao tác xử lý đồ họa chuyên biệt mà chúng ta muốn xây dựng. “**Bài thực nghiệm số 2**” (trang 86) phần nâng cao, sẽ trình bày chi tiết về kỹ thuật Off-Screen thông qua DIB cùng các đánh giá về tốc độ thực thi các thao tác đồ họa cơ bản (SetPixel, GetPixel).

## Chương 3

# CÁC PHÉP XÉN HÌNH VÀ TÔ MÀU

Trong kỹ thuật đồ họa máy tính rất nhiều bài toán đòi hỏi phải sử dụng các giải thuật xén hình (clipping), nhằm cắt bỏ bớt các bộ phận của hình ảnh không nằm trong một phạm vi cho trước nào đó.



Hình 3.1. Minh họa kỹ thuật Clipping trong phần mềm AutoCad

Ví dụ như một bản thiết kế lớn cần xén vào một khung nhìn (viewport) cụ thể khi chúng ta thiết kế các chi tiết. Điều này giúp cho việc thể hiện bản vẽ được nhanh và hiệu quả.

Chúng ta có thể định nghĩa một cách tổng quát như sau:

Định nghĩa:

Cho một miền  $D \subset \mathbb{R}^n$  và  $F$  là một hình trong  $\mathbb{R}^n$  (nghĩa là  $F \subset \mathbb{R}^n$ ). Chúng ta gọi  $F \cap D$  là hình có được từ  $F$  bằng cách cắt nó vào trong  $D$ . Ký hiệu  $F \cap D$  là  $Clip_D(F)$ .

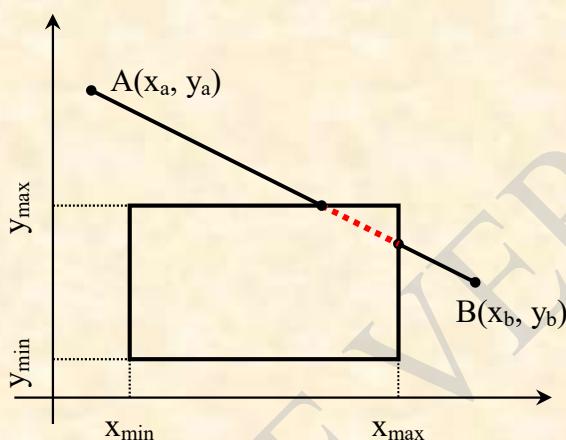
Trong từng ứng dụng cụ thể thì miền  $D$  và  $F$  sẽ được mô tả chính xác, chẳng hạn miền  $D$  là một vùng hình chữ nhật trong  $\mathbb{R}^2$  và  $F$  là một đoạn thẳng hay một đường tròn trong  $\mathbb{R}^2$ .

## 1. TRƯỜNG HỢP HÌNH F LÀ MỘT TẬP HỮU HẠN ĐIỂM

Trong trường hợp này, bài toán tương đương với việc tìm một giải thuật để xác định một điểm của F có nằm trong miền D hay không. Bài toán này đã được giải quyết khi D là một đa giác không tự cắt trong  $R^2$ , khi D là một đa diện không tự cắt trong  $R^3$ , hay khi D là hình tròn hoặc hình cầu,...

## 2. TRƯỜNG HỢP XÉN MỘT ĐOẠN THẲNG VÀO MỘT VÙNG HÌNH CHỮ NHẬT TRONG KHÔNG GIAN 2 CHIỀU

### 2.1. Khi cạnh của hình chữ nhật song song với trục tọa độ



Hình 3.2. Minh họa việc xén đoạn thẳng AB vào hình chữ nhật

Hình chữ nhật D và đoạn thẳng F có thể được biểu diễn như sau:

$$D = \left\{ (x, y) \in R^2 \middle| \begin{array}{l} x_{\min} \leq x \leq x_{\max} \\ y_{\min} \leq y \leq y_{\max} \end{array} \right\}; F = \left\{ (x, y) \in R^2 \middle| \begin{array}{l} x = x_A + (x_B - x_A)t \\ y = y_A + (y_B - y_A)t \\ 0 \leq t \leq 1 \end{array} \right\}$$

Trong đó,  $(x_{\min}, y_{\min})$  và  $(x_{\max}, y_{\max})$  là hai đỉnh xác định nên hình chữ nhật, còn  $(x_A, y_A)$  và  $(x_B, y_B)$  là tọa độ hai đầu mút của đoạn thẳng F kí hiệu là AB.

Trong trường hợp này  $\text{Clip}_D(F)$  có thể là rỗng hay là một đoạn thẳng (có thể suy biến thành 1 điểm). Theo định nghĩa thì giải thuật có thể mô tả như sau:

❖ **Bước 1:** Tìm nghiệm của hệ bất phương trình

$$\begin{cases} x_{\min} \leq x_A + (x_B - x_A)t \leq x_{\max} \\ y_{\min} \leq y_A + (y_B - y_A)t \leq y_{\max} \\ 0 \leq t \leq 1 \end{cases} \quad (3.1)$$

Nghiệm của hệ này sẽ là  $[t_1, t_2]$  hay  $\emptyset$ , chúng ta gọi  $N$  là tập các nghiệm của hệ

❖ **Bước 2:**

- Nếu  $N = \emptyset$ : Thì  $\text{Clip}_D(F) = \emptyset$ .
- Ngược lại,  $N = [t_1, t_2]$  (quy ước  $t_1 \leq t_2$ ). Gọi  $C, D$  là hai điểm xác định bởi:

$$\begin{aligned} x_C &= x_A + (x_B - x_A)t_1; y_C = y_A + (y_B - y_A)t_1 \\ x_D &= x_A + (x_B - x_A)t_2; y_D = y_A + (y_B - y_A)t_2 \end{aligned}$$

thì  $\text{Clip}_D(AB) = CD$ .

Giải thuật trên vừa đòi hỏi phải tính toán và lượng giá nhiều trên các số thực. Trong thực tế, người ta dùng các giải thuật hiệu quả hơn. Phản tiếp theo, chúng ta sẽ xem xét hai giải thuật với 2 chiến lược xử lý bài toán khác nhau.

### 2.1.1. Giải thuật Liang-Barsky

Giải thuật này là một giải pháp cụ thể cho phương pháp chung đã đề cập ở phần đầu trong (mục 2.1). Giải thuật được mô tả qua các bước:

❖ **Bước 1:** Tính các giá trị sau:

$$\begin{array}{ll} \Delta x = x_B - x_A & \Delta y = y_B - y_A \\ P_1 = -\Delta x & Q_1 = x_A - x_{\min} \\ P_2 = \Delta x & Q_2 = x_{\max} - x_A \\ P_3 = -\Delta y & Q_3 = y_A - y_{\min} \\ P_4 = \Delta y & Q_4 = y_{\max} - y_A \end{array}$$

Từ đây, hệ bất phương trình (3.1) có thể được viết lại theo tham số  $P$  và  $Q$  là:

$$\begin{cases} P_k t \leq Q_k; & k = 1, 2, 3, 4 \\ 0 \leq t \leq 1 \end{cases} \quad (3.2)$$

Các bước tiếp theo để giải bất phương trình (3.2) là:

❖ Bước 2:

- Nếu  $\exists k \in \{1, 2, 3, 4\}$  sao cho ( $P_k = 0$  và  $Q_k < 0$ ): Thì suy ra hệ bất phương trình (3.2) vô nghiệm. Vì vậy giải thuật kết thúc với  $\text{Clip}_D(AB) = \emptyset$ .
- Ngược lại, tiếp tục với bước xử lý tiếp theo.

❖ Bước 3:

Lúc này,  $\forall k \in \{1, 2, 3, 4\}$  ta luôn có  $P_k \neq 0$  hoặc  $Q_k \geq 0$ . Từ đó, các bất phương trình ứng với  $P_k = 0$  sẽ bị loại bỏ bởi vì chúng là hiển nhiên. Từ đây, chúng ta cần tính:

$$K_1 = \{k \mid P_k > 0\} \quad K_2 = \{k \mid P_k < 0\}$$

$$U_1 = \min \left( \left\{ \frac{Q_k}{P_k} \mid k \in K_1 \right\} \cup \{1\} \right) \quad U_2 = \max \left( \left\{ \frac{Q_k}{P_k} \mid k \in K_2 \right\} \cup \{0\} \right)$$

- Nếu  $U_1 < U_2$ : Thì kết thúc giải thuật với  $\text{Clip}_D(AB) = \emptyset$ .
- Ngược lại: Thì  $[U_2, U_1]$  chính là đoạn nghiệm của hệ bất phương trình (3.2) (tương đương với  $[t_1, t_2]$  mà phương pháp tổng quát đã nêu). Nên gọi C và D là hai điểm thỏa mãn công thức sau:

$$\begin{aligned} x_C &= x_A + \Delta x \cdot U_1; & y_C &= y_A + \Delta y \cdot U_1 \\ x_D &= x_A + \Delta x \cdot U_2; & y_D &= y_A + \Delta y \cdot U_2 \end{aligned}$$

kết thúc giải thuật với  $\text{Clip}_D(AB) = CD$ .

Dễ thấy, Bước 3 đã chuyển bài toán giải hệ bất phương trình (3.2) thành quy trình tìm Min và Max để xác định ra hai giá trị  $U_1$  và  $U_2$ , đây chính là điểm hay của giải thuật. Để hiểu được, chúng ta cần nghiên cứu sự chuyển đổi qua từng bước như sau:

- Khi gọi  $K_1 = \{k \mid P_k > 0\}$  và  $K_2 = \{k \mid P_k < 0\}$  thì hệ bất phương trình (3.2) có thể phân tích thành:

$$(3.2) \Leftrightarrow \begin{cases} P_k t \leq Q_k; & \forall k \in K_1 \\ t \leq 1 & \\ P_k t \leq Q_k; & \forall k \in K_2 \\ t \geq 0 & \end{cases} \Leftrightarrow \begin{cases} t \leq Q_k / P_k; & \forall k \in K_1 \\ t \leq 1 & \\ t \geq Q_k / P_k; & \forall k \in K_2 \\ t \geq 0 & \end{cases}$$

Nhóm thứ nhất,

$$\begin{cases} t \leq Q_k / P_k; & \forall k \in K_1 \\ t \leq 1 & \end{cases}$$

Sẽ có nghiệm là:  $t \leq \min \left( \left\{ \frac{Q_k}{P_k} \mid k \in K_1 \right\} \cup \{1\} \right) = U_1$

Nhóm thứ hai,

$$\begin{cases} t \geq Q_k / P_k; & \forall k \in K_2 \\ t \geq 0 & \end{cases}$$

Sẽ có nghiệm là:  $t \geq \max \left( \left\{ \frac{Q_k}{P_k} \mid k \in K_2 \right\} \cup \{0\} \right) = U_2$

Từ đó ta có:

$$(3.2) \Leftrightarrow \begin{cases} t \leq U_1 \\ t \geq U_2 \end{cases} \quad (3.3)$$

Vì vậy, nếu  $U_1 < U_2$  thì hệ bất phương trình (3.3) sẽ vô nghiệm. Ngược lại, nghiệm của nó sẽ là các giá trị  $t$  thuộc đoạn  $[U_2, U_1]$ , thay các giá trị này vào phương trình tham số biểu diễn đoạn thẳng AB chúng ta sẽ có được 2 điểm đầu mút C và D như công thức trong thuật giải.

### 2.1.2. Cài đặt giải thuật Liang-Barsky

Sau đây là phần cài đặt minh họa cho giải thuật nhằm mục đích giúp cho sinh viên có thể tham khảo hay so sánh đánh giá với phần cài đặt thực nghiệm của bản thân.

**Đầu vào:**

- Thông tin của một hình chữ nhật R được lưu trong cấu trúc CRect của MFC.
- Tọa độ 2 điểm đầu mút P1 và P2 của đoạn thẳng cần xén được lưu trong cấu trúc CPoint của MFC.
- Địa chỉ bộ nhớ của 2 biến nhớ theo cấu trúc CPoint được truyền vào 2 biến con trả P1\_Out và P2\_Out, nhằm mục đích nhận kết quả xén là 2 điểm C và D theo giải thuật (trong trường hợp kết quả xén khác rỗng).

**Đầu ra:**

- Hàm sẽ trả về kết quả là **false** nếu kết quả xén là rỗng.
- Ngược lại, hàm trả về **true**, đồng thời giá trị tọa độ của đoạn thẳng kết quả xén sẽ được lưu trữ vào 2 tham biến P1\_Out và P2\_Out.

```

bool GraphicAlgorithms::LiangLineClip(CRect R, CPoint P1, CPoint
P2, CPoint * P1_Out, CPoint * P2_Out)
{
    float Dx = P2.x - P1.x, Dy = P2.y - P1.y;
    float P[4] = { -Dx, Dx, -Dy, Dy };
    float Q[4] = { P1.x - R.left, R.right - P1.x, P1.y - R.top,
R.bottom - P1.y };

    for (int i = 0; i < 4; i++)
    {
        if ((P[i] == 0) && (Q[i] < 0)) // Empty
        {
            return false;
        }
    }

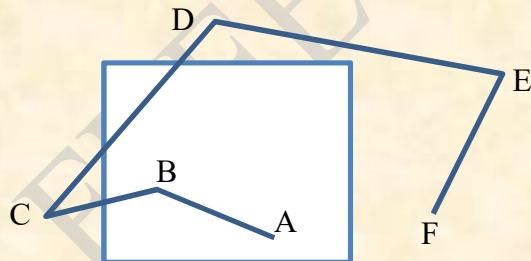
    float U1 = 1, U2 = 0;
    for (int i = 0; i < 4; i++)
    {
        if (P[i] > 0)
        {
    
```

```

        if (U1 > (Q[i] / P[i]))
            U1 = (Q[i] / P[i]);
        }
        else if (P[i] < 0)
        {
            if (U2 < (Q[i] / P[i]))
                U2 = (Q[i] / P[i]);
            }
        }
        if (U1 < U2)
        {
            return false;
        }
        else
        {
            P1_Out->x = int(P1.x + Dx*U1+0.5);
            P1_Out->y = int(P1.y + Dy*U1+0.5);
            P2_Out->x = int(P1.x + Dx*U2 + 0.5);
            P2_Out->y = int(P1.y + Dy*U2+0.5);
            return true;
        }
    }
}

```

### 2.1.3. Giải thuật Cohen-Sutherland



Hình 3.3. Minh họa các tình huống có thể xảy ra khi xén đoạn thẳng vào hình chữ nhật

Xét một cách tổng quát sẽ có các tình huống sau cho bài toán xén một đoạn thẳng vào trong một hình chữ nhật:

- Trường hợp cả hai đầu mút của đoạn thẳng đều nằm trong hình chữ nhật (như đoạn AB) thì Clip<sub>D</sub>(F) sẽ là đoạn thẳng đã cho.

- Trường hợp cả hai đầu mút của đoạn thẳng nằm ở bên ngoài hình chữ nhật, và cùng nằm về một phía so với một cạnh nào đó của hình chữ nhật (như đoạn EF) thì không thể tồn tại phần giao giữa đoạn thẳng với hình chữ nhật D, vậy nên  $\text{Clip}_D(F) = \emptyset$ .
- Trường hợp có một đầu mút của đoạn thẳng nằm ngoài hình chữ nhật (như đoạn BC) thì  $\text{Clip}_D(F)$  sẽ là một phần đoạn thẳng đã cho, được tạo bởi một giao điểm của đoạn thẳng với hình chữ nhật và một đầu của đoạn thẳng nằm trong hình chữ nhật.
- Trường hợp cả hai đầu mút của đoạn thẳng nằm ở ngoài hình chữ nhật, song có một phần của đoạn thẳng nằm trong hình chữ nhật (như đoạn CD) thì đoạn thẳng F cắt hình chữ nhật D đúng hai điểm và khi đó  $\text{Clip}_D(F)$  sẽ là đoạn thẳng tạo bởi hai điểm đó.
- Trường hợp cả hai đầu mút của đoạn thẳng nằm ở ngoài hình chữ nhật, và không có phần nào của đoạn thẳng nằm trong hình chữ nhật (không có giao điểm với hình chữ nhật) thì  $\text{Clip}_D(F) = \emptyset$ .

Với các tình huống như trên, giải thuật sẽ được tiếp cận theo hướng xem xét vị trí hình học của đoạn thẳng so với hình chữ nhật, trên cơ sở đó để tìm kết quả xén, cụ thể:

Trước hết, người ta đánh mã vùng cho không gian mặt phẳng đang xét, sau đó mới tiến hành giải thuật xén dựa trên cơ sở xét mã vùng của các điểm.

- Phương pháp đánh mã vùng:

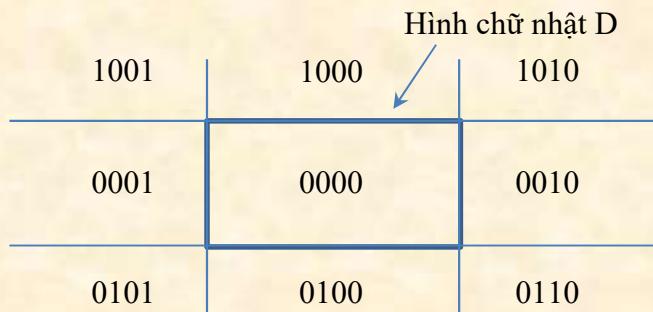
Bảng 3.1. Bảng quy tắc đánh mã

bit 3	bit 2	bit 1	bit 0
Trên	Dưới	Phải	Trái

Mỗi vùng trong mặt phẳng OXY được đánh một mã số tùy theo vị trí của vùng so với hình chữ nhật D, như Hình 3.4. Mỗi mã số có 4 bit, mỗi bit đóng vai trò như một cờ hiệu báo cho chúng ta biết một trạng thái nào đó của vùng. Cụ thể: bit 0 chỉ được bật (bằng 1) khi vùng có mã đang xét nằm về bên *Trái* hình chữ nhật, bit 1 chỉ được bật khi

vùng có mã đang xét nằm về bên *Phải* của hình chữ nhật, bit 2 được bật khi vùng nằm bên *Dưới*, và bit 3 được bật khi vùng nằm bên *Trên*.

Như vậy, mặt phẳng OXY được chia làm 9 phần, mỗi phần được đánh một mã 4 bit biểu hiện trạng thái khác nhau của từng vùng.



Hình 3.4. Phân bổ mã vùng dựa theo vị trí tương ứng với hình chữ nhật

Xét một điểm  $P(x_p, y_p)$  trong mặt phẳng, chúng ta gắn cho nó một mã trùng với mã của vùng chứa nó. Từ đó, chúng ta có thể suy ra cách xác định mã của  $P$  như sau:

$$\begin{aligned} \text{bit 0} &= \begin{cases} 1 & \text{nếu } x_p < x_{min} \\ 0 & \text{nếu ngược lại} \end{cases} & \text{bit 1} &= \begin{cases} 1 & \text{nếu } x_p > x_{max} \\ 0 & \text{nếu ngược lại} \end{cases} \\ \text{bit 2} &= \begin{cases} 1 & \text{nếu } y_p < y_{min} \\ 0 & \text{nếu ngược lại} \end{cases} & \text{bit 3} &= \begin{cases} 1 & \text{nếu } y_p > y_{max} \\ 0 & \text{nếu ngược lại} \end{cases} \end{aligned}$$

Từ đây, quy trình xử lý của giải thuật được thể hiện qua các bước:

❖ Bước 1:

Nếu  $mã(A) = 0000$  và  $mã(B) = 0000$ : Thì  $\text{Clip}_D(F) = AB$ , và kết thúc giải thuật.

❖ Bước 2:

Nếu  $[mã(A) \& mã(B)] \neq 0000$ : Thì  $\text{Clip}_D(F) = \emptyset$ , và kết thúc giải thuật.

*Ở đây phép toán & được hiểu là phép AND bit, hay bitwise-AND operator. Ví dụ:  $1101 \& 1011 = 1001$ .*

*Dễ thấy, trong tình huống này thì đoạn AB nằm hoàn toàn bên ngoài hình chữ nhật. Thật vậy, từ giả thiết suy ra  $mā(A) \neq 0000$  và  $mā(B) \neq 0000$ , nghĩa là A và B cùng nằm bên ngoài hình chữ nhật, mặt khác do kết quả khác 0000 có nghĩa là phải tồn tại ít nhất một bit tại vị trí nào đó bằng 1, giả sử đó là bit i, suy ra cả  $mā(A)$  và  $mā(B)$  đều có bit i có giá trị 1. Điều này có nghĩa là A và B nằm về cùng một phía của hình chữ nhật D. Cả A và B đều ở bên ngoài và nằm về cùng một phía của hình chữ nhật D, nên hiển nhiên là đoạn AB không thể có phần chung với hình chữ nhật D.*

❖ Bước 3:

Nếu  $mā(A) = 0000$ : Thì hoán đổi giá trị A và B.

*Từ đây, chúng ta luôn được đảm bảo rằng điểm A nằm bên ngoài hình chữ nhật khi tiến hành các bước xử lý tiếp theo.*

❖ Bước 4:

Nếu  $x_A = x_B$ : Thì AB là một đoạn thẳng đứng, do đó chúng ta suy ra ngay cách xác định  $Clip_D(AB)$  theo trình tự sau:

➤ Đặt điểm C là giao điểm của AB với hình D như sau:

$$x_C = x_A$$

+ Nếu  $(mā(A) \& 1000) = 1000$  (hay A ở phía trên):

Thì  $y_C = y_{\max}$ .

+ Ngược lại (A ở phía dưới):

Thì  $y_C = y_{\min}$ .

➤ Đặt điểm D như sau:

$$x_D = x_A$$

+ Nếu  $mā(B) = 0000$ : Thì  $y_D = y_B$ .

+ Ngược lại:

- Nếu  $(mā(B) \& 1000) = 1000$  (B ở phía trên):

Thì  $y_D = y_{\max}$ .

- Ngược lại (*B ở phía dưới*):

Thì  $y_D = y_{\min}$ .

$\text{Clip}_D(AB) = CD$ , kết thúc giải thuật.

❖ Bước 5:

Nếu  $x_A \neq x_B$ : *Thì AB là một đoạn thẳng nằm xiên, và A nằm bên ngoài hình chữ nhật.* Tiến hành một trong các yêu cầu dưới đây nếu điều kiện của nó được thỏa mãn:

- Nếu  $[(mã(A) \& 0001) = 0001]$  (*tức A ở bên trái của hình chữ nhật*): Thì thay thế giá trị của A bởi giao điểm của đoạn AB và **cạnh trái nối dài** của hình chữ nhật.
- Nếu  $[(mã(A) \& 0010) = 0010]$  (*tức A ở bên phải của hình chữ nhật*): Thì thay thế giá trị của A bởi giao điểm của đoạn AB và **cạnh phải nối dài** của hình chữ nhật.
- Nếu  $[(mã(A) \& 0100) = 0100]$  (*tức A ở bên dưới của hình chữ nhật*): Thì thay thế giá trị của A bởi giao điểm của đoạn AB và **cạnh dưới nối dài** của hình chữ nhật.
- Nếu  $[(mã(A) \& 1000) = 1000]$  (*tức A ở bên trên của hình chữ nhật*): Thì thay thế giá trị của A bởi giao điểm của đoạn AB và **cạnh trên nối dài** của hình chữ nhật.

Tính lại mã của điểm A vừa thu được, rồi quay lại thực hiện từ Bước 1 (với các giá trị AB mới do đã được thay đổi trong quá trình xử lý ở Bước 5).

**Chú ý:** Trong phần mô tả giải thuật trên, phần chữ in nghiêng chỉ mang tính chất giải thích để làm rõ vấn đề.

❖ Hướng dẫn:

- Giao điểm của một đoạn thẳng AB bất kỳ với các cạnh trái, phải, trên, dưới của hình chữ nhật, có thể tính được dựa trên phương trình biểu diễn đoạn thẳng AB và phương trình biểu diễn cạnh của hình chữ nhật. Cụ thể:

$$\text{Phương trình AB: } \frac{y - y_A}{y_B - y_A} = \frac{x - x_A}{x_B - x_A} \quad \text{hay} \quad \frac{y - y_A}{\Delta y} = \frac{x - x_A}{\Delta x}$$

Phương trình cạnh trái:  $x = x_{\min}$

Suy ra, tọa độ giao điểm của AB với cạnh trái nếu có là:

$$(x_{\min}, \frac{\Delta y}{\Delta x}(x_{\min} - x_A) + y_A)$$

Công thức tính tọa độ giao điểm với các cạnh còn lại của hình chữ nhật được suy luận tương tự.

- Việc tính mã của một điểm có thể được thực hiện dễ dàng nhờ toán tử OR bit (hay bitwise-inclusive-OR operator). Toán tử OR bit cho phép chúng ta dễ dàng bật một bit bất kỳ trong một chuỗi bit. Để tắt một bit trong chuỗi bit, chúng ta cần sử dụng toán tử AND bit.

❖ Cài đặt hàm tính mã

**Đầu vào:** x,y chứa giá trị tọa độ của điểm P; xmin, ymin, xmax, ymax: biểu diễn tọa độ của hình chữ nhật D.

**Đầu ra:** Một byte có 4 bit đầu chứa mã của P và 4 bit sau bằng 0.

```
unsigned char Ma(float x, float y, float xmin, float ymin, float
xmax, float ymax)
{
    unsigned char m = 0; // m = 00000000 Bin

    if (x < xmin)
        m = m | 1; // Tức m or 00000001 Bin, đặt bit 0 bằng 1
    else
        if (x > xmax) m = m | 2; /* Tức m or 00000010 Bin, đặt bit 1
bằng 1 */

    if (y < ymin)
        m = m | 4; // Tức m or 00000100 Bin, đặt bit 2 bằng 1
    else
```

```

    if (y > ymax) m = m | 8; /* Tức m or 00001000Bin, đặt bit 3
bằng 1 */
    return m;
}

```

#### 2.1.4. So sánh, đánh giá hai giải thuật xén Liang-Barsky và Cohen-Sutherland

Do hai giải thuật có hai cách tiếp cận hoàn toàn khác biệt, nên sự so sánh, đánh giá chính xác là rất phức tạp và không phù hợp trong nội dung giáo trình này. Vì vậy, ở đây chúng ta chỉ xem xét (không mang tính đánh giá chính xác) trên một số khía cạnh sau:

##### 1. Yêu cầu về phần cứng thực thi tính toán

Rõ ràng rằng, cả hai giải thuật đều yêu cầu nhiều tính toán trên trường số thực, vì vậy yêu cầu về phần cứng thực thi tính toán của chúng là như nhau chứ không có khác biệt.

##### 2. Số lệnh máy cần thực hiện để có thể tính toán được tọa độ điểm ảnh

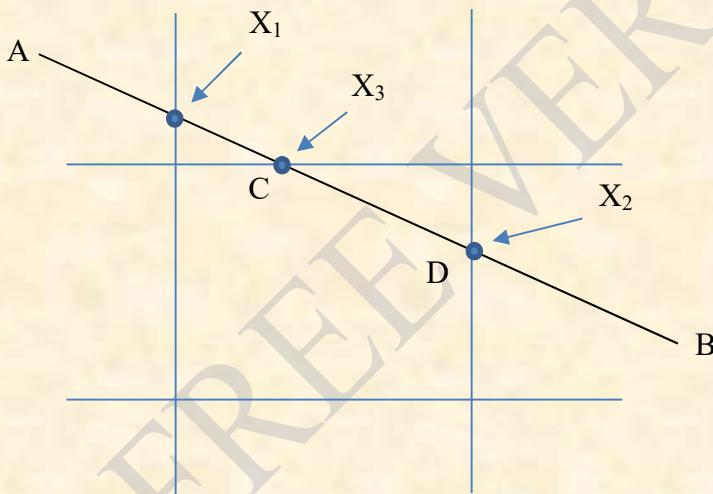
Với cùng một dữ liệu đầu vào, thì các bước xử lý của 2 giải thuật là không giống nhau. Vì thế, việc so sánh đánh giá dựa trên các câu lệnh thực thi tương đương giữa hai giải thuật là điều khó khả thi. Ở đây, chúng ta chỉ ra một số ưu điểm và nhược điểm của mỗi giải thuật và đưa ra một số đánh giá hay nhận định mang tính tương đối.

Tình huống tốt (hay hiệu quả) của thuật toán Cohen-Sutherland là khi đoạn thẳng AB với 2 đầu mút A và B đều ở trong hình chữ nhật D, khi đó thuật toán này chỉ cần tính mã (cần nhiều nhất là 4 lệnh OR bit, một lệnh gán và 4 lệnh rẽ nhánh) của A và B, sau đó thực hiện 2 phép toán so sánh giá trị là thu được kết quả đầu ra chính là giá trị AB đầu vào. Trong khi đó, với thuật toán Liang-Barsky, thì cần phải thực hiện qua tất cả các bước của giải thuật với rất nhiều tính toán mới cho kết quả đầu ra là đoạn thẳng CD với C = A và D = B.

Ngược lại, với tình huống đoạn thẳng AB nằm xiên và cắt hình chữ nhật D tại 2 giao điểm C và D như mô tả trong Hình 3.5, thì giải thuật Cohen-Sutherland phải trải qua nhiều bước lập luận và tính toán. Cụ thể,

giải thuật cần tính mã của A và B, sau đó nó kiểm nghiệm các điều kiện từ **Bước 1** đến **Bước 5**, tiếp đến cần kiểm tra mã của điểm đầu mút thứ nhất (điểm A) để biết nó nằm bên trái hình chữ nhật và tính toán ra tọa độ giao điểm với cạnh trái  $X_1$ , với đoạn thẳng mới còn lại là  $X_1B$  quy trình được lặp lại từ đầu (từ **Bước 1**) để tính được tọa độ giao điểm thứ hai là  $X_2$ . Với đoạn thẳng mới còn lại cho đến lúc này là  $X_1X_2$  quy trình lại được lặp lại (từ **Bước 1**) để thu được giao điểm thứ 3 là  $X_3$ . Rồi lại tiếp tục lặp lại quy trình, với đầu vào  $X_2X_3$  đóng vai trò như AB, lần này chúng ta thu được kết quả  $\text{Clip}_D(AB) = X_2X_3$  (hay đoạn CD nếu gọi tên như trong giải thuật Liang-Barsky) do thỏa mãn điều kiện ở **Bước 1** là  $[\text{mã}(X_1) = 0000 \text{ và } \text{mã}(X_2) = 0000]$ .

Cùng tình huống như trong Hình 3.5, giải thuật Liang-Barsky cũng cần thực hiện qua tất cả các bước của thuật toán. Song, số lệnh cần phải thực hiện dễ thấy là ít hơn.



Hình 3.5. Minh họa tình huống xử lý pharc tạp nhất đối với thuật toán Liang-Barsky

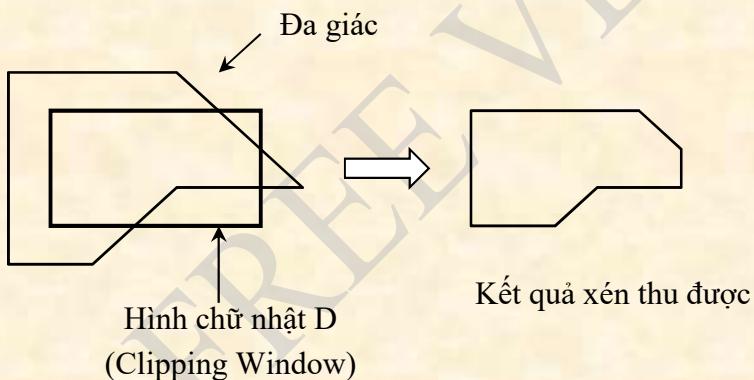
Đối với 2 giải thuật xén này, để thu được những đánh giá hay kết luận có tính khoa học và độ tin cậy cao, thì cần phải thực hiện các thống kê đánh giá về thời gian thực thi trên thực nghiệm với nhiều tình huống khác nhau. Tuy nhiên, điều này không thuộc chủ đề thực hiện và bàn thảo của giáo trình này.

## 2.2. Khi 1 cạnh của hình chữ nhật tạo với trực hoành một góc $\alpha$

Chúng ta dùng phép quay hình để đưa bài toán về trường hợp xén đường thẳng vào hình chữ nhật có cạnh song song với trực tọa độ. Giải thuật được phác thảo như sau:

- ❖ **Bước 1:** Quay hình chữ nhật D và đường thẳng AB một góc  $-\alpha$  để được  $D'$  và  $A'B'$ .
- ❖ **Bước 2:** Thực hiện  $\text{Clip}_{D'}(A'B')$ . Sẽ có hai tình huống xảy ra, cụ thể:
  - Nếu  $\text{Clip}_{D'}(A'B') = \emptyset$ : Thì kết quả  $\text{Clip}_D(AB) = \emptyset$ .
  - Ngược lại: Thì chúng ta sẽ có  $\text{Clip}_{D'}(A'B') = C'D'$ . Tiến hành quay  $C'D'$  một góc  $\alpha$  chúng ta được  $CD$ . Từ đây chúng ta có thể kết luận:  $\text{Clip}_D(AB) = CD$ .

## 3. CLIPPING MỘT ĐA GIÁC VÀO TRONG MỘT VÙNG HÌNH CHỮ NHẬT

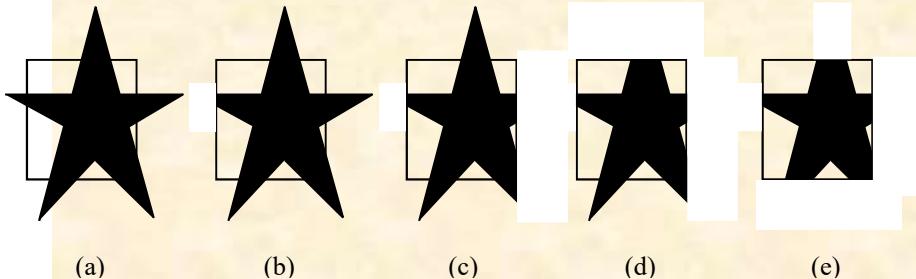


Hình 3.6. Minh họa bài toán xén đa giác vào hình chữ nhật

### 3.1. Giải thuật Sutherland-Hodgman

*Tư tưởng của giải thuật như sau:*

Để Clipping một đa giác F (xem Hình 3.7 (a)) vào trong một hình chữ nhật D chúng ta tiến hành các bước sau:



Hình 3.7. Minh họa kết quả các bước của giải thuật

- ❖ **Bước 1:** Với đa giác  $F$ , thực hiện cắt bỏ những phần nằm bên trái hình chữ nhật (nghĩa là bên trái của cạnh trái nối dài) chúng ta thu được đa giác mới  $F_1$  (xem Hình 3.7 (b)).
- ❖ **Bước 2:** Với đa giác  $F_1$ , thực hiện cắt bỏ những phần nằm bên phải hình chữ nhật chúng ta thu được đa giác mới  $F_2$  (xem Hình 3.7 (c)).
- ❖ **Bước 3:** Với đa giác  $F_2$ , thực hiện cắt bỏ những phần nằm bên trên hình chữ nhật chúng ta thu được đa giác mới  $F_3$  (xem Hình 3.7 (d)).
- ❖ **Bước 4:** Với đa giác  $F_3$ , thực hiện cắt bỏ những phần nằm bên dưới hình chữ nhật chúng ta thu được đa giác mới  $F_4$  (xem Hình 3.7 (e)).

**Kết quả:** Nếu  $F_4 = \emptyset$  thì  $\text{Clip}_D(F) = \emptyset$ . Ngược lại, chúng ta thu được kết quả xén là đa giác  $F_4$ , hay  $\text{Clip}_D(F) = F_4$ .

Để thực hiện các bước (1, 2, 3, 4) chúng ta sẽ tiến hành theo phương pháp sau:

**Chẳng hạn cho bước 1:** Xuất phát từ một đỉnh nào đó của đa giác, chúng ta tiến hành đi dọc theo các cạnh và qua các đỉnh cho đến khi quay về lại đỉnh đầu. Trên quá trình di chuyển:

- Nếu gặp một đỉnh và đỉnh đó ở trên hay bên phải của cạnh trái (nối dài) hình chữ nhật thì chúng ta lưu điểm đó vào  $F_1$ .
- Nếu gặp giao điểm với cạnh trái (hay cạnh trái nối dài) thì lưu giao điểm đó vào  $F_1$ .

Kết quả chúng ta có  $F_1$  là một tập các điểm biểu diễn đa giác  $F$  khi đã xén đi mất phần bên trái.

Các bước còn lại thực hiện tương tự.

### 3.2. Cài đặt giải thuật

Sinh viên cần xây dựng:

- + Một hàm xén đa giác theo giải thuật trên.
- + Một chương trình sử dụng hàm xén để minh họa.

Dưới đây là một cài đặt minh họa, giúp sinh viên có thể tham khảo hay so sánh đánh giá với cách cài đặt của mình:

```

struct Point
{
    double x, y;
};

bool ClipPolygon_Z(Point P[], int Num_P, int Xmin, int Ymin, int
Xmax, int Ymax, Point * &P_clipped, int &Num_P_clipped)
{
    Point *TempP = new Point[Num_P];
    CopyMemory(TempP, P, sizeof(Point)*Num_P);
    int Num_Temp = Num_P;

    P_clipped = new Point[Num_P * 2];
    Num_P_clipped = 0;
    Point A, B, I;
    int ClipEdge = 0; // Xmin edge

    do {

        for (int i = 0; i < Num_Temp; i++)
        {
            A = TempP[i];
            (i < (Num_Temp - 1)) ? B = TempP[i + 1] : B = TempP[0];

            switch (ClipEdge)
            {
                case 0: // Xmin edge

                    if (A.x >= Xmin)
                    {
                        P_clipped[Num_P_clipped] = A;
                        Num_P_clipped++;
                    }
            }
        }
    }
}
```

```

    };

    if (((A.x < Xmin) && (Xmin < B.x)) || (B.x < Xmin) &&
(Xmin < A.x))
    {
        I.x = Xmin;
        I.y = A.y + (Xmin - A.x)*(B.y - A.y) / (B.x - A.x);

        P_clipped[Num_P_clipped] = I;
        Num_P_clipped++;
    }
    break;

case 1: // Xmax edge

if (A.x <= Xmax)
{
    P_clipped[Num_P_clipped] = A;
    Num_P_clipped++;
};

if (((A.x < Xmax) && (Xmax < B.x)) || (B.x < Xmax) &&
(Xmax < A.x))
{
    I.x = Xmax;
    I.y = A.y + (Xmax - A.x)*(B.y - A.y) / (B.x - A.x);

    P_clipped[Num_P_clipped] = I;
    Num_P_clipped++;
}
break;

case 2: // Ymin edge

if (A.y >= Ymin)
{
    P_clipped[Num_P_clipped] = A;
    Num_P_clipped++;
};

if (((A.y < Ymin) && (Ymin < B.y)) || (B.y < Ymin) &&
(Ymin < A.y))

```

```

    {
        I.y = Ymin;
        I.x = A.x + (Ymin - A.y)*(B.x - A.x) / (B.y - A.y);

        P_clipped[Num_P_clipped] = I;
        Num_P_clipped++;
    }
    break;

case 3: // Ymax edge

    if (A.y <= Ymax)
    {
        P_clipped[Num_P_clipped] = A;
        Num_P_clipped++;
    };

    if (((A.y < Ymax) && (Ymax < B.y)) || (B.y < Ymax) &&
(Ymax < A.y))
    {
        I.y = Ymax;
        I.x = A.x + (Ymax - A.y)*(B.x - A.x) / (B.y - A.y);

        P_clipped[Num_P_clipped] = I;
        Num_P_clipped++;
    }
    break;
}

ClipEdge++; // Xử lý với cạnh cắt tiếp theo

if (ClipEdge <= 3)
{
    delete[] Temp;
    Temp = new Point[Num_P_clipped];
    CopyMemory(Temp, P_clipped, sizeof(Point)*Num_P_clipped);
    Num_Temp = Num_P_clipped;

    delete[] P_clipped;
    P_clipped = new Point[Num_Temp * 2];
    Num_P_clipped = 0;
}

```

```

    }

} while (ClipEdge <= 3);

if (Num_P_clipped > 0)
{
    delete[] TempP;
    TempP = new Point[Num_P_clipped];
    CopyMemory(TempP, P_clipped, sizeof(Point)*Num_P_clipped);
    Num_Temp = Num_P_clipped;

    delete[] P_clipped;
    P_clipped = TempP;

    return true;
}
else
{
    delete[] TempP;
    delete[] P_clipped;
    P_clipped = NULL;
    Num_P_clipped = 0;

    return false;
}
}
}

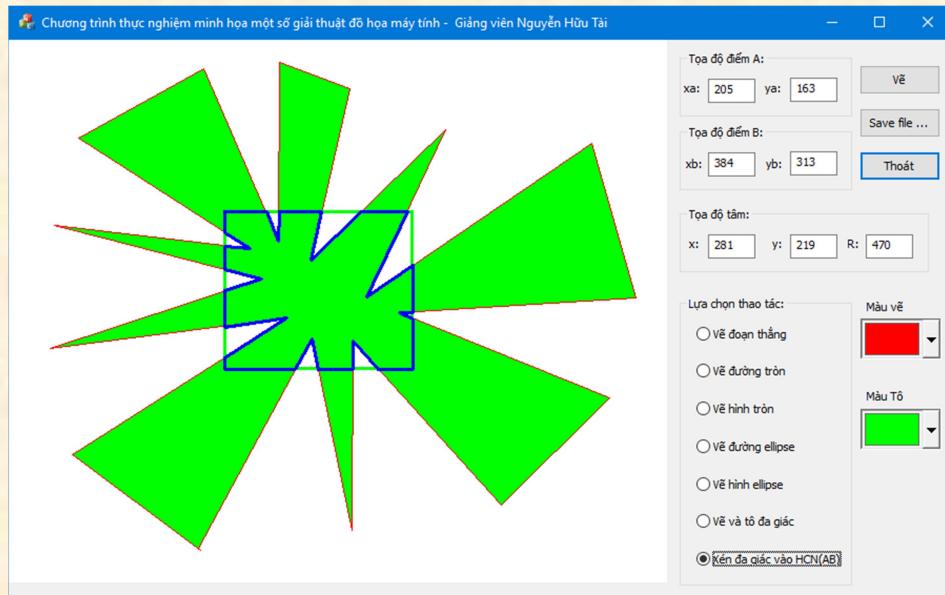
```

Kết quả thực hiện đoạn mã cài đặt trên được thể hiện qua Hình 3.8.

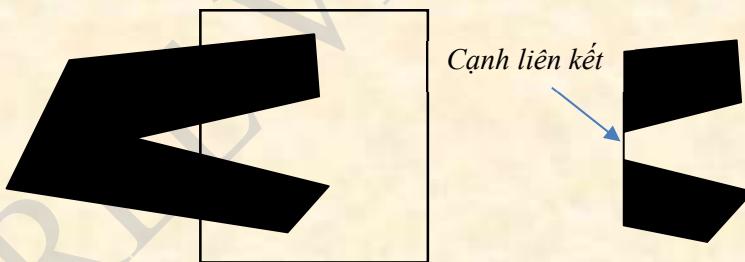
### 3.3. Nhược điểm của giải thuật Sutherland-Hodgman và hướng xử lý khắc phục

Giải thuật xén đa giác Sutherland-Hodgeman còn mắc phải một nhược điểm đó là: Trong tình huống mà kết quả xén bắt buộc phải cho ra 2 đa giác riêng biệt, xem Hình 3.9, thì nó gộp lại làm một đa giác bởi một cạnh liên kết nối đến cả hai đa giác.

Phương pháp khắc phục: Sinh viên tự nghiên cứu. Sau đó, so sánh đối chiếu với giải pháp đã được đề xuất trong tài liệu tham khảo số 2.



Hình 3.8. Hình ảnh thực nghiệm giải thuật Sutherland-Hodgman. Với đầu vào là đa giác lớn (viền màu đỏ) chúng ta thu được đa giác nhỏ (viền màu xanh blue)

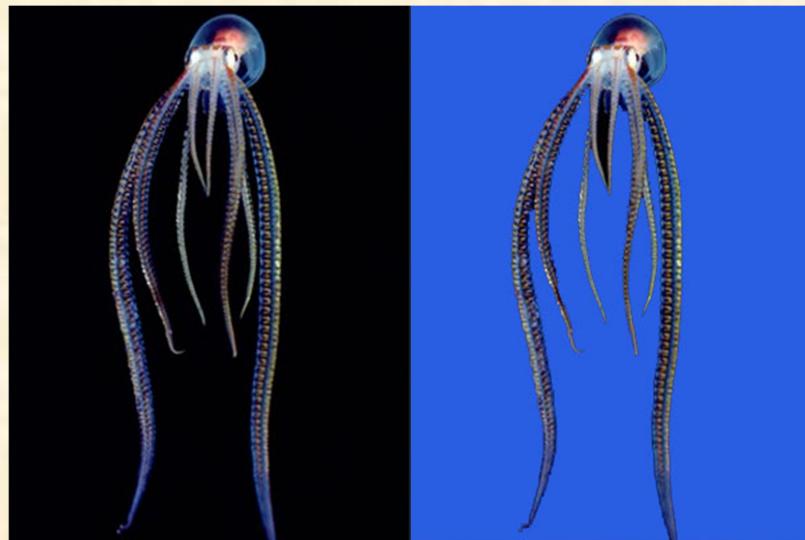


Hình 3.9. Minh họa tình huống còn sai sót của giải thuật

## 4. MỘT SỐ GIẢI THUẬT TÔ MÀU

### 4.1. Giải thuật vết dầu loang

Giải thuật tô màu cho một vùng đồng nhất theo kiểu loang dầu thường được áp dụng để tô màu cho các vùng có đường biên phức tạp rắn, như tô màu cho một vùng ảnh, hay một biến thể của giải thuật là tìm một vùng liên thông đồng nhất màu (hay cùng tone màu) trên ảnh.



Hình 3.10. Ảnh gốc và bản sửa đổi của nó, thu được sau khi tiến hành tô màu vùng nền đen thành nền xanh theo giải thuật vết dầu loang

❖ **Phát biểu bài toán:**

Cho một ma trận điểm ảnh  $M$ , mỗi điểm ảnh có tọa độ là  $(x,y)$  với  $x,y \in Z$ , và có màu là  $\text{Color}(x,y)$ .

Từ một điểm  $P(x_0, y_0) \in M$ , chúng ta xác định một vùng liên thông và đồng nhất về màu sắc, với tên gọi  $Q$ , theo quy tắc sau:

- Đầu tiên, tập  $Q$  chỉ có một phần tử là  $P(x_0, y_0)$ .
- Với mọi điểm ảnh  $T \in M$ , chúng ta sẽ đưa  $T$  vào trong tập  $Q$  nếu tồn tại trong  $Q$  một điểm ảnh  $K$  nào đó sao cho:  $\text{Color}(K) = \text{Color}(T)$  và  $T$  là lân cận của  $K$  (ở đây xét lân cận 4, có nghĩa là  $T$  ở sát trên hoặc sát dưới, hay sát trái hoặc sát phải của  $K$ ).

Biểu diễn về mặt tọa độ thì  $T$  là lân cận của  $K$  khi và chỉ khi:

$$\text{ABS}(X_T - X_K) + \text{ABS}(Y_T - Y_K) = 1$$

Bài toán đặt ra là xuất phát từ một điểm ảnh  $P(x_0, y_0) \in M$  hãy xác định vùng  $Q$  và tô màu cho nó bởi một màu  $C$  nào đó (*dĩ nhiên là  $C$  phải khác màu hiện thời của vùng  $Q$* ).

Trong thực tế, bài toán tô màu theo giải thuật vết dầu loang được ứng dụng rộng rãi trong lĩnh vực đồ họa và xử lý ảnh, để thực hiện tô màu

cho những vùng ảnh với đường biên phức tạp hay chưa xác định được tại thời điểm tiến hành tô màu, hoặc để xác định một vùng ảnh liên thông thỏa mãn một thuộc tính nào đó. Tư tưởng và phương pháp của giải thuật còn được ứng dụng cho: tìm một vùng liên thông khi biết trước 1 điểm, hay lọc ra từ tập hợp các phần tử cùng tính chất theo kiểu lần lượt xích quan hệ...

Có hai phương pháp để giải quyết bài toán này: Thứ nhất là phương pháp đệ quy, thứ hai là phương pháp sử dụng hàng đợi.

#### 4.1.1. Phương pháp đệ quy

❖ **Bước 1:** Lưu giữ màu của điểm ảnh  $P(x_0, y_0)$  vào một biến:

$$\text{OldColor} = \text{Color}(P).$$

❖ **Bước 2:** Tô màu cho điểm ảnh  $P(x_0, y_0)$ .

❖ **Bước 3:** Xác định các điểm ảnh lân cận  $T_i$  ( $i = 1..4$ ) có màu OldColor (cùng màu với điểm ảnh  $P$  trước khi tô).

❖ **Bước 4:** Xem  $T_i$  như vai trò của  $P$  và thực hiện lại từ bước 1.

**Kết luận:** Giải thuật trên sẽ dừng khi không xác định được  $T_i$  ở tất cả các lân đê quy.

*Nhược điểm của phương pháp đệ quy là không thực hiện được khi vùng loang có diện tích lớn do số lần gọi đệ quy lớn sẽ dẫn đến tràn ngắn xếp (stack overflow), và tốc độ thực hiện chậm.*

❖ **Cài đặt giải thuật**

Sinh viên cần xây dựng một hàm đệ quy cho phép tô màu một vùng thuộc cửa sổ của chương trình.

#### Chú ý:

- + Cần tránh tình trạng gọi đệ quy vô hạn không thoát ra được.
- + Cần tránh tô màu cho vùng ảnh có diện tích lớn.

#### 4.1.2. Phương pháp sử dụng hàng đợi

Trên cơ sở của ý tưởng trên, song được xây dựng lại để tránh gọi đệ quy làm hạn chế không gian tô. Giải thuật được mô tả qua hai bước:

❖ **Bước 1:** (Bước khởi động)

Khởi tạo một hàng đợi Q với phần tử đầu tiên là P( $x_0, y_0$ ).

Gọi OldColor là màu của điểm P ( $\text{OldColor} = \text{Color}(P)$ ).

❖ **Bước 2:** (Bước lặp)

Khi hàng đợi Q khác rỗng, lấy từ hàng đợi Q một điểm T và:

➤ Nếu màu hiện thời của T là OldColor thì:

+ Tô màu điểm T.

+ Tìm các điểm lân cận của T có màu là OldColor và đưa chúng vào hàng đợi Q.

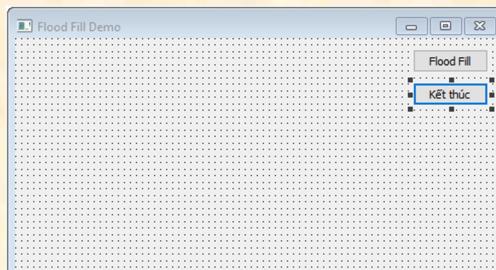
(*Bước 2 được lặp đi lặp lại cho đến khi hàng đợi Q rỗng*).

Giải thuật trên có một nhược điểm là đôi khi một điểm được đưa vào hàng đợi nhiều hơn một lần, từ đó chúng ta có thể tinh chỉnh ở chỗ các điểm lân cận cùng màu chỉ được đưa vào khi nó chưa có trong hàng đợi. Tuy nhiên, việc kiểm tra này sẽ dẫn đến một sự hao phí lớn về thời gian thực hiện kiểm tra hàng đợi, vậy nên chúng ta có thể bỏ qua.

### **Bài thực nghiệm số 2:**

Sau đây, chúng ta sẽ tiến hành các bước thực nghiệm và đánh giá đối với bài toán tô màu theo giải thuật vết dầu loang.

- ❖ Bước 1: Xây dựng một ứng dụng dạng Dialog-Based (xem các bước cần thực hiện ở “***Bài thực nghiệm số 1***”) với tên gọi FloodFill, giao diện đơn giản như hình dưới đây:



Hình 3.11. Giao diện cho ứng dụng minh họa bài toán tô màu sử dụng giải thuật vết dầu loang

- ❖ Bước 2: Tạo một Class mới với tên **PointQueue** như sau:

+ Phần header file:

```
struct PointNode
{
    int x, y;
    PointNode *Next;
};

class PointQueue
{
private:
    PointNode* First = NULL;
    PointNode* Last = NULL;

public:
    PointQueue();
    ~PointQueue();
    bool IsEmpty() { return (First == NULL); };
    void Push(int x, int y);
    bool Pop(int &x, int &y);
    void Delete();
};
};
```

+ Phần source file:

```
PointQueue::PointQueue()
{
}

PointQueue::~PointQueue()
{
    if (First != NULL)
        this->Delete();
}

void PointQueue::Push(int x, int y)
{
    PointNode* P = new PointNode();
    P->x = x; P->y = y; P->Next = NULL;

    if (First == NULL)
        First = P;
```

```

    else
        Last->Next = P;

    Last = P;
}

bool PointQueue::Pop(int &x, int &y)
{
    if (First != NULL)
    {
        PointNode* P = First;

        if (First->Next != NULL)
            First = First->Next;
        else
        {
            First = Last = NULL;
        }

        x = P->x; y = P->y;
        delete P;
        return true;
    }
    else
        return false;
}

void PointQueue::Delete()
{
    while (First != NULL)
    {
        PointNode* P = First;
        First = First->Next;
        delete P;
    }
    First = Last = NULL;
}

```

- ❖ Bước 3: Viết hàm tô màu theo giải thuật vết dầu loang sử dụng kỹ thuật hàng đợi:

```

void CFloodFillDlg::MyFloodFill(int x, int y, COLORREF Color,
CDC* p_DC, CRect R)
{
    int Left = R.left, Right = R.right, Top = R.top, Bottom =
R.bottom;
    if ((x < Left) || (Right < x) || (y < Top) || (Bottom < y))
        // Điểm tô đầu tiên nằm ngoài giới hạn, cần kết thúc
        return;

    COLORREF OldColor = p_DC->GetPixel(x, y);
    if (OldColor == Color)
        /* Màu tô trùng màu hiện tại, do đó không cần thiết thực hiện!
        */
        return;

    PointQueue Queue;
    Queue.Push(x, y);

    while (!Queue.IsEmpty())
    {
        Queue.Pop(x, y);
        if (p_DC->GetPixel(x, y) == OldColor)
            /* Điểm (x,y) chưa hề được xử lý. Đưa vào ĐK này để tránh
            trường hợp 1 điểm được xử lý nhiều lần */
            {
                p_DC->SetPixel(x, y, Color);
                if ((Left <= x-1) && (p_DC->GetPixel(x - 1, y) ==
OldColor))
                    Queue.Push(x - 1, y);
                if ((x+1 <= Right) && (p_DC->GetPixel(x + 1, y) ==
OldColor))
                    Queue.Push(x + 1, y);
                if ((Top <= y-1) && (p_DC->GetPixel(x, y - 1) ==
OldColor))
                    Queue.Push(x, y - 1);
                if ((y + 1 <= Bottom) && (p_DC->GetPixel(x, y + 1) ==
OldColor))
                    Queue.Push(x, y + 1);
            }
    }
}

```

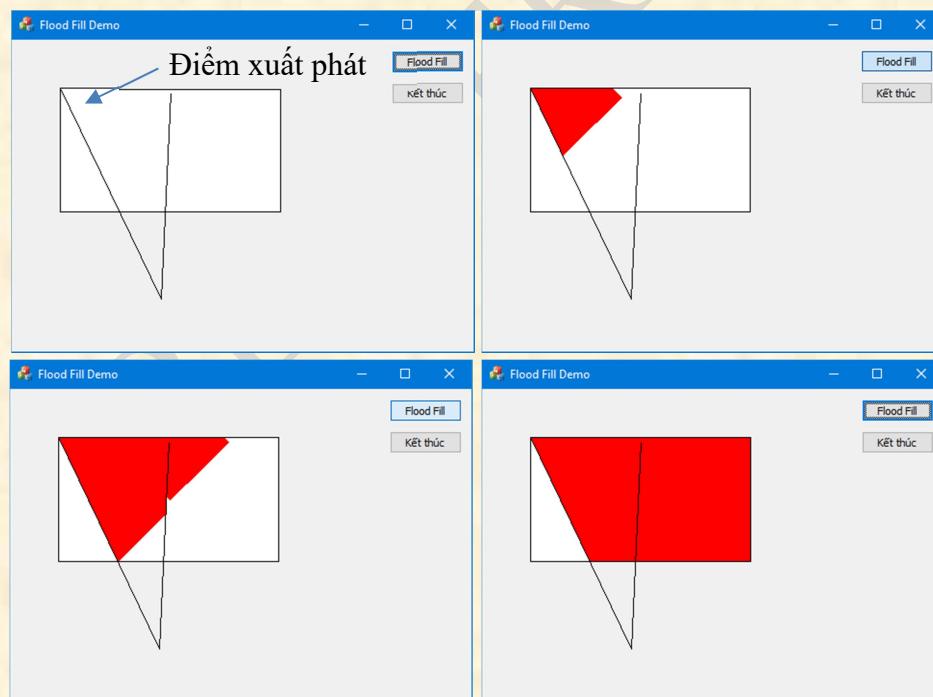
- ❖ Bước 4: Viết hàm xử lý sự kiện nút bấm “Flood Fill” trên cửa sổ:

```
void CFloodFillDlg::OnBnClickedButton1()
{
    CDC *p_DC = this->GetDC();
    CRect R;
    this->GetClientRect(&R);
    p_DC->Rectangle(50, 50, 280, 180);
    p_DC->MoveTo(50, 50); p_DC->LineTo(155, 270); p_DC->LineTo(165,
55);
    COLORREF NewColor = RGB(255, 0, 0);

    MyFloodFill(60, 60, NewColor, p_DC, R);
}
```

- #### ❖ Bước 5: Chạy thực nghiệm:

Kết quả thực nghiệm được thể hiện qua Hình 3.12.

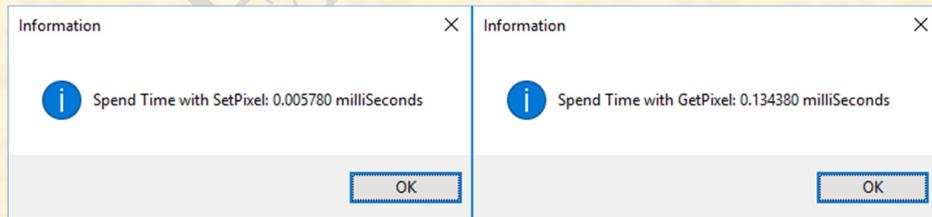


Hình 3.12. Hình ảnh trước khi, trong khi, và sau khi một vùng ảnh trên cửa sổ được tô màu theo giải thuật vết dầu loang

### Phân tích đánh giá:

Tốc độ thực hiện của chương trình “Flood Fill Demo” là quá chậm chạp, *phải mất chừng 20 giây để tô màu vùng ảnh có diện tích bé hơn diện tích hình chữ nhật với kích thước  $230 \times 130$  điểm ảnh khi chạy thực nghiệm trên một máy tính có cấu hình Intel Core i3 CPU mã hiệu M380 tốc độ 2.53GHz + VGA ATI Mobility Radeon HD 5470 với Hệ điều hành Windows 10*, nếu chúng ta đem so sánh với chức năng “fill with color” trong chương trình Paint của Microsoft Windows sẽ thấy được sự chênh lệch về tốc độ là quá lớn và khó có thể chấp nhận. Nguyên nhân của vấn đề nằm ở thời gian thực hiện cho các tác vụ SetPixel và GetPixel trên Device Context của cửa sổ là lớn và khó chấp nhận được trong bài toán tô màu này. Cụ thể:

Thực nghiệm việc gọi hàm SetPixel 1 triệu lần và GetPixel 1 triệu lần trên máy tính windows 10, cấu hình phần cứng là Intel Core i3-380M CPU 2.53GHz + VGA ATI Mobility Radeon HD 5470, thu được thời gian trung bình cho một lần gọi hàm SetPixel vào khoảng 0.005780 millisecond, và hàm GetPixel là 0.134380 millisecond (xem Hình 3.13). Như vậy, nếu chúng ta cần đọc tất cả các điểm ảnh trên một cửa sổ với kích thước  $1920 \times 1080$  (Full HD) để xử lý thì sẽ mất một khoảng thời gian cỡ 278 giây, một khoảng thời gian quá lớn cho các tác vụ xử lý đồ họa hay xử lý ảnh.



Hình 3.13. Chi phí thời gian với các hàm thao tác điểm ảnh SetPixel và GetPixel trên Device Context

Trên nền tảng windows, để tránh tình trạng truy xuất thông tin chậm chạp chúng ta cần tạo cấu trúc DIBSECTION (*tìm hiểu chi tiết về DIBSECTION qua tài liệu tham khảo số 7*) chứa thông tin về một DIB (device-independent bitmap), một định dạng ảnh bitmap không phụ thuộc

thiết bị, từ đó chúng ta có thể tạo ra một dạng “Memory Device Context” (tìm hiểu chi tiết qua tài liệu tham khảo số 8) để có thể thực hiện các tác vụ đồ họa một cách gián tiếp thông qua đối tượng ảnh DIB. Với DIB, chúng ta có thể thực hiện nhanh tác vụ SetPixel hay GetPixel thông qua một con trỏ trả về một mảng các byte dữ liệu (byte array). Vì thế, các tác vụ đọc điểm ảnh hay thiết lập màu cho một điểm ảnh, được thực hiện nhanh chóng chỉ với lệnh truy xuất đến 3 byte dữ liệu trong chế độ đồ họa “24 bit - true color”.

### Cải tiến nâng cấp 1:

Tiếp theo đây, chúng ta sẽ tìm hiểu các bước để tạo một memDC (Memory Device Context) liên kết với một DIB cùng với việc thực hiện tất cả các thao tác đồ họa trên memDC thay cho DC như trước đây, và cuối cùng là đánh giá hiệu quả của giải pháp này. Các bước nâng cấp chương trình FloodFillDemo gồm:

- ❖ Bước 1: Khai báo thêm các biến phục vụ cho việc tạo MemDC trong file FloodFillDlg.h trong Class CFloodFillDlg:

```
private:
    CDC MemDC;
    BITMAPFILEHEADER * pbmfh;
    BITMAPINFO * pbmi;
    HBITMAP hBitmap = NULL;
    BYTE *pBits;
    int BytePerLine; /* Số byte cần thiết để lưu trữ một dòng ảnh.
Giá trị này phải là bộ số của DWORD */
    int BitSize; /* Số byte cần thiết để lưu trữ dữ liệu ảnh */
    bool Created_MemDC = false;
```

- ❖ Bước 2: Thêm hàm tạo MemDC cùng với DIB và hàm hủy MemDC và DIB như dưới đây:

```
bool CFloodFillDlg::CreateMemDC(int Width, int Height)
{
    pbmfh = new BITMAPFILEHEADER;
    pbmi = new BITMAPINFO;

    // Kích thước bộ nhớ cần cấp phát cho ảnh DIB
```

```

BytePerLine = (int)ceil((double)(Width * 3) / 4) * 4;
BitSize = BytePerLine * Height;

pbmfh->bfType = *(WORD*)"BM";

pbmfh->bfOffBits = sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);

pbmfh->bfSize = pbmfh->bfOffBits + BitSize;
pbmfh->bfReserved1 = 0;
pbmfh->bfReserved2 = 0;

pbmi->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
pbmi->bmiHeader.biWidth = Width;
pbmi->bmiHeader.biHeight = -Height; /* Định dạng dữ liệu ảnh được
truy cập theo chiều từ trái sang phải, từ trên xuống dưới (hay Top-
Down) */
pbmi->bmiHeader.biPlanes = 1;
pbmi->bmiHeader.biBitCount = 24; /* Ảnh Bitmap với định dạng 24-
bit */
pbmi->bmiHeader.biCompression = BI_RGB;
pbmi->bmiHeader.biSizeImage = 0;
pbmi->bmiHeader.biXPelsPerMeter = 0;
pbmi->bmiHeader.biYPelsPerMeter = 0;
pbmi->bmiHeader.biClrUsed = 0;
pbmi->bmiHeader.biClrImportant = 0;

hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS,
(void**)&pBits, NULL, 0);
if (hBitmap == NULL)
{
    delete pbmi;
    delete pbmfh;
    delete[] pBits;
    MessageBox(L"Create bitmap failed", L"Information", MB_OK |
MB_ICONINFORMATION);
    return false;
}

FillMemory(pBits, BitSize, 255);

MemDC.CreateCompatibleDC(NULL);

```

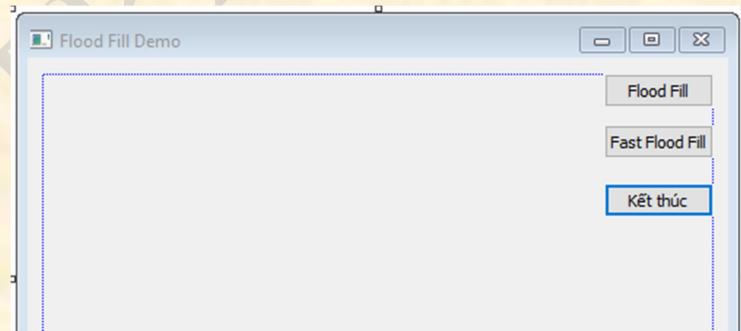
```

CBitmap *OldBitmap = (CBitmap*)MemDC.SelectObject(hBitmap);
}

void CFloodFillDlg::DeleteMemDC()
{
    if (hBitmap != NULL)
    {
        if (DeleteObject(hBitmap))
        {
            hBitmap = NULL;
        }
        else
        {
            hBitmap = NULL;
        }
        delete pbmfh;
        delete pbmi;
    }
    if (MemDC.m_hDC != NULL)
    {
        MemDC.GetCurrentBitmap()->DeleteObject();
        MemDC.DeleteDC();
    };
}

```

- ❖ Bước 3: Thiết kế lại giao diện với một nút mới “Fast Flood Fill” như hình sau:



Hình 3.14. Giao diện chương trình nâng cấp với nút “Fast Flood Fill”

- ❖ Bước 4: Viết hàm đáp ứng sự kiện cho nút “Fast Flood Fill” như sau:

```

void CFloodFillDlg::OnBnClickedButton3()
{
    CRect R(0, 0, 300, 300);

    if (!Created_MemDC)
    {
        Created_MemDC = CreateMemDC(R.Width(), R.Height());
    }

    if (Created_MemDC)
    {
        MemDC.Rectangle(50, 50, 280, 180);
        MemDC.MoveTo(50, 50);
        MemDC.LineTo(155, 270);
        MemDC.LineTo(165, 55);

        COLORREF NewColor = RGB(255, 0, 0);
        MyFloodFill(60, 60, NewColor, &MemDC, R);

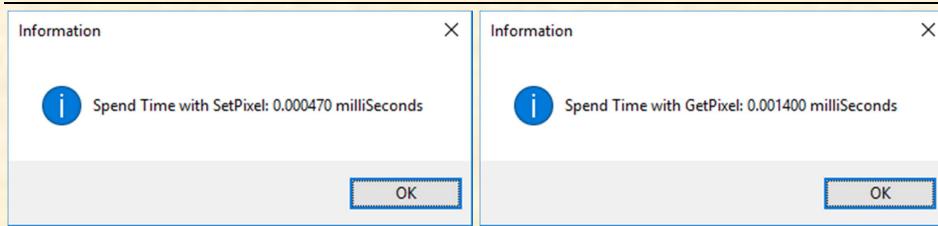
        CDC* p_DC = GetDC();

        p_DC->BitBlt(0, 0, R.Width(), R.Height(), &MemDC, 0, 0,
SRCCOPY); /* Sao chép dữ liệu hình ảnh từ MemDC vào vùng bộ nhớ của
thiết bị (Device Context) */
    }
    else
    {
        MessageBox(L"Can't create MemDC!", L"Error", MB_OK |
MB_ICONERROR);
    }
}

```

❖ Thực nghiệm đánh giá:

- Biên dịch và thực thi chương trình trên với nút “Fast Flood Fill” sẽ cho thấy tốc độ thực hiện rất nhanh, kết quả tô xuất hiện gần như tức thì. Tiến hành đo đạc thời gian thực hiện, *trên máy tính với cấu hình như đã nêu ở phần đánh giá trước*, cho kết quả cỡ 281 mili giây (hay 0.281 giây). Thực nghiệm đo đạc với các hàm SetPixel và GetPixel trên “Memory Device Context” cho kết quả như hình dưới đây:



Hình 3.15. Chi phí thời gian với các hàm thao tác điểm ảnh *SetPixel* và *GetPixel* trên *Memory Device Context*

- Điều dễ thấy là tốc độ xử lý của hàm *SetPixel* khi thực hiện trên ***Memory Device Context*** được cải thiện còn 0.000470 mili giây, so với tốc độ trước đây khi xử lý trên ***Device Context*** là 0.005780 mili giây thì đã nhanh hơn gấp hơn 12 lần. Tương tự, hàm *GetPixel* khi thực hiện trên ***Memory Device Context*** được cải thiện còn 0.001400 mili giây, so với tốc độ trước đây khi xử lý trên ***Device Context*** là 0.134380 mili giây thì đã nhanh hơn gấp gần 96 lần. Kết quả này có thể hiểu một cách giản dị là do chúng ta đã tránh truy xuất trực tiếp đến bộ nhớ của thiết bị đồ họa (hay dành cho thiết bị đồ họa), vốn được quản lý chặt chẽ nhằm đảm bảo khả năng xử lý đồng bộ của thiết bị cho nhiều giao tác nhằm tránh các xung đột. *Tham khảo thêm tài liệu tham khảo số 5 để có kiến thức sâu hơn về vấn đề này.*

Một ví dụ cho chúng ta hình ảnh gần gũi với tình huống này là: Một khách hàng nếu muốn mua một món hàng ở ngoài ranh giới quốc gia của mình, *giả sử chúng được bày bán bên kia biên giới quốc gia*, thì sẽ phải thực hiện các thủ tục xuất cảnh vượt qua biên giới quốc gia, mua món hàng rồi sau đó làm thủ tục nhập cảnh để quay trở lại. Rõ ràng là cho dù nhu cầu chỉ đơn giản là mua một cây kẹo thì thời gian phải bỏ ra cũng không hề ít bởi các thủ tục xuất nhập cảnh mất nhiều thời gian và không thể bỏ qua. Sẽ là thông minh, nếu chúng ta gom nhiều nhu cầu mua sắm để thực hiện cho mỗi một lần xuất nhập cảnh, khi đó chi phí thời gian làm thủ tục tính trên mỗi món hàng sẽ là không đáng kể. Điều này lý giải tại sao nếu chúng ta thực hiện hàm *FillRect*, với kích thước *hình chữ nhật cần tô là  $1000 \times 1000$* , trên ***Device Context*** có thời gian

chênh lệch không nhiều so với thực hiện trên **Memory Device Context**, dù rằng trên lý thuyết để thực hiện hàm FillRect hệ thống cần thực hiện tô 1 triệu điểm, nếu hàm FillRect xử lý trên **Device Context** xử lý bằng một triệu lần gọi hàm Setpixel thì chi phí thời gian sẽ rất lớn và nó tương đương với việc một khách hàng mua một triệu chiếc kẹo với 1 triệu lần xuất nhập cảnh. Cách thức xử lý thực sự của hàm FillRect có thể tham khảo trong các tài liệu chuyên sâu (như *tài liệu tham khảo số 5*).

- Để đẩy nhanh hơn nữa tốc độ truy xuất các điểm ảnh trên Memory Device **Context**, chúng ta cần tránh gọi các hàm SetPixel và GetPixel mà cần truy xuất trực tiếp đến các byte dữ liệu chứa thông tin điểm ảnh của DIB thông qua biến con trỏ **BYTE \*pBits**. Cách làm này sẽ giúp chúng ta tránh lời gọi hàm vốn phải thực hiện khá nhiều công đoạn. Sau đây, chúng ta sẽ tiến hành cách làm này để cải thiện tốc độ cho bài toán tô màu:

### Cải tiến nâng cấp 2:

- ❖ Bước 1: Khai báo thêm hàm MyBestFloodFill, có chức năng tô màu nhưng không sử dụng các hàm SetPixel và GetPixel của hệ thống, mà sẽ thực hiện truy xuất trực tiếp vào vùng bộ nhớ của DIB để thiết lập giá trị điểm ảnh hay lấy thông tin điểm ảnh. Dưới đây là mã lập trình xử lý cho hàm MyBestFloodFill:

```
void CFloodFillDlg::MyBestFloodFill(int x, int y, COLORREF Color,
                                     BYTE *pBits, int BytePerLine, int Width, int Height, CRect R)
{
    /* Định nghĩa cấu trúc dữ liệu của 1 điểm màu lưu trữ trong ảnh
     * DIB định dạng 24-bit. Trật tự lưu trữ sẽ là Blue, Green, Red.
     * Dung Lượng Lưu trữ của cấu trúc này là 3 byte hay 24-bit */
    struct RgbColor
    {
        BYTE b,g,r;
    };

    COLORREF Old_Color = 0x00000000; /* Khai báo biến lưu trữ màu
                                     trước khi tô của vùng ảnh cần tô màu. Nó có dung Lượng là 4 byte
                                     và được gán giá trị mặc định bằng 0 trong cả 4 byte dữ liệu */
```

```

COLORREF Current_Color = 0x00000000; /* Biến lưu trữ màu của
một điểm ảnh đang xử lý trong quá trình thực hiện tô màu */

RgbColor NewRGB_Color = { GetBValue(Color), GetGValue(Color),
GetRValue(Color) };

if (R.IntersectRect(R, CRect(0, 0, Width - 1, Height - 1)) ==
false)
{
    return; /* Vùng tô nằm ngoài phạm vi của ảnh DIB nên kết
thúc */
}

int Left = R.left, Right = R.right, Top = R.top, Bottom =
R.bottom;
if ((x < Left) || (Right < x) || (y < Top) || (Bottom < y)) /* 
Điểm tô đầu tiên nằm ngoài giới hạn, cần kết thúc */
    return;

BYTE *Addr;
Addr = pBits + y*BytePerLine + x * 3; /* Tính địa chỉ vùng nhớ
lưu trữ thông tin của điểm ảnh đầu vào (x,y) trong vùng nhớ cấp
phát cho ảnh DIB, mà địa chỉ bắt đầu của vùng nhớ cấp phát cho
DIB được chứa trong biến con trỏ pBits */

RgbColor *pColor = (RgbColor *)(&Old_Color); /* Lấy địa chỉ
vùng nhớ (4 byte) của biến Old_Color */
*pColor = *((RgbColor *)Addr); /* Thực hiện gán 3 byte đầu tiên
của biến Old_Color bằng 3 giá trị Blue, Green, và Red của điểm
ảnh (x,y). Việc gán này được thực hiện gián tiếp thông qua các
biến con trỏ pColor và Addr đã được tính toán địa chỉ trước đó. Nó
tương đương với Lệnh Old_Color = p_DC->GetPixel(x, y) theo cách
làm trước đây thông qua hàm GetPixel của DC */

if (Old_Color == Color) /* Màu tô trùng với màu hiện tại, do đó
không cần thiết tiến hành tô màu */
    return;

/* Chú ý: Trong hàm này, thay vì so sánh 3 byte dữ liệu màu sắc
của các điểm ảnh trực tiếp với nhau, ví dụ (Color1.r ==
Color2.r)&& (Color1.g == Color2.g)&& (Color1.b == Color2.b) thì
kết luận Color1 bằng Color2, kiểu so sánh này mất thời gian do

```

*phải thực hiện 3 phép toán so sánh và 2 phép logic. Để thực hiện nhanh hơn chúng ta sao chép 3 byte dữ liệu r,g,b này lên các biến kiểu COLORREF độ dài 4 byte, rồi thực hiện so sánh các biến dạng COLORREF Color1 == COLORREF Color2, câu lệnh so sánh này thực hiện hiệu quả trên các hệ điều hành từ 32-bit trở lên \*/*

```

    PointQueue Queue;
    Queue.Push(x, y);

    pColor = (RgbColor *)(&Current_Color); /* Lấy địa chỉ vùng nhớ
(4 byte) của biến Current_Color */

    while (!Queue.IsEmpty())
    {
        Queue.Pop(x, y);
        Addr = pBits + y*BytePerLine + x * 3; /* Tính địa chỉ vùng
nhớ lưu trữ thông tin của điểm ảnh (x,y) (vừa được lấy ra từ hàng
đợi) trong vùng nhớ cấp phát cho ảnh DIB */
        *pColor = *((RgbColor *)(&Addr)); /* Thực hiện gán 3 byte đầu
tiên của biến Current_Color bằng 3 giá trị Blue, Green, và Red
của điểm ảnh (x,y) vừa được lấy ra từ hàng đợi. Nó tương đương
với lệnh Current_Color = p_DC->GetPixel(x, y) theo cách làm trước
đây thông qua hàm GetPixel của DC */

        if (Current_Color == Old_Color) /* Điểm (x,y) chưa hề được
xử lý. Đưa vào ĐK này để tránh trường hợp 1 điểm được xử lý nhiều
lần */
        {
            *((RgbColor *)(&Addr)) = NewRGB_Color; /* Thiết lập giá
trị điểm ảnh tại tọa độ (x,y) bằng màu mới NewRGB_Color, tương
đương cách làm cũ là p_DC->SetPixel(x, y, Color). Chú ý không
dùng biến Color vì nó có định dạng kích thước 4 byte nên không
phù hợp để ghi lên vùng nhớ 3 byte dành cho điểm ảnh (x,y) trên
ảnh DIB định dạng 24-bit. */

            if (Left <= x - 1)
            {
                *pColor = *((RgbColor *)(&Addr - 3)); /* tương đương
hình thức lệnh cũ là p_DC->GetPixel(x - 1, y) */
                if (Current_Color == Old_Color)
                    Queue.Push(x - 1, y);
            }
        }
    }
}

```

```

    if (x + 1 <= Right)
    {
        *pColor = *((RgbColor *) (Addr + 3)); /* tương đương
hình thức lệnh cũ là p_DC->GetPixel(x + 1, y) */

        if (Current_Color == Old_Color)
            Queue.Push(x + 1, y);
    }
    if (Top <= y - 1)
    {
        *pColor = *((RgbColor *) (Addr - BytePerLine)); /* tương
đương hình thức lệnh cũ là p_DC->GetPixel(x, y - 1) */

        if (Current_Color == Old_Color)
            Queue.Push(x, y - 1);
    }
    if (y + 1 <= Bottom)
    {
        *pColor = *((RgbColor *) (Addr + BytePerLine)); /* tương
đương hình thức lệnh cũ là p_DC->GetPixel(x, y + 1) */

        if (Current_Color == Old_Color)
            Queue.Push(x, y + 1);
    }
}
}

```

- ❖ Bước 2: Thiết kế lại giao diện với một nút mới “The best Flood Fill” như hình sau:



Hình 3.16. Nâng cấp giao diện với nút “The best Flood Fill”

- ❖ Bước 3: Viết hàm đáp ứng sự kiện cho nút “The best Flood Fill”:

```

void CFloodFillDlg::OnBnClickedButton4()
{
    CRect R(0, 0, 300, 300);

    if (!Created_MemDC)
    {
        CRect Client_R;
        this->GetClientRect(Client_R);
        Created_MemDC = CreateMemDC(Client_R.Width(),
Client_R.Height());
    }

    if (Created_MemDC)
    {
        MemDC.Rectangle(50, 50, 280, 180);
        MemDC.MoveTo(50, 50); MemDC.LineTo(155, 270);
        MemDC.LineTo(165, 55);

        COLORREF NewColor = RGB(0, 0, 255); // Blue color

        MyBestFloodFill(60, 60, NewColor, pBits, BytePerLine, pbmi-
>bmiHeader.biWidth, abs(pbmi->bmiHeader.biHeight), R);

        CDC* p_DC = GetDC();
        p_DC->BitBlt(0, 0, R.Width(), R.Height(), &MemDC, 0, 0,
SRCCOPY);
    }
    else
    {
        MessageBox(L"Can't create MemDC!", L"Error", MB_OK |
MB_ICONERROR);
    }
}

```

Nếu cần đánh giá thời gian thực hiện công việc tô màu (tính theo mili giây), chúng ta cần thêm vào các hàm **GetTickCount()** để thu về thời gian đã trôi qua kể từ thời điểm máy tính được khởi động (*system was started*) và hiển thị kết quả qua hàm thông báo. Cụ thể hàm đáp ứng sự kiện cho nút “The best Flood Fill” được viết lại như sau:

```

void CFloodFillDlg::OnBnClickedButton4()
{
    CRect R(0, 0, 300, 300);
    if (!Created_MemDC)
    {
        CRect Client_R;
        this->GetClientRect(Client_R);
        Created_MemDC = CreateMemDC(Client_R.Width(),
        Client_R.Height());
    }

    if (Created_MemDC)
    {
        MemDC.Rectangle(50, 50, 280, 180);
        MemDC.MoveTo(50, 50); MemDC.LineTo(155, 270);
        MemDC.LineTo(165, 55);
        COLORREF NewColor = RGB(0, 0, 255); // Blue color

        DWORD Begin = GetTickCount(); // Thời điểm bắt đầu

        MyBestFloodFill(60, 60, NewColor, pBits, BytePerLine, pbmi-
        >bmiHeader.biWidth, abs(pbmi->bmiHeader.biHeight), R);

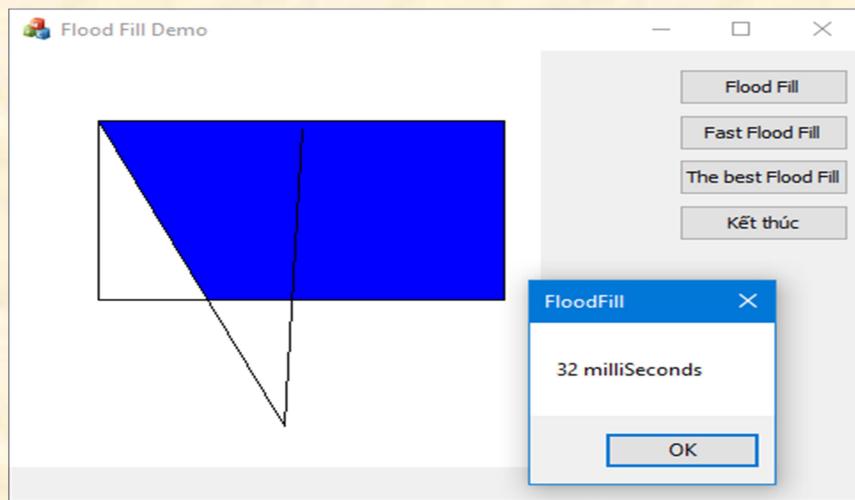
        DWORD End = GetTickCount(); // Thời điểm kết thúc

        CDC* p_DC = GetDC();
        p_DC->BitBlt(0, 0, R.Width(), R.Height(), &MemDC, 0, 0,
        SRCCOPY);

        CString st;
        st.Format(L" %d milliseconds", int(End - Begin)); /* Khoảng
        thời gian đã trôi qua khi thực hiện hàm MyBestFloodFill là : End
        - Begin */
        MessageBox(st);
    }
    else
    {
        MessageBox(L"Can't create MemDC!", L"Error", MB_OK |
        MB_ICONERROR);
    }
}

```

❖ Thực nghiệm đánh giá:

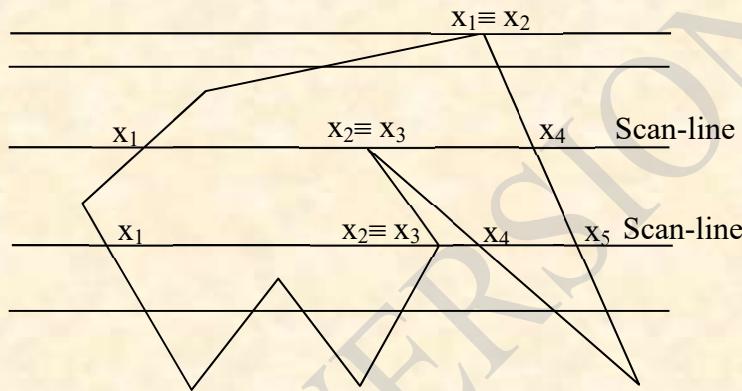


Hình 3.17. Kết quả tô màu và chi phí thời gian thực hiện hàm MyBestFloodFill, thông qua việc truy xuất trực tiếp vào bộ nhớ của ảnh DIB để lấy thông tin hay thiết lập thông tin của điểm ảnh

- Biên dịch và thực thi chương trình trên với nút “The best Flood Fill” sẽ cho thấy tốc độ thực hiện cực kỳ nhanh chóng. Tiến hành đo đạc, trên máy tính với *cáu hình như đã nêu ở phần đánh giá trước*, thời gian thực hiện cho kết quả chỉ chừng 31 mili giây (hay 0.031 giây). Nhanh gấp chừng 645 lần so với cách thực hiện trực tiếp trên vùng nhớ thiết bị màn hình, hay chừng 91 lần khi so với cách thực hiện trên MemDC, nhưng giàn tiếp thông qua các hàm SetPixel và GetPixel.
- Về mặt lý thuyết, cách làm mới cho phép chúng ta thực hiện thiết lập điểm ảnh chỉ với hai thao tác: (1) tính toán địa chỉ, và (2) truy xuất 3 byte nhớ trong vùng bộ nhớ RAM của hệ thống. Sự đơn giản của cách làm mới lý giải cho sự cải thiện hiệu quả về mặt tốc độ. Cách làm này sẽ được áp dụng cho các thuật toán xử lý đồ họa cần nhiều truy xuất thông tin điểm ảnh, vì vậy nó sẽ được áp dụng trong **Bài thực nghiệm số 5 và 6** cài đặt cho thuật toán vùng đệm độ sâu và thuật toán tô bóng Phong.

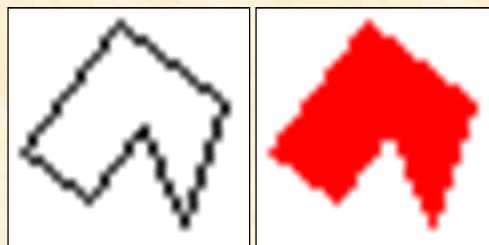
## 4.2. Giải thuật tô màu đa giác theo dòng quét (Scan-line Algorithm)

Giải thuật này tiến hành tìm giao điểm của một đường thẳng nằm ngang, được gọi là *dòng quét (Scan-line)* quét từ đỉnh xuống đáy của đa giác, với các cạnh biên của đa giác. Từ tập các giao điểm thu được, sẽ tiến hành xác định ra các đoạn con nằm bên trong đa giác, để từ đó tiến hành tô màu cho chúng. Kết quả, chúng ta sẽ có được hình ảnh một đa giác với phần bên trong đường biên được tô màu.

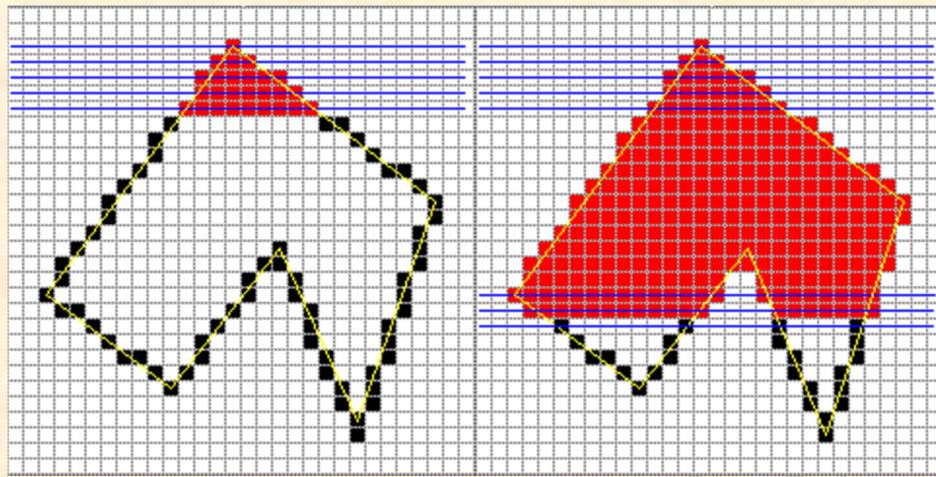


Hình 3.18. Minh họa một số tình huống của dòng quét  
trong giải thuật Scan-line

**Ví dụ:** Cho đa giác đầu vào với 6 đỉnh, thể hiện qua Hình 3.19 (a). Sau khi thực hiện tô màu theo giải thuật Scan-line, chúng ta sẽ thu được kết quả như Hình 3.19 (b). Quá trình thực hiện tô theo dòng quét được mô tả trực qua Hình 3.20, với mỗi dòng quét được xác định tương ứng với một hàng điểm ảnh (Pixel).



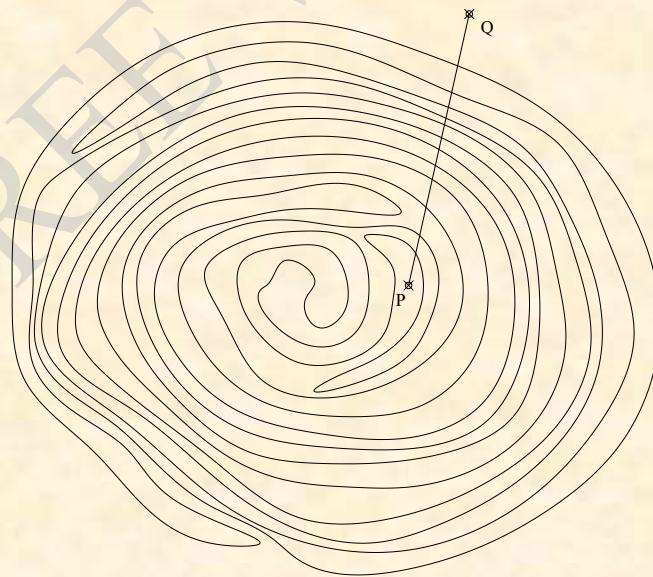
Hình 3.19. Minh họa hình ảnh đa giác trước (a) và sau khi được tô (b)



Hình 3.20. Hình ảnh phóng lớn mô tả quá trình quét (scan) theo hàng để tiến hành tô màu đa giác theo thời gian

Để hiểu rõ hơn tư tưởng của giải thuật chúng ta hãy tìm hiểu một ví dụ sau:

Cho miền D được bao bởi một đường cong kín không tự cắt G và một điểm P như trong Hình 3.21, cần xác định điểm P là trong hay ngoài miền D?



Hình 3.21. Minh họa bài toán “mê cung”

Để trả lời câu hỏi này chúng ta tiến hành như sau:

+ Xác định một điểm Q nào đó ở ngoài miền D.

+ Tìm số giao điểm của đoạn thẳng PQ với G, nếu số giao điểm này là chẵn thì P không thuộc D, ngược lại thì P nằm trong miền D khi số giao điểm là lẻ.

Sở dĩ có được kết luận như vậy là nhờ vào việc áp dụng kết quả rút ra từ một bài toán vui cỗ điển như sau:

Miền D xem như được bao bọc kín bởi tường thành G. Nếu một con kiến xuất phát từ Q và bò dọc theo đoạn thẳng QP để hướng đến P. Rõ ràng, lúc đầu nó ở ngoài thành (ngoài miền D), sau lần vượt thành đầu tiên (qua giao điểm) thì nó sẽ vào trong, nếu nó vượt thành lần nữa thì rõ ràng là nó ra khỏi thành, nếu nó lại tiếp tục vượt thành thì nó lại vào trong, rồi lại vượt để ra ngoài... Rõ ràng, nếu số lần vượt là lẻ thì nó ở trong thành, ngược lại khi số lần vượt là chẵn thì nó ở ngoài thành.

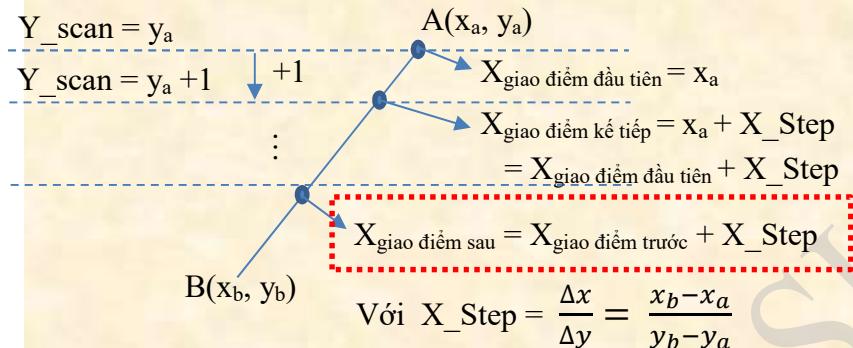
Nhìn ở góc độ của bài toán tô đa giác, chúng ta dễ dàng kết luận rằng: Các đoạn thẳng nằm trong đa giác là các đoạn được tạo bởi cặp giao điểm  $x_k$  với  $x_{k+1}$  với  $k = 1, 3, 5, 7\dots$

Từ đó, để tô màu đa giác chúng ta chỉ việc cho dòng quét (Scan-line) chạy từ đỉnh cao nhất của đa giác đến đỉnh thấp nhất của đa giác, trong mỗi lần chạy chúng ta phải tìm được các giao điểm của dòng quét với đa giác. Tiếp đến, các giao điểm cần được sắp xếp theo thứ tự tăng dần theo hoành độ, gọi các giao điểm đã được sắp xếp theo trật tự là  $x_1, x_2, \dots, x_{2n}$  ( $n > 0$ ), thì chúng ta chỉ việc tô các đoạn  $x_1x_2, x_3x_4, \dots, x_{2n-1}x_{2n}$ .

Vì đa giác được cấu thành từ các đỉnh và các cạnh nối hai đỉnh liền kề. Vậy nên, để tìm giao điểm của dòng quét với đa giác thì chúng ta phải tìm giao điểm với các cạnh của đa giác. Điều tất yếu là, khi dòng quét quét qua một đỉnh của đa giác, thì tại đỉnh đó chúng ta sẽ tìm được đến 2 giao điểm (hay giao điểm kép), và nó sẽ làm cho quy tắc tô màu mô tả ở trên trở nên sai lầm nếu hai cạnh xuất phát từ đỉnh này nằm về hai phía của đường quét. Vì vậy, trong tình huống dòng quét đi qua đỉnh: Nếu hai cạnh của đa giác xuất phát từ đỉnh này nằm về hai phía của dòng quét, thì chúng ta cần bỏ bớt đi một giao điểm.

- ❖ Việc tính toán giao điểm có thể được thực hiện một cách hiệu quả qua phương pháp nội suy kế thừa, cụ thể:

Với một cạnh bất kỳ trong đa giác, gọi A là đỉnh ở trên (Top) và B là đỉnh ở dưới (Bottom, theo hệ tọa độ thiết bị), chúng ta có thể tính toán nội suy giao điểm như sau:



Hình 3.22. Minh họa cơ chế nội suy giao điểm trong giải thuật tô đa giác theo dòng quét (Scanline Algorithm)

- ❖ Cài đặt giải thuật

### Bài thực nghiệm số 3:

Sau đây sẽ là phần cài đặt cho giải thuật tô đa giác theo dòng quét:

```
struct EdgeInfType {
    long Y_Top;
    double X_Intersection;
    int TopIndex;
    double Xstep;
    long Y_Bottom;
    long X_Bottom;
    int BottomIndex;
};

struct IntersectionType {
    long X;
    long VertexIndex;
};
```

```

void FillPoly_ScanLineAlgorithm(POINT P[], int Num_Vertex, CDC*
pDC, COLORREF color)
{ /* phải đảm bảo không có các đoạn thẳng nằm ngang liên tiếp nhau.
Hay nói cách khác là một đoạn thẳng nằm ngang không được chia thành
nhiều cạnh */
}

EdgeInfType * List_Edge;
List_Edge=(EdgeInfType* )calloc(Num_Vertex, sizeof(EdgeInfType));
int cnt=0,i,j;
long Y_Min=P[0].y, Y_Max=P[0].y;
// Ghép nối các đoạn thẳng nằm ngang liên tiếp nếu có
cnt = 1; i = 1;
while (i<Num_Vertex - 1)
{
    if (!(P[i-1].y == P[i].y == P[i + 1].y))
    { //Sao lưu
        if (cnt != i)
        {
            P[cnt] = P[i];
        }
        cnt++; i++;
    }
    else
    {
        P[cnt].x = P[i + 1].x;
        cnt++;
        i += 2;
    }
}

if (i<Num_Vertex)
{
    P[cnt] = P[i];
    cnt++;
}

```

}

```

Num_Vertex = cnt;
/* Trích xuất danh sách các cạnh của đa giác nhằm phục vụ cho
tiến trình tìm giao điểm với Scan-Line được hiệu quả */
cnt = 0;
for (i=0; i<Num_Vertex; i++)
{
    j=ForwardIndex(Num_Vertex,i);

    if (P[i].y<P[j].y)
    {
        List_Edge[cnt].Y_Top=P[i].y;
        List_Edge[cnt].X_Intersection =P[i].x;
        List_Edge[cnt].TopIndex =i;
        List_Edge[cnt].Y_Bottom=P[j].y;
        List_Edge[cnt].X_Bottom=P[j].x;
        List_Edge[cnt].BottomIndex =j;
    }
    else if (P[i].y>P[j].y)
    {
        List_Edge[cnt].Y_Top=P[j].y;
        List_Edge[cnt].X_Intersection =P[j].x;
        List_Edge[cnt].TopIndex =j;
        List_Edge[cnt].Y_Bottom=P[i].y;
        List_Edge[cnt].X_Bottom=P[i].x;
        List_Edge[cnt].BottomIndex =i;
    }
    else //Đoạn thẳng nằm ngang
    {
        if (P[i].x<P[j].x)
        {
            List_Edge[cnt].Y_Top=P[i].y;
            List_Edge[cnt].X_Intersection =P[i].x;
        }
    }
}

```

```

        List_Edge[cnt].TopIndex =i;
        List_Edge[cnt].Y_Bottom=P[j].y;
        List_Edge[cnt].X_Bottom=P[j].x;
        List_Edge[cnt].BottomIndex =j;
    }
    else
    {
        List_Edge[cnt].Y_Top=P[j].y;
        List_Edge[cnt].X_Intersection =P[j].x;
        List_Edge[cnt].TopIndex =j;
        List_Edge[cnt].Y_Bottom=P[i].y;
        List_Edge[cnt].X_Bottom=P[i].x;
        List_Edge[cnt].BottomIndex =i;
    }
}

double DX=P[j].x - P[i].x, DY=P[j].y-P[i].y;
if (DY==0)
    List_Edge[cnt].Xstep =0;
else
    List_Edge[cnt].Xstep =DX/DY;
cnt++;
// Tìm Min Max
if (Y_Min>P[i].y)
    Y_Min=P[i].y;
if (Y_Max<P[i].y)
    Y_Max=P[i].y;
}

IntersectionType *ListIntersection;
ListIntersection=(IntersectionType*)calloc(Num_Vertex*2,sizeof(IntersectionType));

for (int Yscan=Y_Min; Yscan<=Y_Max; Yscan++)

```

```

{
    cnt=0;
    for (i=0; i<Num_Vertex; i++)
    {
        if ((List_Edge[i].Y_Top<=Yscan)&&
            (Yscan<=List_Edge[i].Y_Bottom))
        {
            if (Yscan==List_Edge[i].Y_Top)
            {
                ListIntersection[cnt].X = List_Edge[i].X_Intersection;
                ListIntersection[cnt].VertexIndex =
                    List_Edge[i].TopIndex;
            }
            else if (Yscan==List_Edge[i].Y_Bottom )
            {
                ListIntersection[cnt].X = List_Edge[i].X_Bottom;
                ListIntersection[cnt].VertexIndex =
                    List_Edge[i].BottomIndex;
            }
            else
            {
                ListIntersection[cnt].X =
int(List_Edge[i].X_Intersection+0.5); /* chú ý làm tròn số thực về
số nguyên */
                ListIntersection[cnt].VertexIndex = -1; /* Dấu hiệu
nhận biết không đi qua đỉnh, nên không cần xét giao điểm kép */
            }
            cnt++;
        }

        List_Edge[i].X_Intersection += List_Edge[i].Xstep ;

        if (List_Edge[i].Y_Top==List_Edge[i].Y_Bottom)
            //đường nằm ngang
        {
    }
}

```

```

        int Y1=List_Edge[i-1].Y_Top;
        if (Y1==Yscan)
            Y1=List_Edge[i-1].Y_Bottom;

        j=ForwardIndex(Num_Vertex,i);
        int Y2=List_Edge[j].Y_Top;
        if (Y2==Yscan)
            Y2=List_Edge[j].Y_Bottom;

        if ((Y1-Yscan)*(Y2-Yscan)>0) /* hai cạnh liền kề nằm
về 1 phía => chấp nhận giao điểm kép ở 2 đầu của đoạn thẳng */
        {
            ListIntersection[cnt-1].VertexIndex = -1; /* Dấu
hiệu nhận biết không cần xét giao điểm kép đối với đường nằm ngang */
            ListIntersection[cnt].X =List_Edge[i].X_Bottom;
            ListIntersection[cnt].VertexIndex = -1; /* Dấu hiệu
nhận biết không cần xét giao điểm kép đối với đường nằm ngang */
            cnt++;

            ListIntersection[cnt].X =List_Edge[i].X_Bottom;
            ListIntersection[cnt].VertexIndex = -1; /* Dấu hiệu
nhận biết không cần xét giao điểm kép đối với cạnh nằm kế tiếp với
đường nằm ngang */
            cnt++;
            i++; // bỏ qua cạnh tiếp theo do đã đưa giao điểm
vào danh sách
        }
        else // chỉ nhận 1 giao điểm tại đầu phía bên phải
        {
            ListIntersection[cnt-1].VertexIndex = -1; /* Dấu
hiệu nhận biết không cần xét giao điểm kép đối với đường nằm ngang */
            ListIntersection[cnt].X =List_Edge[i].X_Bottom;
            ListIntersection[cnt].VertexIndex = -1; /* Dấu hiệu
nhận biết không cần xét giao điểm kép đối với đường nằm ngang */
        }
    }
}

```

```

        cnt++;
        i++; /* bỏ qua cạnh tiếp theo do đã đưa giao điểm
vào danh sách */
    }
}
}

/* Kết thúc vòng lặp duyệt qua các cạnh để tìm giao điểm
với Scanline */

// Sắp xếp giao điểm từ trái sang phải (hay từ bé đến lớn)
for (i=0; i<cnt-1; i++)
{
    long Min=ListIntersection[i].X, ChiSo=i;
    for (j=i+1; j<cnt; j++)
        if (Min>ListIntersection[j].X)
    {
        Min=ListIntersection[j].X; ChiSo=j;
    }
    if (ChiSo!=i)
    {
        IntersectionType tg=ListIntersection[i];

        ListIntersection[i]=ListIntersection[ChiSo];
        ListIntersection[ChiSo]=tg;
    }
}

long *ListX;
ListX=(long*)calloc(cnt*2,sizeof(long));
long cnt2=0;
// Xem xét với các giao điểm kép
for (int k=0; k<cnt-1; k++)
{
    ListX[cnt2]=ListIntersection[k].X;
    cnt2++;
}

```

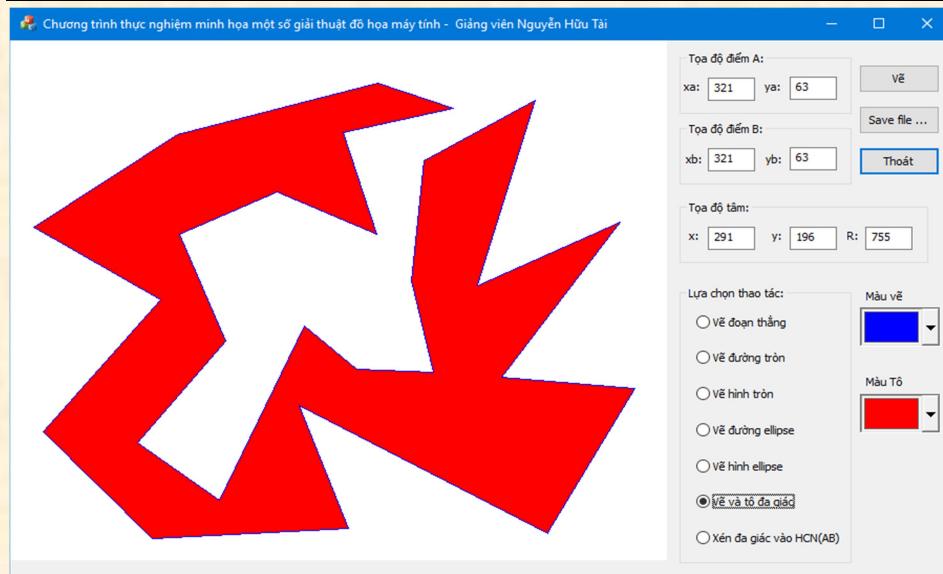
```

    if ((ListIntersection[k].VertexIndex!=-1)&&
(ListIntersection[k].VertexIndex==ListIntersection[k+1].VertexIndex
))
{
    i=BackwardIndex(Num_Vertex,ListIntersection[k].VertexIndex);
    j=ForwardIndex(Num_Vertex,ListIntersection[k].VertexIndex);
    if ((Yscan-P[i].y)*(Yscan-P[j].y)<0) /* hai cạnh nằm về
hai phía */
        cnt2--;
}
ListX[cnt2]=ListIntersection[cnt-1].X;
cnt2++;

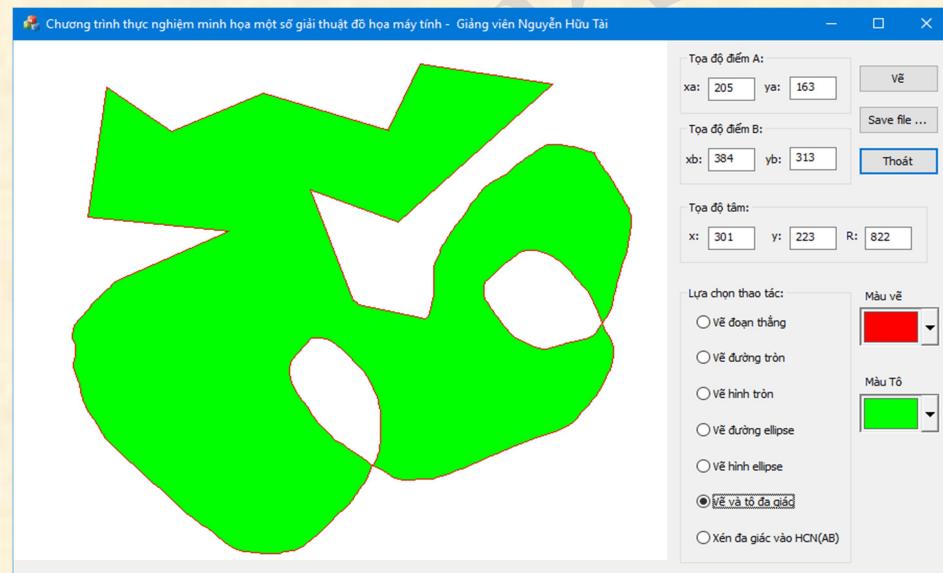
for (int k=0; k<cnt2-1; k+=2)
    pDC->FillSolidRect( ListX[k], Yscan, ListX[k+1]-ListX[k],
1, color);
}
pDC->MoveTo(P[0]);
for (i=0; i<Num_Vertex; i++)
{
    j=i+1;
    if (i==(Num_Vertex-1))
        j=0;
    pDC->LineTo(P[j]);
}
}
}

```

Dưới đây là hình ảnh minh họa kết quả thực hiện hàm tô màu đa giác được xây dựng ở trên:



Hình 3.23. Kết quả tô đa giác có 28 đỉnh



Hình 3.24. Kết quả tô đa giác có 561 đỉnh, trong đó phần nhiều là các cạnh với độ dài rất bé

## 5. BÀI TẬP CUỐI CHƯƠNG

- Cài đặt hàm xén đoạn thẳng vào đa giác, xén đoạn thẳng vào Ellipse, xén Ellipse vào hình chữ nhật.

2. Cài đặt hàm tô màu đa giác theo mẫu tô (tô có hoa văn) như sau:
  - a. Khi mẫu là một ma trận  $8 \times 8$  bit, nếu bit = 1 thì được tô ngược lại thì không tô.
  - b. Khi mẫu tô là một ảnh bitmap kích thước  $m \times n$ .
3. Bổ sung vào thư viện do bạn cài đặt được ở chương trước các hàm xén và tô màu.

## Chương 4

### CÁC PHÉP BIẾN ĐỔI HÌNH HỌC

Biến đổi hình học là một nội dung quan trọng của các hệ thống đồ họa máy tính. Trong không gian 2 chiều, các phép biến đổi giúp người dùng (hay người lập trình) tạo ra các hình ảnh đa dạng phong phú từ một tập các hình cơ bản mà hệ thống đã xây dựng. Ví dụ, hệ thống chỉ cung cấp hàm dựng ellipse cơ bản với 2 trục a và b của ellipse lần lượt song song với hai trục x và y của hệ thống, để có thể tạo nên hình ellipse có trục theo hướng bất kỳ chúng ta cần thực hiện một phép quay. Trong không gian 3 chiều, các phép biến đổi hình học có vai trò rất quan trọng giúp thể hiện các thay đổi về hình dáng và chuyển động của đối tượng trong không gian mô phỏng, là chìa khóa để chúng ta có thể mô phỏng đối tượng trong không gian ảo và quan sát đối tượng một cách tự do dưới mọi góc nhìn.



Hình 4.1. Hình ảnh thu được từ các góc quan sát khác nhau của cùng một đối tượng. Việc thay đổi góc quan sát được thực hiện thông qua các phép biến đổi hình học 3 chiều

## 1. CÁC PHÉP BIẾN ĐỔI HÌNH HỌC HAI CHIỀU (AFFINE 2D)

Phép biến đổi Affine 2D sẽ biến đổi điểm P có tọa độ ( $P_x, P_y$ ) thành điểm Q có tọa độ ( $Q_x, Q_y$ ) theo hệ phương trình sau:

$$\begin{cases} Q_x = a.P_x + c.P_y + tr_x \\ Q_y = b.P_x + d.P_y + tr_y \end{cases}$$

Dưới dạng ma trận, hệ này có dạng:

$$(Q_x, Q_y) = (P_x, P_y) \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} + (tr_x, tr_y)$$

Hay viết gọn hơn:  $\mathbf{Q} = \mathbf{P} \cdot \mathbf{M} + \mathbf{Tr}$  (4.1)

với  $\mathbf{Q} = (Q_x, Q_y), \mathbf{P} = (P_x, P_y)$

$\mathbf{Tr} = (tr_x, tr_y)$ : Vector tịnh tiến

$$\mathbf{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

M: Ma trận biến đổi

### 1.1. Phép tịnh tiến

Phép tịnh tiến điểm P thành điểm Q theo vector  $\mathbf{Tr} = (tr_x, tr_y)$  có hệ phương trình:

$$\begin{cases} Q_x = P_x + tr_x \\ Q_y = P_y + tr_y \end{cases}$$

hay theo dạng ma trận như công thức (4.1) thì:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ và } \mathbf{Tr} = (tr_x, tr_y)$$

### 1.2. Phép đồng dạng

Ở dạng phương trình là:  $\begin{cases} Q_x = s_x P_x \\ Q_y = s_y P_y \end{cases}$

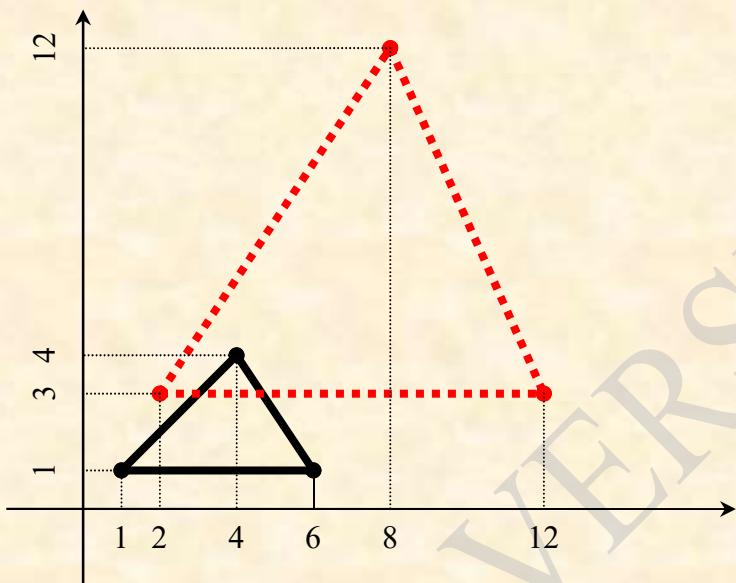
Dưới dạng ma trận:

$$\mathbf{M} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \text{ và } \mathbf{Tr} = (0,0)$$

Với:  $s_x$ : Là hệ số tỷ lệ theo trục x.

$s_y$ : Là hệ số tỷ lệ theo trục y.

Phép đồng dạng (hay còn gọi là phép biến đổi tỷ lệ) cho phép chúng ta phóng to hay thu nhỏ hình theo một hay hai chiều mà vẫn giữ được hình dáng cơ bản của chúng (đồng dạng).



Hình 4.2. Minh họa phép biến đổi đồng dạng cho một tam giác

### 1.3. Phép đối xứng

Đây là trường hợp đặc biệt của phép đồng dạng với  $a$  và  $d$  đối nhau.

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{đối xứng qua } Oy$$

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \text{đối xứng qua } Ox$$

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \quad \text{đối xứng qua gốc tọa độ}$$

### 1.4. Phép quay quanh gốc tọa độ

Ma trận tổng quát của phép quay quanh gốc tọa độ một góc  $\alpha$  là:

$$M = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \text{ và } Tr = (0,0)$$

Chú ý:

- Tâm của phép quay được xét ở đây là gốc tọa độ.
- Định thức của ma trận phép quay luôn luôn bằng 1.

### 1.5. Phép biến dạng (Twist Transformation)

$$\text{Ma trận tổng quát là: } M = \begin{pmatrix} 1 & t_y \\ t_x & 1 \end{pmatrix} \text{ và } Tr = (0,0)$$

Trong đó:

$t_x$ : Là hệ số biến dạng theo trục x.

$t_y$ : Là hệ số biến dạng theo trục y.

### 1.6. Tọa độ thuần nhất (Homogeneous Coordinates)

Để chỉ còn phép nhân ma trận trong các phép biến đổi, chúng ta thêm vào một thành phần trong các ma trận và đưa nó về dạng:

$$(Q_x, Q_y, 1) = (P_x, P_y, 1) \cdot \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ tr_x & tr_y & 1 \end{pmatrix}$$

Ký hiệu  $(Q_x, Q_y, 1), (P_x, P_y, 1)$  được gọi là tọa độ thuần nhất, và công thức của phép biến đổi theo tọa độ thuần nhất là:

$$Q = P \cdot T$$

Với T là ma trận  $3 \times 3$  được gọi là ma trận biến đổi.

### 1.7. Tổng hợp các phép biến đổi Affine

Gọi  $f_1$  và  $f_2$  là 2 phép biến đổi như sau:

$$f_1 : P \rightarrow Q$$

$$f_2 : Q \rightarrow W$$

Chúng ta cần tìm phép biến đổi  $P \rightarrow W$ .

Giả sử  $Q = P \cdot M_1 + Tr_1$  và  $W = Q \cdot M_2 + Tr_2$  thế thì:

$$W = (P \cdot M_1 + Tr_1) \cdot M_2 + Tr_2 = P \cdot M_1 \cdot M_2 + Tr_1 \cdot M_2 + Tr_2$$

Đặt  $M = M_1 \cdot M_2$  và  $Tr = Tr_1 \cdot M_2 + Tr_2$  thì:  $W = P \cdot M + Tr$ .

Như vậy, tổng hợp (hay tích) của hai phép biến đổi Affine cũng là một phép biến đổi Affine, có các ma trận biến đổi là:

$$M = M_1 \cdot M_2 \text{ và } Tr = Tr_1 \cdot M_2 + Tr_2$$

❖ Nếu sử dụng tọa độ thuần nhất thì:

$$Q = P \cdot T_1; \quad W = Q \cdot T_2$$

$$\text{Suy ra: } W = (P \cdot T_1) \cdot T_2 = P \cdot T \quad \text{với } T = T_1 \cdot T_2$$

Như vậy, tổng hợp (hay tích) của hai phép biến đổi trong hệ tọa độ thuần nhất cũng là một phép biến đổi trong hệ tọa độ thuần nhất, có ma trận biến đổi là:

$$T = T_1 \cdot T_2$$

Lập luận tương tự chúng ta có: Tổng hợp của  $N$  phép biến đổi trong hệ tọa độ thuần nhất cũng là một phép biến đổi, có ma trận biến đổi là:

$$T = T_1 \cdot T_2 \dots T_N$$

**Chú ý:** Từ đây trở đi, chúng ta chỉ sử dụng tọa độ homogeneous trong các phép biến đổi để việc tìm tích của các phép biến đổi được trở nên đơn giản.

Sau đây là ma trận biến đổi theo tọa độ homogeneous của một số phép biến đổi cơ bản:

Bảng 4.1. Bảng ma trận của các phép biến đổi cơ bản trong không gian 2 chiều theo hệ tọa độ thuần nhất

Phép biến đổi	Ma trận Homogeneous
Phép tịnh tiến	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tr_x & tr_y & 1 \end{pmatrix}$

Phép quay	$\begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Phép đối xứng qua trục OX	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Phép đối xứng qua trục OY	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Phép đối xứng qua gốc tọa độ	$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Phép đồng dạng	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Phép biến dạng	$\begin{pmatrix} 1 & t_y & 0 \\ t_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

### 1.8. Phép quay quanh điểm bất kỳ

Giả sử cho điểm P quay quanh điểm V( $V_x, V_y$ ) một góc  $\alpha$ , chúng ta được ảnh của P qua phép biến đổi là Q.

Phép quay này bao gồm các phép biến đổi cơ bản sau:

- Tịnh tiến P theo vector  $\vec{VQ} = (-V_x, -V_y)$  chúng ta được P'.
- Quay P' quanh gốc tọa độ một góc  $\alpha$  chúng ta được Q'.
- Tịnh tiến Q' theo vector  $\vec{OV} = (V_x, V_y)$  chúng ta được Q.

Q chính là ảnh của P qua phép quay quanh điểm V một góc  $\alpha$ . Ma trận biến đổi là:

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -V_x & -V_y & 1 \end{pmatrix} \times \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ V_x & V_y & 1 \end{pmatrix}$$

### 1.9. Các ví dụ minh họa

#### 1.9.1. Ví dụ 1

Cho tam giác ABC có tọa độ lần lượt là A(0,0), B(6,0) và C(3,5). Quay tam giác ABC một góc  $90^\circ$  quanh điểm M(3,3) chúng ta được tam giác A<sub>1</sub>B<sub>1</sub>C<sub>1</sub>. Hãy tìm ma trận của phép biến đổi và tọa độ của A<sub>1</sub>B<sub>1</sub>C<sub>1</sub>.

Giai: Phép quay quanh điểm M(3,3) một góc  $90^\circ$  được biểu diễn qua 3 phép biến đổi cơ bản lần lượt là:

+ Tịnh tiến theo vector  $\overrightarrow{MO} = (x_O - x_M, y_O - y_M) = (-3, -3)$ .

$$\text{Ma trận biểu diễn là } T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -3 & 1 \end{pmatrix}$$

+ Quay quanh gốc tọa độ một góc  $90^\circ$

Ma trận biểu diễn là:

$$T_2 = \begin{pmatrix} \cos(90^\circ) & \sin(90^\circ) & 0 \\ -\sin(90^\circ) & \cos(90^\circ) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

+ Tịnh tiến ngược trở lại theo vector  $\overrightarrow{OM} = (3,3)$ .

$$\text{Ma trận biểu diễn là } T_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 3 & 1 \end{pmatrix}$$

Như vậy, ma trận của phép quay quanh điểm M một góc  $90^\circ$  sẽ được tính bằng tích các ma trận biến đổi thành phần:

$$T = T_1 \times T_2 \times T_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -3 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 3 & 1 \end{pmatrix}$$

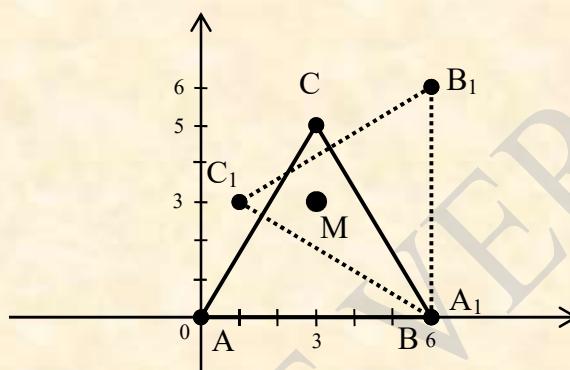
$$= \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 3 & -3 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 6 & 0 & 1 \end{pmatrix}$$

$$A_1B_1C_1 \text{ (tọa độ homogeneous)} = ABC \text{ (tọa độ homogeneous)} \times T$$

$$= \begin{pmatrix} 0 & 0 & 1 \\ 6 & 0 & 1 \\ 3 & 5 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 6 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 6 & 0 & 1 \\ 6 & 6 & 1 \\ 1 & 3 & 1 \end{pmatrix}$$

Vậy  $A_1(6,0)$   $B_1(6,6)$  và  $C_1(1,3)$ .

Vẽ hình minh họa:



biến đổi cơ bản sau:

- Phép tịnh tiến theo vector  $v_1 = (0, -4)$  (tịnh tiến CD về trung với trục OX):

$$\text{Ma trận } T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -4 & 1 \end{pmatrix}$$

- Phép lấy đối xứng qua trục OX:

$$\text{Ma trận } T_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Phép tịnh tiến ngược trở lại theo vector  $v_2 = -v_1 = (0, 4)$ :

$$\text{Ma trận } T_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 4 & 1 \end{pmatrix}$$

Vậy ma trận của phép lấy đối xứng qua CD là:

$$T_4 = T_1 \times T_2 \times T_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 8 & 1 \end{pmatrix}$$

Tiếp đến phép quay một góc  $-90^\circ$  có ma trận là:

$$T_5 = \begin{pmatrix} \cos(-90^\circ) & \sin(-90^\circ) & 0 \\ -\sin(-90^\circ) & \cos(-90^\circ) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Phép biến đổi tổng hợp biến đổi ABCD thành A<sub>1</sub>B<sub>1</sub>C<sub>1</sub>D<sub>1</sub> là tổng hợp của 2 phép biến đổi thành phần, gồm phép lấy đối xứng qua CD và phép quay một góc  $-90^\circ$ . Vì vậy, ma trận của phép biến đổi ABCD thành A<sub>1</sub>B<sub>1</sub>C<sub>1</sub>D<sub>1</sub> là:

$$T_a = T_4 \times T_5 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 8 & 0 & 1 \end{pmatrix}$$

Tọa độ của tứ giác  $A_1B_1C_1D_1$  được tính theo công thức:

$$A_1B_1C_1D_1 \text{ tọa độ homogeneous} = ABCD \text{ tọa độ homogeneous} \cdot T_a$$

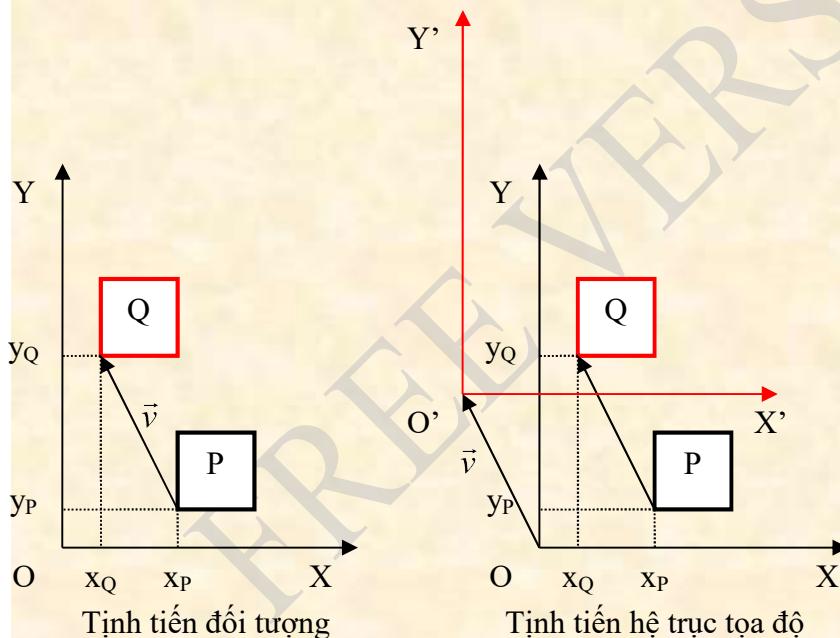
$$A_1B_1C_1D_1 \text{ tọa độ homogeneous} = \begin{pmatrix} 3 & 0 & 1 \\ 7 & 0 & 1 \\ 6 & 4 & 1 \\ 4 & 4 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 8 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 8 & -3 & 1 \\ 8 & -7 & 1 \\ 4 & -6 & 1 \\ 4 & -4 & 1 \end{pmatrix}$$

Vậy  $A_1=(8, -3); B_1=(8, -7); C_1=(4, -6); D_1=(4, -4)$

b) Sinh viên tự giải (xem như bài tập rèn luyện)

### 1.10. Biến đổi hệ trực tọa độ (hay biến đổi ngược)

Xét ví dụ biến đổi **tịnh tiến** hình vuông P theo vector  $v(1,2)$  thành hình vuông Q như thể hiện trong Hình 4.4.



Hình 4.4. Minh họa phép biến đổi thuận và biến đổi nghịch

Khi tịnh tiến hình P theo vector  $\vec{v}$  chúng ta được hình Q, đây được gọi là phép biến đổi thuận. Ngược lại, khi chúng ta tịnh tiến hệ trực tọa độ OXY theo vector  $\vec{v}$  để được hệ trực tọa độ mới O'X'Y'. Lúc này, tọa độ của hình P trong hệ trực cù OXY tương đương tọa độ của hình Q

trong hệ tọa độ mới O'X'Y'. Do đó:

$$P_{OXY} = Q_{O'X'Y'} \quad (4.2)$$

mà  $Q_{OXY} = P_{OXY} T$  với  $T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ v_x & v_y & 1 \end{pmatrix}$

suy ra:

$$Q_{OXY} \cdot T^{-1} = P_{OXY} \cdot T \cdot T^{-1} = P_{OXY} \quad (4.3)$$

từ (4.2) và (4.3) suy ra:

$$Q_{O'X'Y'} = Q_{OXY} \cdot T^{-1} \text{ với } T^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -v_x & -v_y & 1 \end{pmatrix}$$

Hay nói cách khác, một điểm Q có tọa độ  $(x_Q, y_Q)$  trong hệ tọa độ OXY sẽ có tọa độ là  $(x'_Q, y'_Q)$  trong hệ tọa độ O'X'Y' và

$$(x'_Q, y'_Q, 1) = (x_Q, y_Q, 1) \cdot T^{-1}$$

Một cách tổng quát:

- Giả sử gọi  $f_1$  là một phép biến đổi Homogeneous biến đổi P thành điểm Q với ma trận biến đổi là  $T_{\text{thuần}}$ .
- Cũng sử dụng phép biến đổi  $f_1$  nhưng lần này là để biến đổi hệ trực OXY thành hệ trực O'X'Y'. Thì ma trận biến đổi sẽ là  $T_{\text{nghịch}} = T^{-1}$ .

Và lúc đó một điểm P trong hệ trực OXY sẽ có giá trị tọa độ trong hệ trực O'X'Y' là:

$$P' = P \cdot T_{\text{nghịch}} = P \cdot T^{-1}$$

### 1.11. Cài đặt

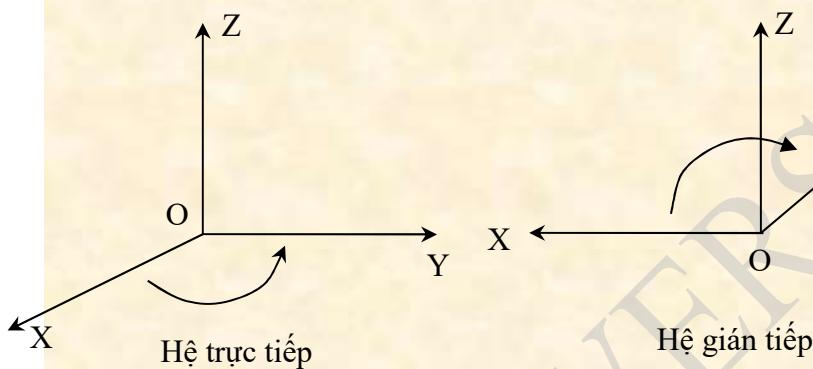
- Sinh viên cần xây dựng các hàm tìm ma trận của các phép biến đổi cơ bản trên. Mỗi hàm được cung cấp các đầu vào cần thiết và đầu ra là một ma trận biến đổi (mảng 2 chiều 3x3 phần tử).
- Xây dựng một hàm với đầu vào là một đa giác và một ma trận biến

đổi, điều ra là đa giác đã được biến đổi (bằng cách nhân các đỉnh của đa giác với ma trận biến đổi).

- Viết chương trình sử dụng hai hàm trên để minh họa các phép biến đổi cơ bản. Chương trình cho phép biến đổi (như tịnh tiến, quay, biến đổi tỷ lệ, biến dạng,...) các hình cơ bản như tam giác, tứ giác, ngũ giác,... Hoặc một hình bất kỳ nào đó như hình các kí tự A, F, H, K.

## 2. CÁC PHÉP BIẾN ĐỔI AFFINE 3D

### 2.1. Các hệ trục tọa độ



Hình 4.5. Hệ tọa độ 3 chiều trực tiếp và gián tiếp

Để định vị một điểm trong không gian, chúng ta có thể chọn nhiều hệ trục tọa độ:

- Hệ tọa độ trực tiếp*: Nếu chúng ta đang nhìn mặt phẳng (OXY) thì trục Z hướng vào mắt chúng ta (qui tắc bàn tay phải).
- Hệ tọa độ gián tiếp*: Ngược lại (qui tắc bàn tay trái).

Thông thường, chúng ta luôn luôn định vị một điểm trong không gian qua hệ trục tiếp.

Trong hệ tọa độ trực tiếp, chúng ta có công thức chuyển đổi tọa độ từ hệ tọa độ cầu sang hệ Đè-cát như sau:

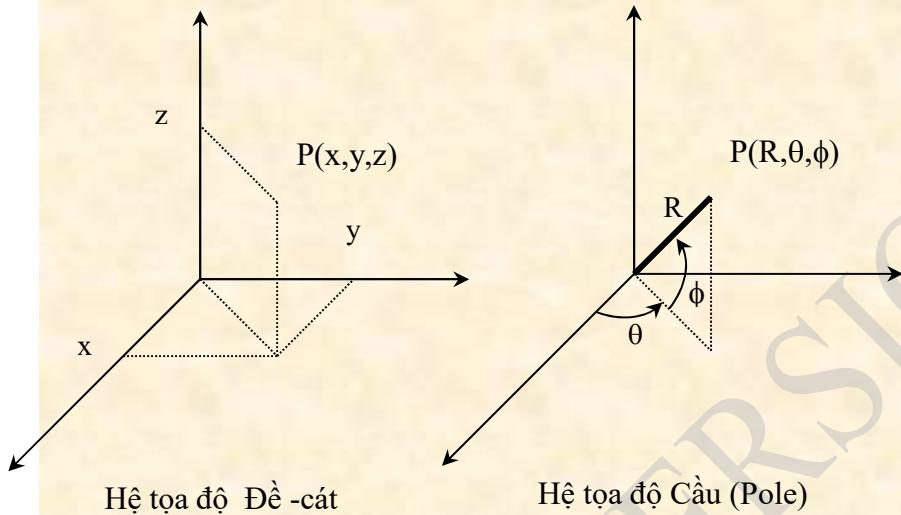
$$x = R \cdot \text{Cos}(\theta) \cdot \text{Cos}(\phi)$$

$$y = R \cdot \text{Sin}(\theta) \cdot \text{Cos}(\phi)$$

$$z = R \cdot \text{Sin}(\phi)$$

$$R^2 = x^2 + y^2 + z^2$$

Để thuận tiện cho việc tính toán chúng ta sẽ sử dụng hệ tọa độ thuận nhất (homogeneous). Từ đó, ma trận của các điểm phải có dạng  $(x,y,z,1)$ , và ma trận biến đổi là ma trận vuông  $4 \times 4$ .



Hình 4.6. Hệ tọa độ Dé-cát và hệ tọa độ cầu

## 2.2. Các công thức biến đổi

Phép biến đổi Affine 3D dạng thuận nhất có dạng:  $\mathbf{Q} = \mathbf{P} \cdot \mathbf{T}$

với  $\mathbf{P} = (P_x, P_y, P_z, 1)$  và  $\mathbf{Q} = (Q_x, Q_y, Q_z, 1)$ , còn ma trận biến đổi  $\mathbf{T}$  sẽ có dạng:

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} & 0 \\ t_{21} & t_{22} & t_{23} & 0 \\ t_{31} & t_{32} & t_{33} & 0 \\ tr_x & tr_y & tr_z & 1 \end{pmatrix}$$

### 2.2.1. Phép tịnh tiến

Ma trận của phép tịnh tiến theo vector  $\mathbf{Tr} = (tr_x, tr_y, tr_z)$  là:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tr_x & tr_y & tr_z & 1 \end{pmatrix}$$

### 2.2.2. Phép biến đổi tỷ lệ

Ma trận của phép biến đổi tỷ lệ trong không gian 3 chiều có dạng:

$$T = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Trong đó:

$s_x$ : Hệ số tỷ lệ theo trục X.

$s_y$ : Hệ số tỷ lệ theo trục Y.

$s_z$ : Hệ số tỷ lệ theo trục Z.

### 2.2.3. Phép đối xứng

Đối xứng qua mặt (OXY):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Đối xứng qua mặt (OXZ):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Đối xứng qua mặt (OYZ):

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

#### 2.2.4. Phép quay

Chúng ta nhận thấy rằng, nếu phép quay quay quanh một trục nào đó thì thành phần tọa độ ứng với trục đó của vật thể sẽ không thay đổi. Do đó, ta có ma trận của các phép quay như sau:

$$\text{Quay quanh trục Z: } \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

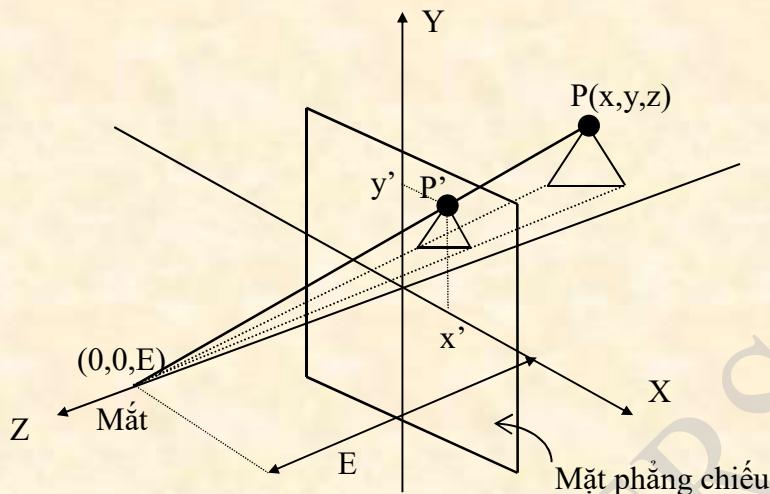
$$\text{Quay quanh trục X: } \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Quay quanh trục Y: } \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Chú ý: Tích của 2 ma trận không có tính chất giao hoán.

### 3. CÁC PHÉP CHIẾU VẬT THỂ TRONG KHÔNG GIAN LÊN MẶT PHẲNG

#### 3.1. Phép chiếu phối cảnh (Perspective)



Hình 4.7. Minh họa phép chiếu phối cảnh

Phép chiếu phối cảnh cho hình ảnh gần giống như khi chúng ta nhìn vật thể trong thế giới thực.

Để tìm hình chiếu  $P'(x',y')$  của điểm  $P(x,y,z)$ , chúng ta nối  $P$  với mắt (tâm chiếu). Giao điểm của đường này với mặt quan sát chính là  $P'$ .

Giả sử  $P$  nằm phía trước mắt, tức là  $P.z < E$ . Phương trình tham số của tia chiếu đi qua  $P$  được viết:

$$E.(1-t) + P.t \text{ với } t \in \mathbb{R}$$

$$\text{Hay viết cụ thể là: } Q(t) = (0,0,E).(1-t) + (x,y,z).t \quad (4.4)$$

Với  $Q(t)$  là điểm trên tia chiếu phụ thuộc vào tham số  $t$ . Khai triển (4.4) chúng ta được:

$$\begin{cases} x_{Q(t)} = 0(1-t) + x.t \\ y_{Q(t)} = 0(1-t) + y.t \\ z_{Q(t)} = E(1-t) + z.t \end{cases} \quad (4.5)$$

Giao điểm với mặt phẳng quan sát là điểm  $P'$  sẽ có thành phần  $z = 0$ .

nên phương trình (4.5) trở thành:

$$\begin{cases} x_{P'} = 0(1-t) + xt \\ y_{P'} = 0(1-t) + yt \\ z_{P'} = E(1-t) + zt = 0 \end{cases}$$

Từ  $E(1-t) + zt = 0$  suy ra  $t = \frac{1}{1-z/E}$ . Thay  $t$  vào 2 phương trình

trên chúng ta tính được:

$$x' = x_{P'} = \frac{x}{1-z/E} \quad \text{và} \quad y' = y_{P'} = \frac{y}{1-z/E}$$

#### Nhận xét:

- a) Phép chiếu phôi cảnh không giữ nguyên hình dạng của vật thể.
- b) Chỉ có những đường thẳng song song với nhau, đồng thời song song với mặt phẳng chiếu thì cho ảnh song song với nhau, còn tất cả các đường thẳng khác đều cho ảnh hội tụ đến tâm chiếu (mắt).
- c) Vật ở trước mặt phẳng chiếu thì được phóng lớn, sau mặt phẳng chiếu thì bị thu nhỏ. Vật ở xa thì trông nhỏ, ở gần thì trông lớn.

### 3.2. Phép chiếu song song

Nếu chúng ta chọn phương chiếu vuông góc với mặt phẳng chiếu (và cũng là vuông góc với mặt phẳng OXY) thì:

$$x' = x; \quad y' = y$$

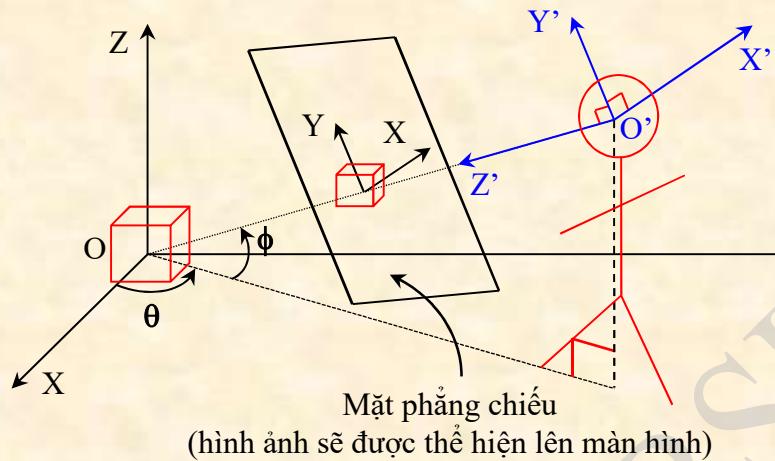
### 4. QUAN SÁT VẬT THỂ 3 CHIỀU VÀ QUAY HỆ QUAN SÁT

Trong phần này, chúng ta sẽ tìm hiểu các bước và các công thức cho phép thu được hình ảnh mô phỏng bài toán quan sát vật thể 3 chiều dưới một góc nhìn bất kỳ của người quan sát. Để giải quyết bài toán này, xem Hình 4.8, chúng ta cần tiến hành các bước sau:

1. Chuyển đổi số đo của đối tượng từ hệ tọa độ cục bộ, *là hệ tọa độ được sinh ra nhằm đo đặc đối tượng*, sang hệ tọa độ người quan sát.
2. Trong hệ tọa độ người quan sát, tiến hành chiếu vật thể lên mặt phẳng chiếu đặt vuông góc với hướng nhìn.
3. Vẽ hình chiếu của vật thể trên mặt phẳng chiếu lên màn hình đồ

họa, hình ảnh này thể hiện vật thể dưới một góc nhìn cụ thể của người quan sát.

#### 4.1. Biến đổi từ hệ trục cục bộ sang hệ trục người quan sát



Hình 4.8. Minh họa bài toán quan sát vật thể trong không gian 3 chiều

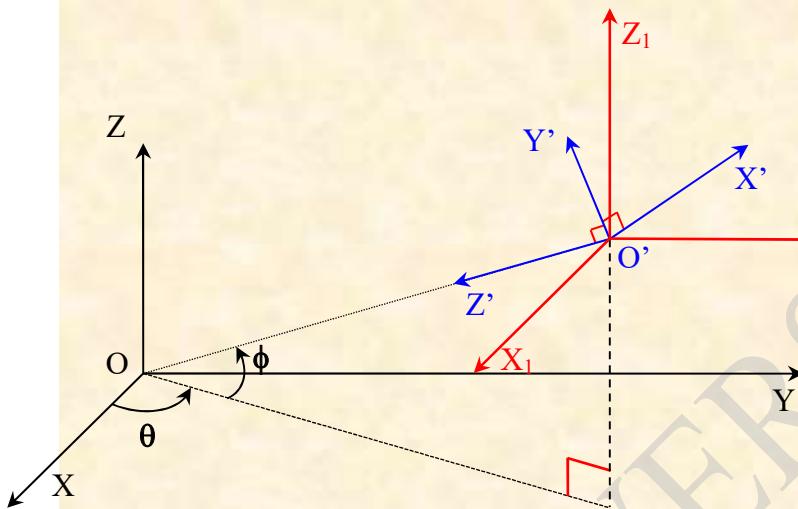
Khi mô tả việc quan sát một vật thể trong không gian chúng ta cần lưu ý các điểm sau:

- Vật thể được mô tả (hay *đo đạc*) trong hệ trục tọa độ quy chiếu của riêng nó và được gọi là hệ trục cục bộ OXYZ (hay *thuật ngữ* là *Local Coordinates system*).
- Mắt nằm ở gốc của một hệ tọa độ gián tiếp thứ hai tạm gọi là O'X'Y'Z' và được gọi là hệ tọa độ người quan sát (hay *hệ quan sát* - *View coordinate system*).
- Mặt phẳng chiếu vuông góc với đường thẳng OO'.
- Trục Z' của hệ tọa độ quan sát hướng vào gốc O.

Tiếp đến, chúng ta cần khảo sát phép chuyển đổi giá trị độ đo một điểm P(x,y,z) trong hệ tọa độ thứ nhất sang P'(x',y',z') trong hệ tọa độ thứ hai, rồi chuyển sang tọa độ trên mặt phẳng quan sát (mặt phẳng chiếu). Nói một cách khác, với mỗi điểm P đã được đo đạc trong hệ tọa độ OXYZ, chúng ta cần tìm số đo của nó nhưng trong hệ tọa độ O'X'Y'Z'?

Để trả lời được câu hỏi trên chúng ta cần tìm phép biến đổi ngược cho phép chuyển đổi hệ trục OXYZ thành hệ trục O'X'Y'Z'. Các bước thực hiện như sau:

**Bước 1:** Tịnh tiến hệ trục tọa độ từ gốc O đến gốc O', trong đó vector  $\overrightarrow{OO'} = (R.\cos(\theta).\cos(\phi), R.\sin(\theta).\cos(\phi), R.\sin(\phi))$  (xem Hình 4.9).



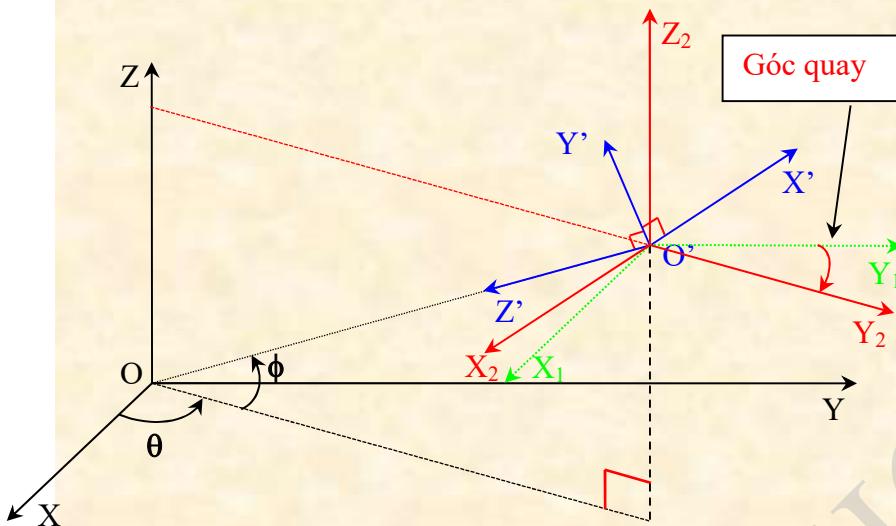
Hình 4.9. Tịnh tiến hệ trục OXYZ thành O'X<sub>1</sub>Y<sub>1</sub>Z<sub>1</sub>

Ma trận của phép tịnh tiến hệ trục tọa độ (chú ý đây là phép biến đổi nghịch nên ma trận biến đổi được lấy nghịch đảo của ma trận biến đổi thuận) theo vector  $\overrightarrow{OO'}$  là:

$$T_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -R.\cos(\theta).\cos(\phi) & -R.\sin(\theta).\cos(\phi) & -R.\sin(\phi) & 1 \end{pmatrix}$$

và hệ OXYZ biến đổi thành hệ O'X<sub>1</sub>Y<sub>1</sub>Z<sub>1</sub>.

**Bước 2:** Quay hệ O'X<sub>1</sub>Y<sub>1</sub>Z<sub>1</sub> một góc  $-\theta'$  (với  $\theta' = 90^\circ - \theta$ ) quanh trục Z<sub>1</sub>, dấu âm thể hiện góc quay thuận theo chiều kim đồng hồ. Phép quay này làm cho phần âm của trục Y<sub>1</sub> cắt trục Z (xem Hình 4.10).

Hình 4.10. Quay hệ trục  $O'X_1Y_1Z_1$  thành  $O'X_2Y_2Z_2$ 

Theo lý thuyết, ma trận biến đổi của phép quay đối tượng (điểm hay vật) quanh trục z một góc  $\alpha$  sẽ là:

$$T_{Z-Rotate} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Từ đó, ma trận ứng với phép biến đổi hệ trục (phép biến đổi nghịch) trong tình huống này sẽ là nghịch đảo của ma trận  $T_{Z-Rotate}$ :

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

Thay góc  $\alpha = -\theta' = -(90 - \theta)$  thì:

$$\sin(\alpha) = \sin(-(90 - \theta)) = -\sin(90^\circ - \theta) = -\cos(\theta)$$

$$\cos(\alpha) = \cos(-(90 - \theta)) = \cos(90^\circ - \theta) = \sin(\theta)$$

Từ đó, chúng ta thu được ma trận của phép biến đổi ở bước 2 này là:

$$T_2 = \begin{pmatrix} \sin(\theta) & \cos(\theta) & 0 & 0 \\ -\cos(\theta) & \sin(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

và hệ O'X<sub>1</sub>Y<sub>1</sub>Z<sub>1</sub> biến đổi thành hệ O'X<sub>2</sub>Y<sub>2</sub>Z<sub>2</sub>.

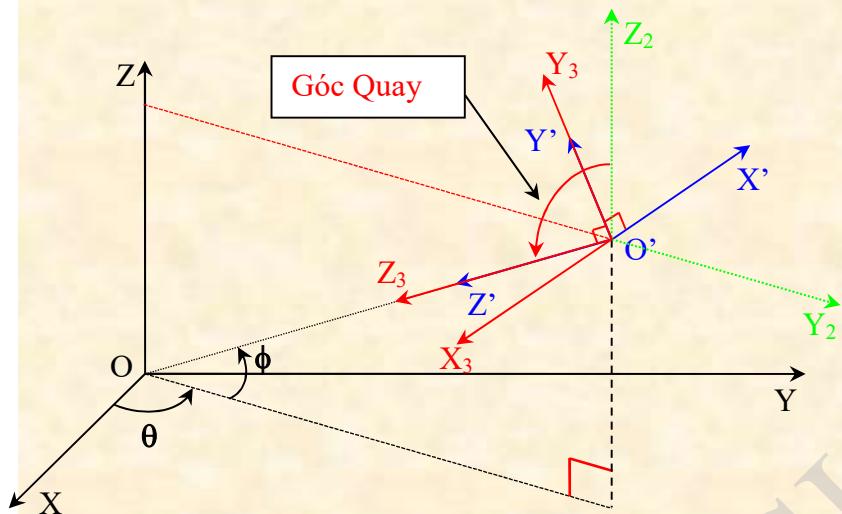
**Bước 3:** Quay hệ O'X<sub>2</sub>Y<sub>2</sub>Z<sub>2</sub> một góc ( $90^\circ + \phi$ ) quanh trục X<sub>2</sub>. Phép biến đổi này sẽ làm cho trục Z<sub>2</sub> hướng đến gốc O, và đồng thời khiến cho 2 mặt phẳng O'Y'Z' và O'Y<sub>2</sub>Z<sub>2</sub> trùng nhau (xem Hình 4.11).

Chúng ta có ma trận biến đổi thuận của phép quay đối tượng quanh trục X một góc  $\alpha$  là:

$$T_{X-Rotate} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Từ đó, suy ra ma trận biến đổi nghịch của phép biến đổi này là:

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

Hình 4.11. Quay hệ trục  $O'X_2Y_2Z_2$  thành  $O'X_3Y_3Z_3$ 

Thay góc  $\alpha = 90^\circ + \phi$ , ta có:

$$\cos(90^\circ + \phi) = -\sin(\phi) \text{ và } \sin(90^\circ + \phi) = \cos(\phi).$$

Từ đó, ta thu được ma trận của phép biến đổi trong bước này là:

$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\sin(\phi) & -\cos(\phi) & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

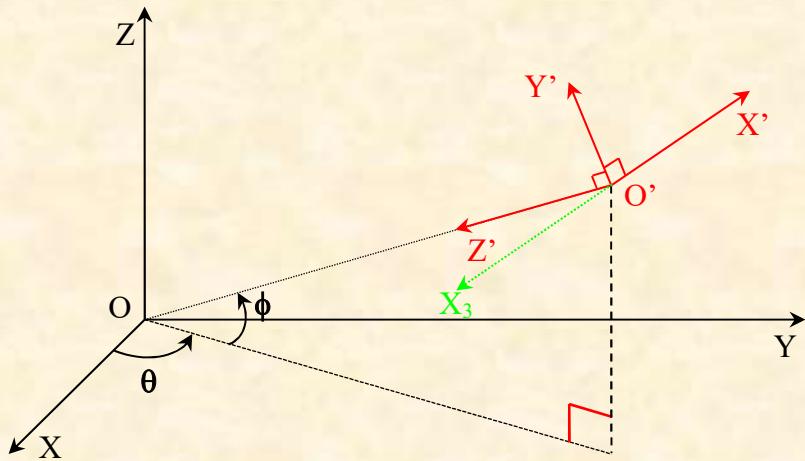
Lúc này, hệ  $O'X_2Y_2Z_2$  biến đổi thành hệ  $O'X_3Y_2Z_3$ .

#### Bước 4: Biến đổi hệ trực tiếp $O'X_3Y_3Z_3$ thành hệ gián tiếp $O'X'Y'Z'$ .

Trong bước này, chúng ta phải đổi hướng trục  $X_3$ , bằng cách đổi dấu thành phần tọa độ  $X$  chúng ta thu được ma trận biến đổi:

$$T_4 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

và hệ  $O'X_3Y_3Z_3$  biến đổi thành hệ tọa độ quan sát  $O'X'Y'Z'$ .



Hình 4.12. Đảo chiều trục  $X$  của hệ trục  $O'X_3Y_3Z_3$  để thu được hệ trục quan sát  $O'X'Y'Z'$

❖ Tóm lại:

- Các điểm trong không gian tọa độ cục bộ sẽ nhận trong hệ quan sát một tọa độ có dạng:

$$(x', y', z', 1) = (x, y, z, 1) \times T_1 \times T_2 \times T_3 \times T_4$$

Gọi  $T = T_1 \times T_2 \times T_3 \times T_4$ , là ma trận chuyển đổi từ hệ tọa độ cục bộ OXYZ sang hệ tọa độ người quan sát, từ đó:

$$T = \begin{pmatrix} -\sin(\theta) & -\cos(\theta).\sin(\phi) & -\cos(\theta).\cos(\phi) & 0 \\ \cos(\theta) & -\sin(\theta).\sin(\phi) & -\sin(\theta).\cos(\phi) & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & 0 & R & 1 \end{pmatrix}$$

Cuối cùng, chúng ta có công thức chuyển đổi tọa độ:

$$(x', y', z', 1) = (x, y, z, 1).T$$

hoặc dưới dạng phương trình:

$$x' = -x.\sin(\theta) + y.\cos(\theta)$$

$$y' = -x.\cos(\theta).\sin(\phi) - y.\sin(\theta).\sin(\phi) + z.\cos(\phi)$$

$$z' = -x.\cos(\theta).\cos(\phi) - y.\sin(\theta).\cos(\phi) - z.\sin(\phi) + R$$

- Trong tình huống ngược lại, khi biết giá trị số đo (tọa độ) của một điểm P đo trong hệ trục O'X'Y'Z' và cần xác giá trị số đo trong hệ trục OXYZ thì công thức sẽ là:

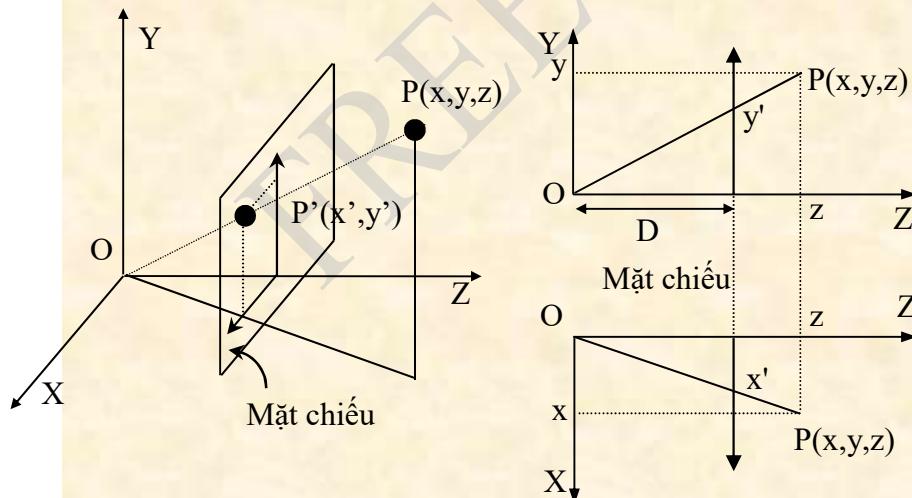
$$\begin{aligned} P_{O'X'Y'Z'} &= P_{OXYZ} \cdot T \\ \Leftrightarrow P_{O'X'Y'Z'} \cdot T^{-1} &= P_{OXYZ} \cdot T \cdot T^{-1} \\ \Leftrightarrow P_{OXYZ} &= P_{O'X'Y'Z'} \cdot T^{-1} \end{aligned}$$

Trong đó,  $T^{-1}$  là ma trận nghịch đảo của  $T$  với giá trị:

$$T^{-1} = \begin{pmatrix} -\sin(\theta) & \cos(\theta) & 0 & 0 \\ -\cos(\theta).\sin(\phi) & -\sin(\theta).\sin(\phi) & \cos(\phi) & 0 \\ -\cos(\theta).\cos(\phi) & -\sin(\theta).\cos(\phi) & -\sin(\phi) & 0 \\ R.\cos(\theta).\cos(\phi) & R.\sin(\theta).\cos(\phi) & R.\sin(\phi) & 1 \end{pmatrix}$$

Giai đoạn tiếp theo, chúng ta cần tìm hiểu phép chiếu cho phép chiếu hình ảnh của vật thể trong hệ quan sát lên mặt phẳng chiếu 2 chiều. Hình ảnh 2 chiều thu được trên mặt phẳng chiếu sau đó được biểu diễn (ánh xạ) lên màn hình máy tính giúp người sử dụng có thể quan sát được vật thể dưới một góc nhìn cụ thể. Nếu muốn thay đổi góc nhìn, cần thay đổi các thông số liên quan như: góc  $\theta$ , góc  $\phi$ , khoảng cách  $R$ ,...

## 4.2. Phép chiếu phối cảnh



Hình 4.13. Phép chiếu phối cảnh trong bài toán quan sát vật thể 3 chiều

Lúc này, chúng ta quan tâm đến bài toán chiếu một điểm P trong hệ tọa độ quan sát lên mặt phẳng chiếu đặt trước mặt người quan sát (*tức vuông góc với trục z*) và cách mắt một khoảng là D. Lúc này, bài toán cô lập lại chỉ còn một hệ tọa độ 3 chiều (hệ tọa độ quan sát) và một hệ tọa độ 2 chiều (*hệ tọa độ trên mặt phẳng chiếu*), vì vậy để cho dễ hiểu (và quen thuộc) chúng ta tạm gọi hệ tọa độ quan sát là hệ OXYZ, và hệ tọa độ trên mặt phẳng chiếu là hệ O'X'Y'.

Gọi D là khoảng cách từ mặt phẳng chiếu đến mắt người quan sát (cũng là gốc của hệ tọa độ quan sát).

Cho điểm P có số đo trong hệ tọa độ quan sát là  $(x,y,z)$ , gọi  $P'$  là hình chiếu của P lên mặt phẳng chiếu với tọa độ là  $(x',y')$ .

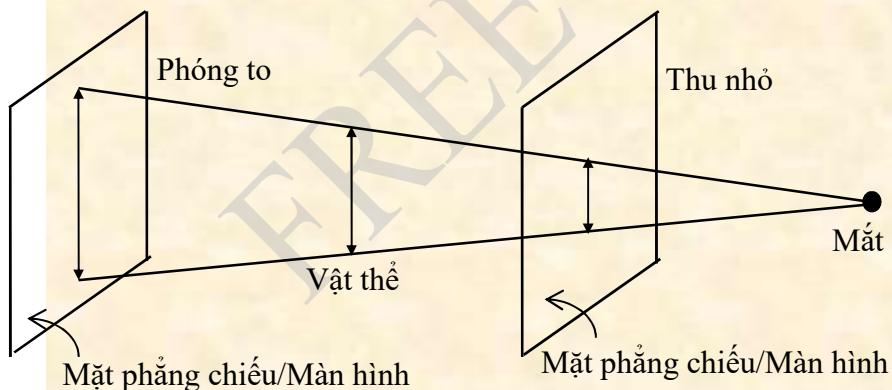
Xét các tam giác đồng dạng, chúng ta có:

$$x'/x = D/z \quad \text{và} \quad y'/y = D/z$$

Từ đó, chúng ta suy ra công thức tính tọa độ điểm chiếu:

$$x' = x.D/z \quad \text{và} \quad y' = y.D/z$$

**Chú ý:** Nói một cách khác, tọa độ x và y của điểm P được nhân với một hệ số  $D/z$  để thu được tọa độ điểm chiếu  $P'$ . Vì vậy, nếu  $z > D$  thì kết quả của phép chiếu là thu nhỏ vật thể (hay hội tụ). Ngược lại, thì kết quả được phóng lớn.



Hình 4.14. Minh họa tính chất của phép chiếu phối cảnh

### 4.3. Phép chiếu song song

Do mặt phẳng chiếu song song với mặt phẳng  $O'X'Y'$  của hệ trục

quan sát, vì vậy phép chiếu song song trong tình huống này trở thành phép chiếu vuông góc. Từ đó, chúng ta dễ dàng có công thức sau:

$$x' = x \quad \text{và} \quad y' = y$$

**Chú ý:** Có 4 tham số ảnh hưởng đến phép chiếu vật thể 3D gồm: Góc  $\theta$ , góc  $\phi$ , khoảng cách  $R$  từ  $O$  đến  $O'$ , và khoảng cách  $D$  từ  $O'$  đến mặt phẳng quan sát. Như vậy:

- Tăng giảm  $\theta$  sẽ giúp chúng ta có thể di chuyển hệ quan sát quanh trục  $OZ$  để quan sát bốn mặt bên của vật thể.
- Tăng giảm  $\phi$  sẽ giúp chúng ta có thể di chuyển quanh gốc  $O$  theo hướng vuông góc với mặt phẳng  $OXY$  để quan sát được mặt trên hay mặt dưới của vật thể.
- Tăng giảm  $R$  để quan sát vật từ xa hay gần.
- Tăng giảm  $D$  để phóng to hay thu nhỏ ảnh.

#### 4.4. Cài đặt

- Sinh viên cần xây dựng một hàm cho phép chuyển tọa độ của đối tượng từ hệ tọa độ trực tiếp sang hệ tọa độ quan sát.
- Xây dựng một hàm cho phép chiếu vật thể từ hệ tọa độ quan sát lên mặt phẳng quan sát theo phép chiếu song song hay phô cản rồi vẽ kết quả chiếu lên màn hình.
- Xây dựng chương trình sử dụng 2 hàm trên để minh họa hình ảnh của một hình hộp. Nâng cao hơn sinh viên cần theo dõi các góc  $\theta$  và  $\phi$  để chương trình có thể thể hiện được hình hộp từ nhiều góc độ.

### 5. BÀI TẬP CUỐI CHƯƠNG

1. Cài đặt giải thuật xén một đoạn thẳng vào một hình chữ nhật có cạnh không song song với trục tọa độ.
2. Viết chương trình vẽ một Ellipse có các trục không song song với hệ trục tọa độ.
3. Dựa vào bài tập 2, hãy mô phỏng quá trình quay của một Ellipse xung quanh tâm của nó.

- 
- 4. Mô phỏng quá trình quay, tịnh tiến của một hình bất kỳ trong mặt phẳng quanh trục tọa độ.
  - 5. Mô phỏng chuyển động của trái đất xung quanh mặt trời, đồng thời mô tả chuyển động của mặt trăng xung quanh trái đất.
  - 6. Viết chương trình vẽ đồng hồ đang hoạt động.
  - 7. Viết chương trình vẽ các khối đa diện đều trong không gian.

FREE VERSION

## Chương 5

# MÔ HÌNH WIREFRAME

Thế giới quanh chúng ta hầu hết đều là những đối tượng 3 chiều (3D, gồm chiều rộng, chiều dài và chiều cao). Bằng những kỹ thuật như vẽ, chụp hình, quay phim,... chúng ta thường cố gắng để biểu diễn hình ảnh thực 3 chiều của đối tượng dưới những góc nhìn khác nhau bằng các hình vẽ trên mặt phẳng hai chiều của giấy, khung vẽ hay màn ảnh,... để thể hiện lại các đối tượng. Các hình ảnh đó phải tuân theo một số quy luật về phối cảnh, sáng tối nhằm giúp người xem có thể tưởng tượng lại hình ảnh mong muốn.

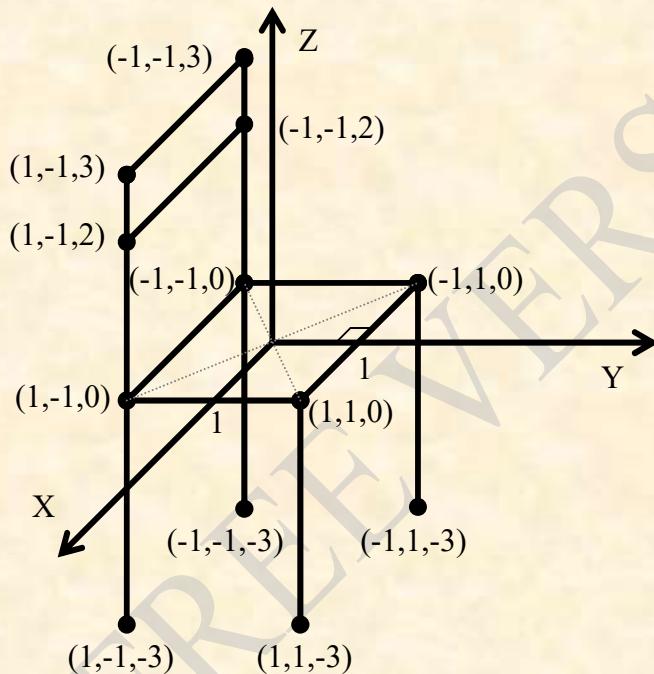
Ngày nay, dưới sự tiến bộ của máy tính điện tử thì nhu cầu thể hiện các đối tượng bằng máy tính trở thành một nhu cầu thiết yếu. Thể hiện các đối tượng thực bằng máy tính, đặc biệt là biểu diễn các đối tượng 3 chiều là một công việc đem lại lợi ích rất lớn, bởi nó dễ dàng cho chúng ta những góc nhìn khác nhau, những cách thể hiện, những biến đổi và nhiều tác động khác một cách hiệu quả mà không mất nhiều thời gian và tiền của. Trước đây, các kiến trúc sư thường làm việc trên bàn giấy với công cụ là bút vẽ và giấy, do đó công việc rất khó nhọc. Đôi khi, một bảng vẽ phải được tiến hành vẽ lại toàn bộ chỉ vì một vài sai sót nhỏ, hay vì nhu cầu sửa đổi một số chi tiết trên bản vẽ. Ngày nay, với sự trợ giúp đặc lực của máy tính, các kiến trúc sư có thể nhanh chóng thể hiện các ý tưởng của mình trên máy tính, công việc thể hiện bảng vẽ trở nên dễ dàng hơn, các sai sót hay sửa đổi được thực hiện một cách dễ dàng mà không nhất thiết phải thực hiện vẽ lại toàn bộ bảng vẽ. Ngoài ra, máy tính còn có thể thể hiện ra nhiều bảng vẽ khác nhau cho cùng một đối tượng dưới nhiều góc nhìn, và nhiều chức năng tiện ích khác đi kèm như: Tính kết cấu lực, tạo phối cảnh, tạo đoạn phim mô tả công trình,...

Trong phần này, chúng ta sẽ giới thiệu một số khái niệm và kỹ thuật trong việc biểu diễn các đối tượng 3 chiều trên máy tính, bắt đầu từ các đối tượng đơn giản như các mặt, các hình khối, các đa diện,...

## 1. MÔ HÌNH WIREFRAME

Mô hình Wireframe, hay còn gọi là mô hình khung dây, lưu trữ thông tin về hình dáng (hay bộ khung) của đối tượng. Thông tin lưu trữ được tổ chức thành 2 danh sách, một *danh sách đỉnh*, và một *danh sách cạnh* được tạo nên từ các đỉnh *chứa trong danh sách đỉnh*. Danh sách đỉnh lưu giữ tọa độ các đỉnh của đối tượng mà mỗi đỉnh bao gồm các thành phần tọa độ ( $x, y, z$ ), danh sách cạnh lưu giữ danh sách các cặp thứ tự điểm tạo nên cạnh mà tọa độ của các điểm này chứa trong danh sách đỉnh.

Ví dụ: Để lưu trữ hình dáng một chiếc ghế gỗ thô sơ như hình vẽ sau:



Hình 5.1. Minh họa công đoạn số hóa đối tượng 3 chiều

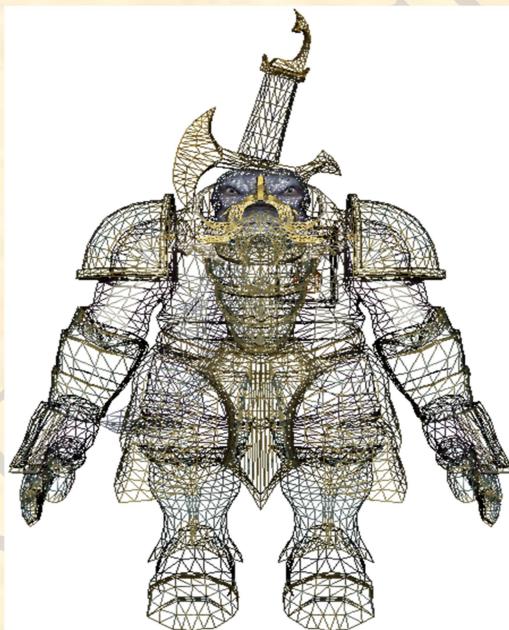
Thông tin mô tả đối tượng chiếc ghế là 2 danh sách sau:

Bảng 5.1. Danh sách thông tin lưu trữ  
theo mô hình WireFrame của chiếc ghế

Danh sách đỉnh				Danh sách cạnh		
Sđt	X	Y	Z	Sđt	đỉnh 1	đỉnh 2
1	1	1	-3	1	1	5

2	-1	1	-3
3	-1	-1	-3
4	1	-1	-3
5	1	1	0
6	-1	1	0
7	-1	-1	0
8	1	-1	0
9	1	-1	2
10	-1	-1	2
11	1	-1	3
12	-1	-1	3

2	2	6
3	3	12
4	4	11
5	5	6
6	6	7
7	7	8
8	8	5
9	9	10
10	11	12



Hình 5.2. Mô hình wireframe cho một nhân vật trong game

Có nhiều cách thức để tổ chức lưu trữ và xử lý thông tin của đối tượng theo mô hình WireFrame trong một ứng dụng và mỗi hình thức đều có những ưu nhược điểm riêng. Ở đây, chúng ta sẽ từng bước xây dựng một lớp (class) có tên CWireFrame, với các thuộc tính và các phương thức có thể đáp ứng được bài toán mô phỏng đối tượng 3 chiều trong không gian trên máy tính với các chức năng cơ bản như: Hiển thị

hình ảnh đối tượng 3D lên mặt phẳng 2 chiều của thiết bị đồ họa (màn hình), thay đổi vị trí quan sát, thay đổi kích cỡ của đối tượng,... Chi tiết được trình bày ở mục 2.3 dưới đây.

## 2. VẼ HÌNH DỰA TRÊN DỮ LIỆU KIẾU WIREFRAME VỚI CÁC PHÉP CHIỀU

Vẽ một đối tượng ở dạng mô hình khung thì một cách đơn giản là chúng ta vẽ các cạnh trong danh sách. Song, để vẽ một cạnh trong không gian 3 chiều lên mặt phẳng 2 chiều thì chúng ta cần thực hiện các phép chiếu để chiếu chúng lên mặt phẳng chiếu, hay nói một cách cụ thể là chúng ta cần chiếu các điểm đầu mút của các cạnh lên mặt phẳng chiếu rồi sau đó nối chúng lại.

Kỹ thuật chung để vẽ một đoạn thẳng 3D:

- Chiếu mỗi điểm đầu mút thành từng điểm 2 chiều.
- Vẽ đoạn thẳng nối hai điểm chiếu chúng ta được đoạn thẳng chiếu.

Điều này làm được, bởi vì các phép chiếu nói chung bảo toàn đường thẳng. Vấn đề còn lại là đi tìm hình chiếu của một điểm trong không gian 3 chiều lên mặt phẳng chiếu. Chúng ta xem xét hai phép chiếu dưới đây:

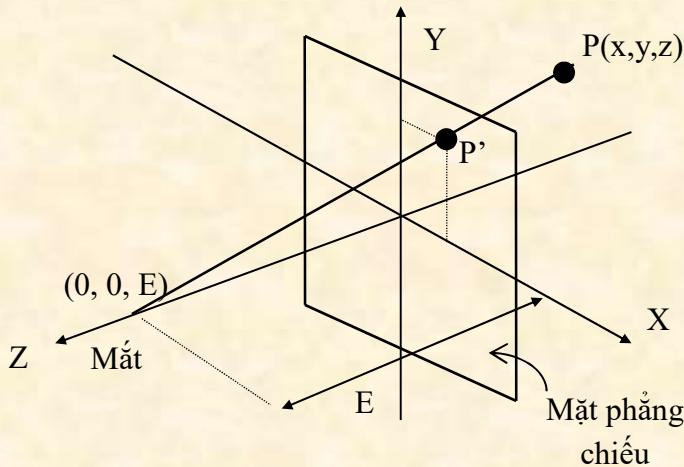
### 2.1. Phép chiếu trực giao đơn giản

Chiếu một điểm  $P(x,y,z)$  lên một mặt phẳng thì đơn giản nhất là chọn mặt phẳng chiếu là mặt phẳng tọa độ. Ví dụ như chiếu lên mặt phẳng OXY, khi đó chúng ta chỉ việc bỏ đi thành phần  $z$  trong tọa độ của điểm và giữ nguyên các thành phần  $x, y$  chúng ta được tọa độ của điểm chiếu trên mặt phẳng OXY.

### 2.2. Phép chiếu phối cảnh đơn giản

Phép chiếu phối cảnh được thực hiện khá đơn giản, song nó cho chúng ta một cái nhìn khá tự nhiên hơn về đối tượng 3D so với phép chiếu trực giao. Chúng ta đã làm quen với cách nhìn phối cảnh trong đời sống hàng ngày.

Phép chiếu phối cảnh phụ thuộc vào vị trí tương đối của mắt và mặt phẳng chiếu. Ví dụ như trong Hình 5.3:



Hình 5.3. Cách bố trí một phép chiếu phối cảnh đơn giản

Như đã trình bày trong chương IV (Các phép biến đổi hình học) chúng ta có hình chiếu phối cảnh  $P'$  của  $P$  lên mặt phẳng OXY có các thành phần tọa độ là:

$$x' = \frac{x}{1 - \frac{z}{E}} \text{ và } y' = \frac{y}{1 - \frac{z}{E}}$$

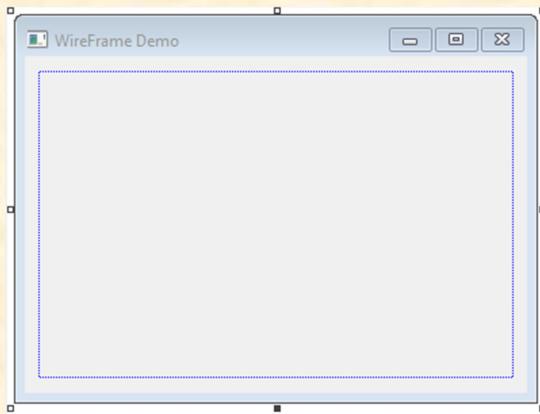
Dễ dàng thấy rằng phép chiếu phối cảnh gần giống với phép chiếu trực giao trước đây, ngoại trừ  $x$  và  $y$  được nhân một hệ số  $t=1/(1-z/E)$ . Hệ số này thu thuộc vào khoảng cách tương đối  $E$  giữa mắt nhìn và mặt phẳng chiếu, và nó sẽ làm cho vật thể (hay thành phần) ở gần mắt thì có hình dáng to ra, còn những vật thể (hay thành phần) ở xa thì có hình dáng nhỏ lại.

### 2.3. Cài đặt thực nghiệm cho mô hình wireframe

Chúng ta cần kết hợp phần kiến thức ở chương 4, mục 4. “Quan sát vật thể 3 chiều và quay hệ quan sát”, với nội dung của chương này. Từ đó, xây dựng một chương trình cho phép mô phỏng việc quan sát một vật thể 3 chiều. Chương trình cho phép người sử dụng quan sát vật thể dưới nhiều góc độ khác nhau thông qua thao tác rê chuột. Sau đây là các bước thiết kế chương trình:

**Bài thực nghiệm số 3:**

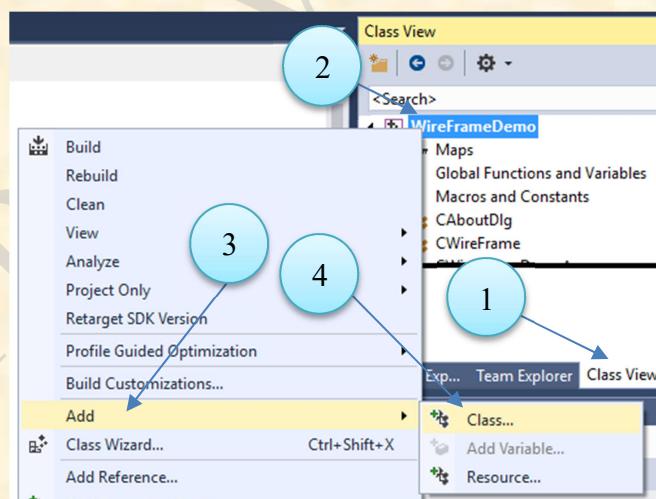
- ❖ Bước 1: Tạo một chương trình dạng Dialog based như hướng dẫn ở mục 2.3 với tên gọi WireFrameDemo.
- ❖ Bước 2: Thiết kế giao diện như hình sau:



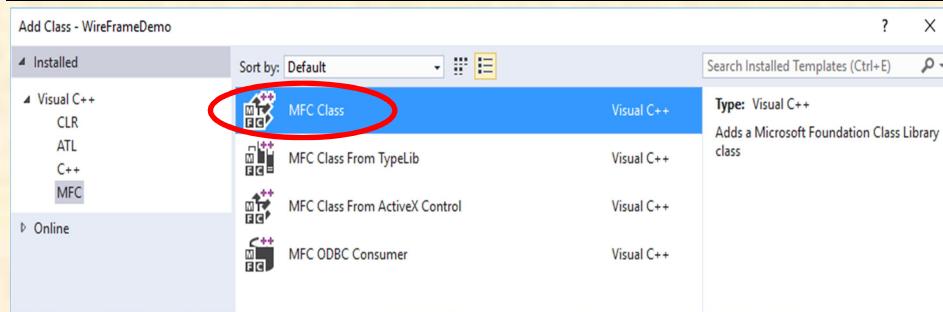
Hình 5.4. Giao diện chương trình WireFrameDemo

- ❖ Bước 3: Tạo một MFC Class với tên gọi CWireFrame:

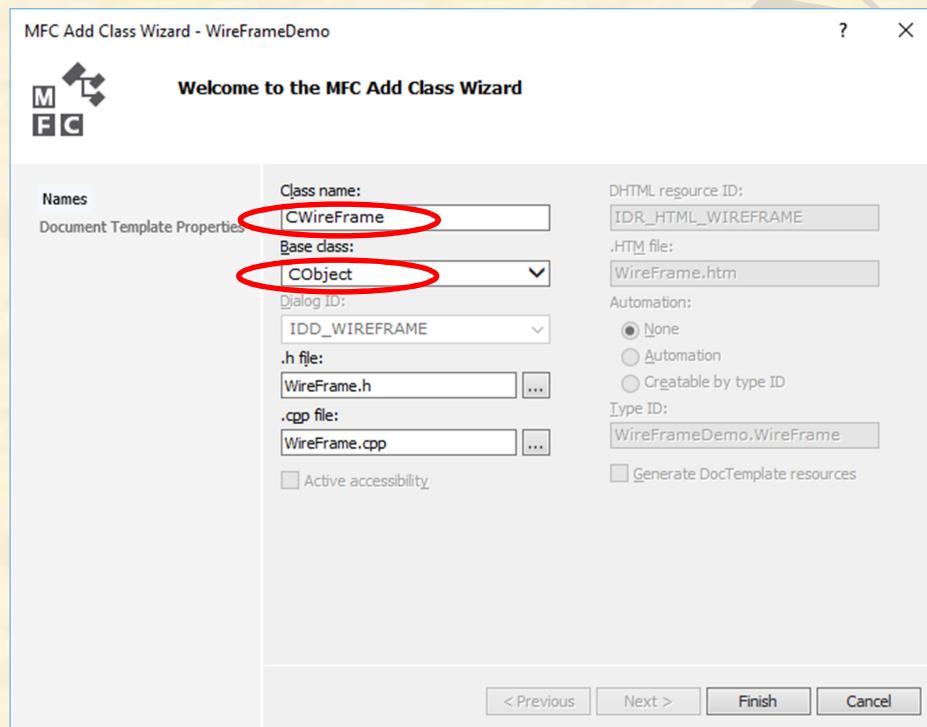
Các bước thực hiện được mô tả qua các hình ảnh dưới đây:



Hình 5.5. Menu ngữ cảnh cho phép tạo một class cho project



Hình 5.6. Tạo một MFC Class



Hình 5.7. Thiết lập tham số cho lớp CWireFrame

+ File header của lớp CWireFrame được khai báo với các nội dung chính như dưới đây:

```
struct Vertex // Lưu trữ giá trị tọa độ một đỉnh của đối tượng
{
    double x, y, z;
};

struct Edge /* Lưu trữ thông tin một cạnh, bao gồm 2 giá trị
```

```

chỉ mục đính */
{
    int VertexIndex1;
    int VertexIndex2;
};

class CWireFrame : public CObject
{
public:
    CWireFrame();           // Hàm khởi dụng đối tượng
    virtual ~CWireFrame(); // Hàm hủy đối tượng
    int NumVertex;          // Số đỉnh của đối tượng
    Vertex *ListVertex;     // Danh sách đỉnh (được cấp phát động)
    int NumEdge;            // Số cạnh của đối tượng
    Edge *ListEdge;          // Danh sách cạnh (được cấp phát động)
    double Zoom; // Hệ số co giãn (phóng Lớn hay thu nhỏ) đối tượng

    Vertex *ListVertexInViewCoordinates = NULL; /* Danh sách chứa
tọa độ của đối tượng đó trong hệ tọa độ người quan sát. Được cấp
phát động, khi không cần thiết có thể giải phóng bộ nhớ nhưng cần
nhớ gán lại giá trị NULL sau khi giải phóng */

    void CreateChair(); /* Khởi tạo thông tin cho đối tượng, mô tả
một chiếc ghế */
    void TransVertexToViewCoordinates(unsigned int R, unsigned int
DeltaAngle, unsigned int PhiAngle); /* Hàm chuyển đổi số đo các đỉnh
của đối tượng từ hệ tọa độ cục bộ sang hệ tọa độ người quan sát, dữ
liệu được chứa vào danh sách ListVertexInViewCoordinates */
    void DrawObject(CDC *p_DC, CRect R); /* Hàm vẽ đối tượng trong
hệ tọa độ người quan sát lên màn hình */

private:
    double LUT_Cos[360]; // Bảng tra giá trị hàm Cos
    double LUT_Sin[360]; // Bảng tra giá trị hàm Sin
};

```

+ File source (mã nguồn) của lớp CWireFrame được mã hóa như sau:

```

#include "stdafx.h"
#include "WireFrameDemo.h"
#include "WireFrame.h"

```

```
const double PI = 3.14159265358979323846;

// CWireFrame

CWireFrame::CWireFrame()
{
    for (unsigned int i = 0; i<360; i++)
    {
        double Angle = i*PI / 180;
        this->LUT_Cos[i] = cos(Angle);
        this->LUT_Sin[i] = sin(Angle);
    }
}

CWireFrame::~CWireFrame()
{
    delete[] ListVertex;
    delete[] ListEdge;

    if (ListVertexInViewCoordinates != NULL)
    {
        delete[] ListVertexInViewCoordinates;
    }
}

void CWireFrame::CreateChair()
{
    NumVertex = 12;
    ListVertex = new Vertex[NumVertex];
    ListVertex[0].x= 1; ListVertex[0].y = 1;   ListVertex[0].z =-3;
    ListVertex[1].x=-1; ListVertex[1].y = 1;   ListVertex[1].z =-3;
    ListVertex[2].x=-1; ListVertex[2].y =-1;  ListVertex[2].z =-3;
    ListVertex[3].x= 1; ListVertex[3].y =-1;  ListVertex[3].z =-3;
    ListVertex[4].x= 1; ListVertex[4].y = 1;   ListVertex[4].z = 0;
    ListVertex[5].x=-1; ListVertex[5].y = 1;   ListVertex[5].z = 0;
    ListVertex[6].x=-1; ListVertex[6].y =-1;  ListVertex[6].z = 0;
    ListVertex[7].x= 1; ListVertex[7].y =-1;  ListVertex[7].z = 0;
    ListVertex[8].x= 1; ListVertex[8].y =-1;  ListVertex[8].z = 2;
    ListVertex[9].x=-1; ListVertex[9].y =-1;  ListVertex[9].z = 2;
    ListVertex[10].x= 1;  ListVertex[10].y=-1; ListVertex[10].z =
3;
```

```

ListVertex[11].x=-1; ListVertex[11].y=-1; ListVertex[11].z =
3;

NumEdge = 10;
ListEdge = new Edge[NumEdge];
ListEdge[0].VertexIndex1 = 0; ListEdge[0].VertexIndex2 = 4;
ListEdge[1].VertexIndex1 = 1; ListEdge[1].VertexIndex2 = 5;
ListEdge[2].VertexIndex1 = 2; ListEdge[2].VertexIndex2 = 11;
ListEdge[3].VertexIndex1 = 3; ListEdge[3].VertexIndex2 = 10;
ListEdge[4].VertexIndex1 = 4; ListEdge[4].VertexIndex2 = 5;
ListEdge[5].VertexIndex1 = 5; ListEdge[5].VertexIndex2 = 6;
ListEdge[6].VertexIndex1 = 6; ListEdge[6].VertexIndex2 = 7;
ListEdge[7].VertexIndex1 = 7; ListEdge[7].VertexIndex2 = 4;
ListEdge[8].VertexIndex1 = 8; ListEdge[8].VertexIndex2 = 9;
ListEdge[9].VertexIndex1 = 10; ListEdge[9].VertexIndex2 = 11;
Zoom = 30;
}

void CWireFrame::TransVertexToViewCoordinates(unsigned int R,
unsigned int DeltaAngle, unsigned int PhiAngle)
{
    if (ListVertexInViewCoordinates == NULL)
    {
        this->ListVertexInViewCoordinates = new Vertex[this-
>NumVertex];
    }

    double x, y, z;

    for (unsigned int i = 0; i<this->NumVertex; i++)
    {
        x = ListVertex[i].x*this->Zoom;
        y = ListVertex[i].y*this->Zoom;
        z = ListVertex[i].z*this->Zoom;

        ListVertexInViewCoordinates[i].x =
-x*LUT_Sin[DeltaAngle] + y*LUT_Cos[DeltaAngle];

        ListVertexInViewCoordinates[i].y =
-x*LUT_Cos[DeltaAngle] * LUT_Sin[PhiAngle] -
y*LUT_Sin[DeltaAngle] * LUT_Sin[PhiAngle] + z*LUT_Cos[PhiAngle];

        ListVertexInViewCoordinates[i].z = -x*LUT_Cos[DeltaAngle] *

```

```

LUT_Cos[PhiAngle] - y*LUT_Sin[DeltaAngle] *
LUT_Cos[PhiAngle] - z*LUT_Sin[PhiAngle] + R;
};

}

void CWireFrame::DrawObject(CDC *p_DC, CRect R)
{
    long X_C = R.CenterPoint().x;
    long Y_C = R.CenterPoint().y;
    int x1, y1, x2, y2;

    CPen RedPen(PS_SOLID, 1, RGB(255, 0, 0));
    CPen* OldPen = p_DC->SelectObject(&RedPen);

    for (unsigned int i = 0; i < this->NumEdge; i++)
    {
        x1 =
int(ListVertexInViewCoordinates[ListEdge[i].VertexIndex1].x + 0.5)
+ X_C;
        y1 =
int(ListVertexInViewCoordinates[ListEdge[i].VertexIndex1].y + 0.5)
+ Y_C; // Đảo chiều trục Y rồi tính tiến đến tâm của vùng R

        x2 =
int(ListVertexInViewCoordinates[ListEdge[i].VertexIndex2].x + 0.5)
+ X_C;
        y2 =
int(ListVertexInViewCoordinates[ListEdge[i].VertexIndex2].y + 0.5)
+ Y_C;

        p_DC->MoveTo(x1, y1);
        p_DC->LineTo(x2, y2);
    }

    p_DC->SelectObject(OldPen);
}

```

- ❖ Bước 4: Lập trình xử lý các sự kiện chuột nhằm thể hiện hình ảnh của đối tượng (*là chiếc ghé*) dưới góc nhìn của người quan sát:
  - + Các khai báo **cần bő sung** trong file WireFrameDemoDlg.h là:

```

1. #include "WireFrame.h" /* Thêm khai báo để có thể sử dụng Lớp
CWireFrame */

2. public:

    CWireFrame WF_Object; /* Khai báo biến đối tượng thuộc Lớp
WireFrame để Lưu trữ và xử lý với đối tượng 3 chiều theo mô hình
wireframe */

    int R=50, DeltaAngle=30, PhiAngle=35; /* Khai báo các thông
số: khoảng cách (R), góc θ (DeltaAngle), và góc φ (PhiAngle) */

    bool MouseDown=false; // Báo hiệu phím trái chuột có được bấm
xuống?

    CPoint Position_MouseDown; /* Lưu trữ vị trí tọa độ (Position)
trên cửa sổ mà tại đó người sử dụng đã bấm giữ nút chuột trái */

    int MouseDown_DeltaAngle, MouseDown_PhiAngle; /* Lưu giữ giá
trị các góc θ và φ tại thời điểm bắt đầu bấm nút trái chuột, phục
vụ cho việc tính toán giá trị các góc θ và φ mới khi di chuyển
chuột (mouse move) */

```

+ Add một hàm có dạng `void ViewObject();` vào trong lớp `CWireFrameDemoDlg` nhằm cho phép thể hiện hình ảnh của đối tượng lên cửa sổ chương trình. Mã lệnh như sau:

```

void CWireFrameDemoDlg::ViewObject()
{
    CDC *p_DC = this->GetDC(); /*Lấy device context của cửa sổ
chương trình */
    CRect Rect;
    this->GetClientRect(&Rect); /* Lấy tọa độ của vùng không gian
bên trong cửa sổ phục vụ cho việc vẽ đối tượng */

    WF_Object.TransVertexToViewCoordinates(R, DeltaAngle,
PhiAngle); /* Chuyển số đo của đối tượng từ hệ tọa độ cục bộ sang
hệ quan sát */
    p_DC->FillSolidRect(Rect, RGB(255, 255, 255)); /* Xóa trắng
cửa sổ bằng màu trắng */

    WF_Object.DrawObject(p_DC, Rect); // Vẽ đối tượng lên cửa sổ
}

```

+ Thêm đoạn mã lệnh sau vào cuối hàm đáp ứng sự kiện `CWireFrameDemoDlg::OnInitDialog()` như sau:

```
BOOL CWireFrameDemoDlg::OnInitDialog()
{
    ... /* Các câu lệnh mặc định có sẵn trong hàm OnInitDialog() do trình Wizard tạo ra */

    // TODO: Add extra initialization here
    WF_Object.CreateChair(); /* Tạo thông tin mô tả về chiếc ghế tựa cho biến đối tượng WF_Object */

    ViewObject(); // Thể hiện đối tượng Lên trên cửa sổ chương trình

    return TRUE;
}
```

Thêm đoạn mã lệnh sau vào cuối hàm đáp ứng sự kiện `CWireFrameDemoDlg:: OnPaint` như sau:

```
void CWireFrameDemoDlg::OnPaint()
{
    ... /* Các lệnh mặc định có sẵn trong hàm OnPaint() do trình Wizard tạo ra */

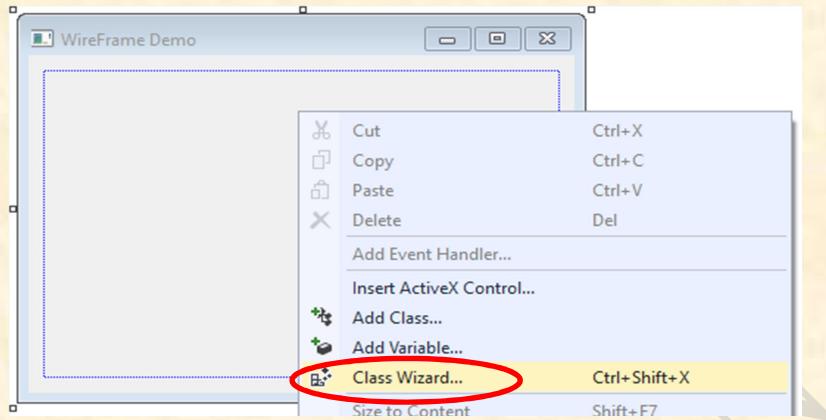
    ViewObject(); // Câu Lệnh thêm vào cuối nhằm vẽ đối tượng Lên cửa sổ
}
```

+ Thêm 3 hàm xử lý sự kiện chuột trên cửa sổ giao diện gồm:

- WM\_LBUTTONDOWN
- WM\_LBUTTONUP
- WM\_MOUSEMOVE.

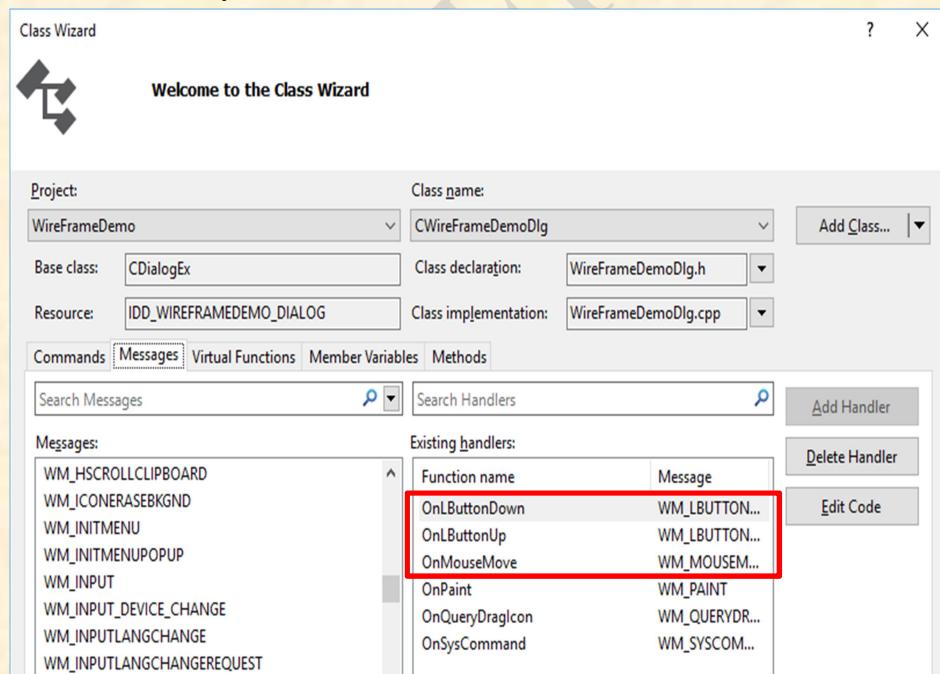
Để thực hiện cần tiến hành theo các bước sau:

- Click chuột phải trên cửa sổ chính của chương trình để làm xuất hiện menu ngữ cảnh, kế đến chọn mục “Class Wizard...”:



Hình 5.8. Thực hiện Class Wizard với lớp CWireFrameDemoDlg để thêm các sự kiện

- Chọn Messages Tab, tiếp đến cần tìm kiếm và add hàm xử lý cho các sự kiện chuột gồm: WM\_LBUTTONDOWN, WM\_LBUTTONUP, WM\_MOUSEMOVE như trong hình vẽ dưới đây:



Hình 5.9. Thêm các hàm xử lý sự kiện chuột trên cửa sổ chính của chương trình

- Viết mã lệnh cho các hàm đáp ứng sự kiện như sau:

```

void CWireFrameDemoDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    MouseDown = true;
    Position_MouseDown = point;
    MouseDown_DeltaAngle = DeltaAngle;
    MouseDown_PhiAngle = PhiAngle;
}

void CWireFrameDemoDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
    MouseDown = false;
}

void CWireFrameDemoDlg::OnMouseMove(UINT nFlags, CPoint point)
{
    if (MouseDown)
    {
        int Del_X = point.x - Position_MouseDown.x;
        int Del_Y = point.y - Position_MouseDown.y;

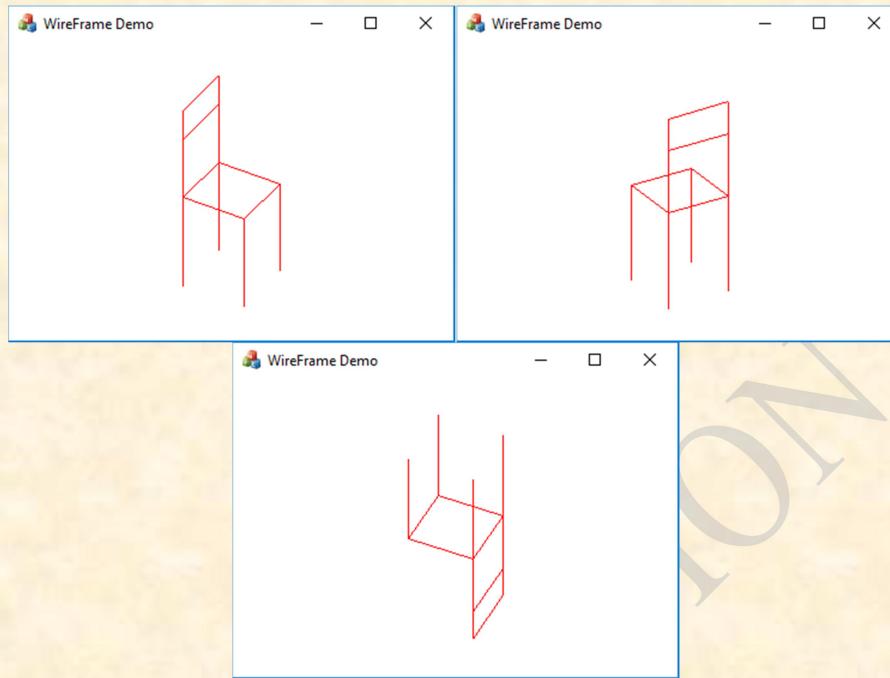
        DeltaAngle = MouseDown_DeltaAngle - Del_X;
        PhiAngle = MouseDown_PhiAngle + Del_Y;
        DeltaAngle = DeltaAngle % 360;
        PhiAngle = PhiAngle % 360;

        if (DeltaAngle < 0) DeltaAngle += 360;
        if (PhiAngle < 0) PhiAngle += 360;

        ViewObject();
    }
}

```

- ❖ Bước 5: Biên dịch và chạy thực nghiệm. Cửa sổ chính của chương trình xuất hiện hình ảnh một chiếc ghế tựa màu đỏ, các thao tác rê chuột trái (gồm bấm giữ nút trái chuột và di chuyển) sẽ kéo theo sự thay đổi góc thể hiện hình ảnh trên cửa sổ, dưới đây là hình minh họa tương ứng với 3 góc nhìn khác nhau mà người sử dụng có được bằng thao tác rê chuột trái:



Hình 5.10. Một số góc quan sát đối tượng được thiết lập thông qua các thao tác rê chuột

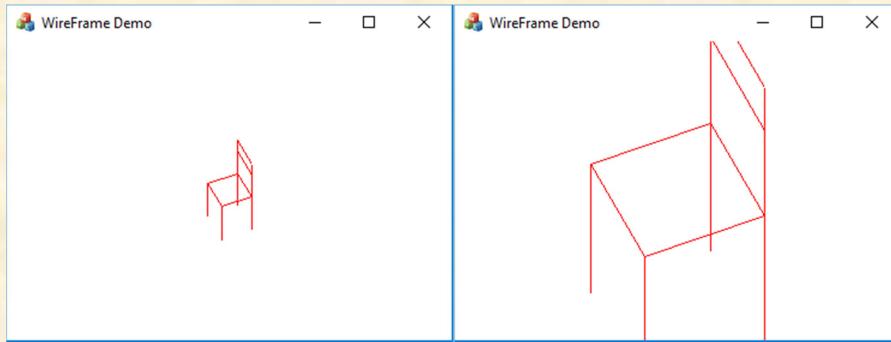
- ❖ Bước cuối: Nâng cấp chương trình với chức năng phóng to hay thu nhỏ đối tượng thông qua thao tác lăn chuột giữa:
  - + Add thêm hàm xử lý sự kiện chuột WM\_MOUSEWHEEL
  - + Viết mã đáp ứng sự kiện như sau:

```
BOOL CWireFrameDemoDlg::OnMouseWheel(UINT nFlags, short zDelta,
CPoint pt)
{
    WF_Object.Zoom += 0.01*WF_Object.Zoom*(zDelta / WHEEL_DELTA);
    if (WF_Object.Zoom > 100)    WF_Object.Zoom = 100;

    if (WF_Object.Zoom < 0.01)  WF_Object.Zoom = 0.01;

    ViewObject();
    return CDialogEx::OnMouseWheel(nFlags, zDelta, pt);
}
```

Chức năng phóng to thu nhỏ được thể hiện qua hình dưới đây:



Hình 5.11. Đối tượng được thể hiện với các kích cỡ khác nhau

### 3. BÀI TẬP CUỐI CHƯƠNG

- ◆ **Hướng dẫn nâng cao:** Sinh viên cần nâng cấp chương trình trên bằng việc xây dựng các mô-đun sau:
  - Xây dựng hàm đọc thông tin mô tả đối tượng 3 chiều theo mô hình wireframe trong một file text với cấu trúc dạng:
    - + Dòng đầu tiên: Chứa tên của đối tượng.
    - + Dòng thứ 2: Chứa 2 số nguyên, lần lượt là số đỉnh (gọi là M) và số cạnh (gọi là N) của đối tượng.
    - + Dòng thứ 3 đến dòng thứ (M+2): Mỗi dòng mô tả thông tin tọa độ của một đỉnh, bao gồm 3 số thực tương ứng với 3 thành phần tọa độ x, y, z của đỉnh.
    - + Dòng thứ (M+3) đến dòng thứ (M+N+2): Mỗi dòng mô tả thông tin của một cạnh. Thông tin gồm 2 số nguyên tương ứng với thứ tự của hai đỉnh tạo nên cạnh.

Ví dụ, thông tin mô tả chiếc ghế trên sẽ được mô tả trong file text như sau:

Hinh cai ghe tua
12 10
1 1 -3
-1 1 -3

-1 -1 -3
1 -1 -3
1 1 0
-1 1 0
-1 -1 0
1 -1 0
1 -1 2
-1 -1 2
1 -1 3
-1 -1 3
1 5
2 6
3 12
4 11
5 6
6 7
7 8
8 5
9 10
11 12

- Xây dựng thêm hàm cho phép thực hiện chiếu phổi cảnh rồi ánh xạ lên màn hình thay thế cho phép chiếu trực giao đang được sử dụng trong chương trình gốc. Ở đây, chúng ta dùng công thức chiếu phổi cảnh đã tìm ra trong chương **các phép biến đổi hình học** mục **Quan sát vật thể 3 chiều và quay hệ quan sát**, đó là công thức:

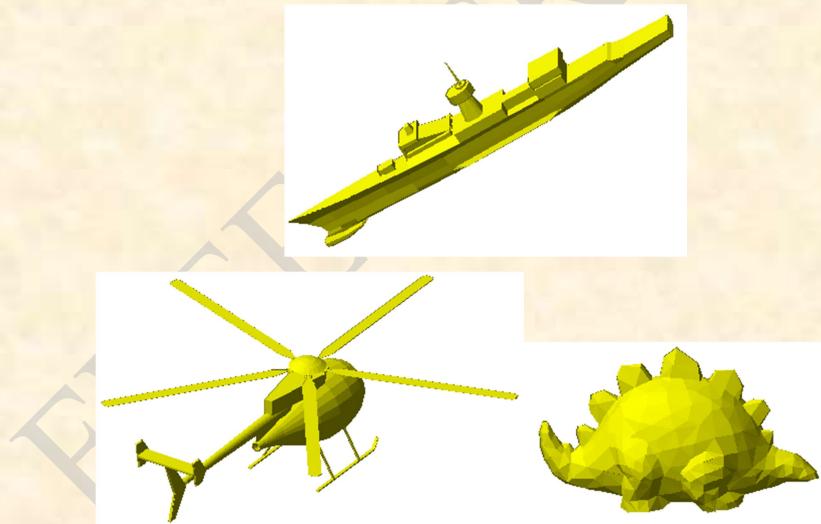
$$x' = x.D/z \quad \text{và} \quad y' = y.D/z$$

- Người sử dụng có thể lựa chọn việc thay thế hiện hình ảnh lên mặt phẳng chiếu qua phép chiếu trực giao hay chiếu phổi cảnh, dựa trên một nút chọn (Radio button) hay hộp kiểm (Check box).

## Chương 6

# MÔ HÌNH CÁC MẶT ĐA GIÁC VÀ VẤN ĐỀ KHỦ MẶT KHUẤT

Tiếp tục vấn đề mô phỏng đối tượng 3 chiều trên máy tính ở chương trước, trong chương này chúng ta sẽ tìm hiểu một mô hình khá hiệu quả để biểu diễn các đối tượng 3 chiều đó là mô hình các mặt đa giác (Polygon mesh model). Với mô hình các mặt đa giác, các vật thể 3 chiều sẽ được mô tả đầy đủ thông tin hơn về hình dáng và tính chất bề mặt, giúp tạo ra hình ảnh chân thực cho đối tượng. Có thể nói, mô hình các mặt đa giác là mô hình phổ biến nhất được áp dụng trong việc mô phỏng đối tượng 3 chiều trong lĩnh vực đồ họa 3 chiều. Tuy nhiên, mô hình này cũng đặt ra nhiều khó khăn thách thức trong việc giải quyết các vấn đề liên quan như xử lý mặt khuất, chiếu sáng...



Hình 6.1. Hình ảnh của một số đối tượng 3 chiều thể hiện theo mô hình các mặt đa giác

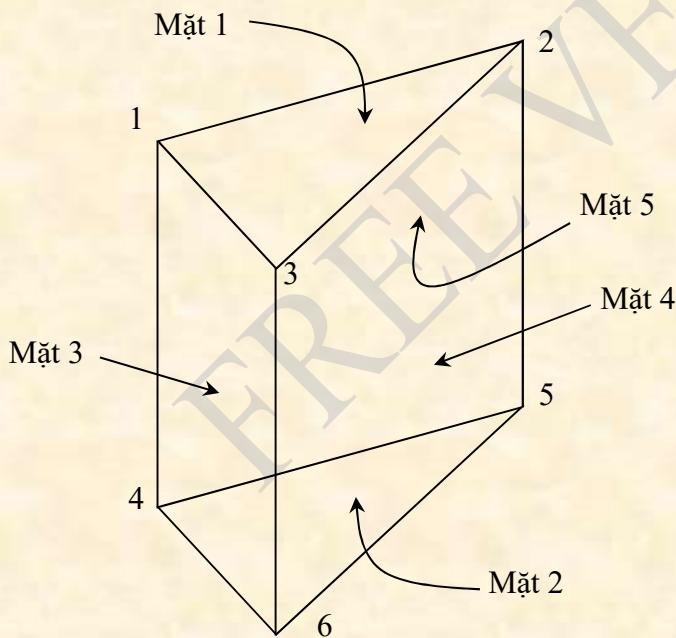
### 1. MÔ TẢ ĐỐI TƯỢNG 3 CHIỀU BẰNG MÔ HÌNH CÁC MẶT ĐA GIÁC

Một vật thể 3 chiều (3D) có thể biểu diễn trong máy tính bằng nhiều mô hình khác nhau, song hai mô hình phổ biến nhất đó là mô hình khung

dây (WireFrame) và mô hình **các mặt đa giác** (Polygon mesh model).

- Mô hình WireFrame: Đã trình bày ở chương trước, nó cho chúng ta hình dáng của vật thể dưới dạng một bộ khung.
- Mô hình các mặt đa giác: Ở đây một vật thể 3D được xác định thông qua các mặt (thay vì các cạnh như trong mô hình WireFrame), và mỗi một mặt lại được xác định thông qua các điểm mà các điểm này được xem như là các đỉnh của mặt đa giác, với mô hình các mặt đa giác thì chúng ta không chỉ tạo ra được hình dáng của vật thể như mô hình Wireframe mà còn thể hiện được các đặc tính về màu sắc và nhiều tính chất khác của vật thể. Song, để có thể mô tả vật thể 3D một cách trung thực (như trong thế giới thực) thì đòi hỏi người lập trình phải tính toán và giả lập nhiều thông tin, mà máu chót là vấn đề khử mặt khuất và chiếu sáng. Trong chương này, chúng ta sẽ tập trung nghiên cứu vấn đề khử mặt khuất.

Ví dụ: Mô tả vật thể như trong hình vẽ sau:



Hình 6.2. Minh họa việc số hóa thông tin vật thể 3 chiều  
theo mô hình các mặt đa giác

- Danh sách các đỉnh: 1,2,3,4,5,6 (như với mô hình Wireframe).
- Danh sách các mặt được xác định theo bảng sau:

Bảng 6.1. Bảng danh sách thông tin các đỉnh của đa giác  
theo mô hình các mặt đa giác

Mặt	Đỉnh
1	1,2,3
2	4,5,6
3	1,3,6,4
4	3,2,5,6
5	1,2,5,4



Hình 6.3. Minh họa đối tượng theo mô hình các mặt đa giác: (a) Một nhân vật game theo mô hình các mặt đa giác cùng với phép ánh xạ hình ảnh bê mặt vật liệu lên các đa giác; (b) Một con Hổ với các mặt đa giác chưa tô màu; (c) Con Hổ với các đa giác được tô màu bằng phương pháp ánh xạ hình ảnh bê mặt vật liệu lên các đa giác

## 2. XÂY DỰNG CẤU TRÚC DỮ LIỆU CHO MÔ HÌNH CÁC MẶT ĐA GIÁC

Chúng ta có thể đưa ra nhiều cấu trúc dữ liệu khác nhau để lưu trữ cho đa giác. Dưới đây là một bản phát thảo:

```

struct Vertex // Cấu trúc lưu trữ thông tin một đỉnh 3 chiều
{   double x,y,z; };

struct Vector // Cấu trúc lưu trữ thông tin một vector 3 chiều
{   double x,y,z; };

struct Surface_3D // Cấu trúc lưu trữ thông tin một mặt đa giác
{
    unsigned int     NumVert; // Số đỉnh
    int *           ListIndex; /* Danh sách chỉ mục của các đỉnh
đa giác */
    COLORREF        Color;    // Màu sắc của đa giác
    Vector          Normal;   // Vector pháp tuyến của mặt đa
giác */
};

class CObject_3D : public CObject /* Khai báo Lớp xử lý với đối
tượng 3 chiều, kế thừa từ Lớp CObject của MFC. */
{
public:
    unsigned int     NumVertex; // Số đỉnh của đối tượng
    Vertex *         ListVertex; // Danh sách đỉnh (cấp phát động)
    unsigned int     NumSurface; // Số mặt của đối tượng
    Surface_3D*      ListSurface; // Danh sách mặt (cấp phát động)
    Vertex           World_Location; /* Vị trí của đối tượng
trong hệ tọa độ thế giới */
    double           Zoom;       // Hệ số thu phóng
    Vertex *         ListVertexInViewCoordinates = NULL; /* Danh sách
đỉnh được chuyển đổi sang hệ tọa độ người quan sát (View
coordinate system), được cấp phát động khi cần và xóa bỏ nếu
không cần sử dụng. Khi xóa cần gán lại về giá trị NULL */
    static bool ReadObj(CString FileName, CObject_3D* pObj); /* Hàm đọc file chứa thông tin lưu trữ đối tượng và trả về đối
tượng CObject_3D tương ứng */
    void DrawObject(CDC* p_DC, CRect R);      /* Hàm vẽ đối
tượng lên Device Context */
};

```

```

void TransVertexToViewCoordinates(unsigned int R, unsigned
int DeltaAngle, unsigned int PhiAngle); /* Hàm chuyển đổi tọa
độ của đối tượng từ hệ tọa độ cục bộ sang hệ quan sát */
void PerspectiveProjection(unsigned int D); // Chiếu phối
cánh */
static Vector CalVector(Vertex P1, Vertex P2); /* Tính
vector tạo thành từ 2 đỉnh P1 và P2 */
static Vector NormalVector (Vertex P1, Vertex P2, Vertex
P3); /* Tính vector pháp tuyến của bề mặt đa giác từ 3 đỉnh của
đa giác. Chú ý tratt tự phù hợp */
double ScalarProductofVectors(Vector V1, Vector V2); /* Tính
tích vô hướng của 2 vector */
CObject_3D(); // Hàm tạo đối tượng
virtual ~CObject_3D(); // Hàm hủy đối tượng

private:
    double LUT_Cos[360]; // Bảng tra hàm Cos
    double LUT_Sin[360]; // Bảng tra hàm Sin
};

```

Khi cài đặt cho một ứng dụng cụ thể, việc sử dụng mảng cố định có thể gây ra các trở ngại về kích thước tối đa hay tối thiểu, cũng như việc sử dụng bộ nhớ không tối ưu. Vì thế, chúng ta cần sử dụng mảng động trong một số ngôn ngữ như Visual Basic, Delphi hay Visual C++,... hoặc dùng cấu trúc danh sách móc nối. Song song với điều đó là việc bớt đi hay đưa thêm các thuộc tính cần thiết để biểu diễn các đặc tính khác của mặt hay của đối tượng

❖ Vấn đề khử mặt khuất:

Khi thể hiện vật thể 3D, một vấn đề này sinh là làm sao chỉ thể hiện các mặt có thể nhìn thấy được mà không thể hiện các mặt bị che khuất, mà một mặt bị che khuất hay không thì tùy thuộc vào cấu trúc các mặt của vật thể và vị trí quan sát, cùng với bối cảnh không gian mà vật thể đó được đặt vào. Chúng ta sẽ tìm hiểu một số phương pháp khử mặt khuất trong phần tiếp theo dưới đây.

### 3. CÁC PHƯƠNG PHÁP KHỬ MẶT KHUẤT

#### 3.1. Giải thuật người thợ sơn với chiến lược sắp xếp theo chiều sâu (Depth-Sorting)

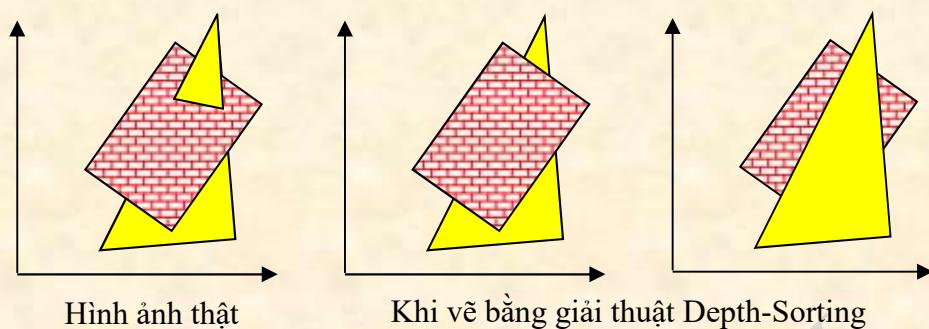
Người thợ sơn (hay Depth-sorting) là tên của một giải thuật đơn giản nhất trong số các giải thuật vẽ ảnh thực 3 chiều. Nếu để ý người thợ sơn làm việc, chúng ta sẽ thấy anh ta sơn bức tranh từ trong ra ngoài, với các cảnh vật từ xa đến gần. Từ đó, chúng ta có thể áp dụng một cách tương tự để vẽ các đa giác trong danh sách các đa giác. Tuy nhiên, việc xác định một giá trị độ sâu làm đại diện cho mặt đa giác, *để dựa vào đó làm tiêu chí sắp xếp lại các đa giác theo thứ tự từ xa đến gần*, là vấn đề có nhiều chọn lựa với các ưu điểm và nhược điểm đi kèm. Cụ thể, một đa giác tồn tại trong không gian 3 chiều có tới 3 bốn đỉnh, và những đỉnh này có thể có các giá trị  $z$  (giá trị độ sâu) khác nhau. Từ đó, nảy sinh một câu hỏi là chúng ta sẽ xác định giá trị độ sâu đại diện cho đa giác như thế nào từ các giá trị độ sâu của các đỉnh? Từ những kinh nghiệm trong thực tế, người ta cho rằng nên lấy giá trị  $z$  là trung bình cộng từ các giá trị độ sâu của đỉnh, sẽ cho kết quả khả quan trong hầu hết các trường hợp.

Như vậy, chúng ta cần phải sắp xếp các mặt theo thứ tự từ xa đến gần, rồi sau đó vẽ các mặt từ xa trước, rồi vẽ các mặt ở gần sau, như thế thì các mặt ở gần sẽ không bị che khuất bởi các mặt ở xa, mà chỉ có các mặt ở xa mới có thể bị các mặt ở gần che khuất, do các mặt ở gần vẽ sau nên có thể được vẽ chồng lên hình ảnh của các mặt xa.

Như vậy, giải thuật Depth-Sorting được thực hiện một cách dễ dàng khi chúng ta xác định một giá trị độ sâu (là giá trị  $z$  trong hệ tọa độ quan sát) đại diện cho cả mặt. Các mặt dựa vào độ sâu đại diện của mình để so sánh rồi sắp xếp theo một danh sách giảm dần (theo độ sâu đại diện). Bước tiếp theo là vẽ các mặt lên mặt phẳng theo thứ tự trong danh sách.

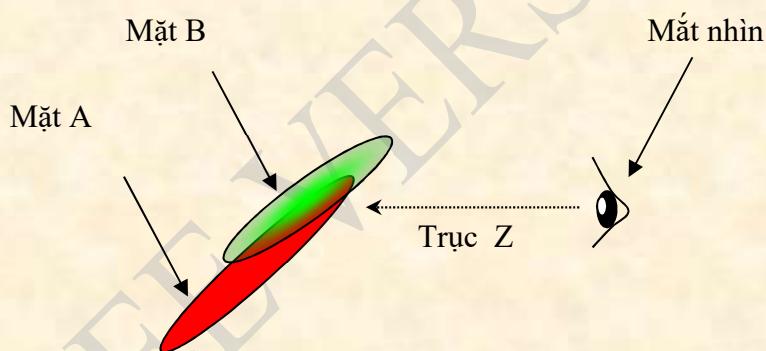
Giải thuật còn một số vướng mắc sau:

- ❖ Khi hai mặt cắt nhau, xem Hình 6.4, thì giải thuật này chỉ thể hiện như chúng chồng lên nhau.



Hình 6.4. Minh họa sai lệch của giải thuật sắp xếp theo độ sâu khi hai  
mặt phẳng ở trong trạng thái cắt nhau

- ❖ Khi hai mặt là ở trong cùng một khoảng không gian về độ sâu và hình chiếu của chúng lên mặt phẳng chiếu là chồng lên nhau, hay chồng một phần lên nhau, như thể hiện trong Hình 6.5.



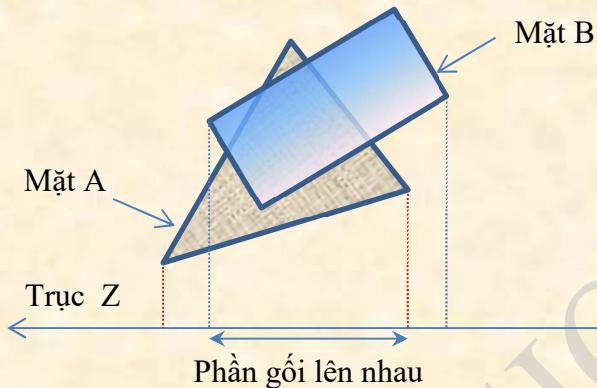
Hình 6.5. Minh họa sai lệch của giải thuật sắp xếp theo độ sâu khi hai mặt đa giác ở trong trạng thái chồng lên nhau

Từ những ví dụ trên chúng ta có thể thấy rằng, có những trường hợp các đa giác được sắp xếp sai dẫn đến kết quả hiển thị không đúng. Liệu chúng ta có thể khắc phục được vấn đề này không? Câu trả lời dĩ nhiên là được, nhưng cũng có nghĩa là chúng ta sẽ phải xử lý thêm rất nhiều các trường hợp và làm tăng độ phức tạp tính toán. Cụ thể:

- Phép kiểm tra phần kéo dài Z

Phép kiểm tra này nhằm xác định xem phần kéo dài (hay khoảng biến thiên, hình chiếu) trên trục Z của hai đa giác có gối đầu lén nhau

hay không? Nếu các phần kéo dài trên trục Z là gối lên nhau rất có thể hai đa giác này cần được hoán đổi thứ tự sắp xếp. Vì thế, phép kiểm tra tiếp theo phải được thực hiện.



Hình 6.6. Minh họa phép kiểm tra phần kéo dài trên trục Z

- Phép kiểm tra phần kéo dài X

Phép kiểm tra này tương tự như phép kiểm tra trước, nhưng nó sẽ kiểm tra phần kéo dài trên trục X của hai đa giác có gối lên nhau hay không? Nếu có, thì rất có thể hai đa giác này cần được hoán đổi thứ tự sắp xếp. Vì thế, phép kiểm tra tiếp theo phải được thực hiện.

- Phép kiểm tra phần kéo dài Y

Phép kiểm tra này kiểm tra phần kéo dài trên trục Y của hai đa giác có gối lên nhau hay không? Nếu có, thì rất có thể hai đa giác này cần được hoán đổi thứ tự sắp xếp. Vì thế, phép kiểm tra tiếp theo phải được thực hiện.

- Phép kiểm tra cạnh gần

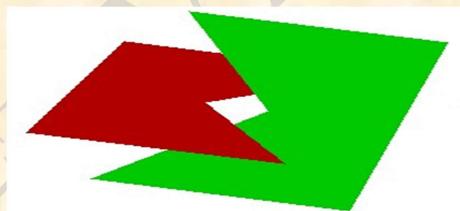
Giả sử A và B là hai đa giác mà sau khi sắp xếp theo độ sâu đại diện thì A đứng trước B (hay mặt A được xem là ở xa hơn mặt B). Song, qua 3 phép kiểm tra nêu trên mà vẫn không xác định được liệu trật tự trên là chính xác hay chưa? Lúc này, chúng ta phải tiến hành phép kiểm tra cạnh gần. Phép kiểm tra cạnh gần, nhằm xác định xem các đỉnh của đa giác B có nằm về cùng một phía của mặt phẳng A, và ở về phía gần với mắt nhìn (hay gốc của hệ tọa độ quan sát) hay

không? Nếu có thì trật tự này là đúng, ngược lại thì phải qua bước kiểm tra tiếp theo.

Để kiểm tra đa giác B có nằm trước cạnh gần của đa giác A hay không, chúng ta phải thực hiện việc kiểm tra mỗi đỉnh của đa giác B. Nếu các đỉnh này đều nằm về cùng một phía của đa giác A theo chiều ngược lại chiều của trục Z hay không, nếu đúng thì kết quả trật tự trên là đúng. Ngược lại, có thể xảy ra một trong hai tình huống như minh họa trong Hình 6.4 và Hình 6.5, để xác định được chúng ta phải tiếp tục sang bước kiểm tra tiếp theo.

- Phép kiểm tra cạnh xa

Phép kiểm tra cạnh xa nhằm xác định xem đa giác A có nằm phía sau cạnh xa của đa giác B hay không? Nếu có thì trật tự xác định trước đây là chính xác. Ngược lại, thì rõ ràng hai đa giác đang cắt nhau như Hình 6.4 hoặc chéo vào nhau như Hình 6.7, lúc này chúng ta phải tiến hành chia nhỏ hai đa giác A và B thành 3 (hoặc 4) đa giác con, đường chia cắt chính là đường giao cắt của 2 đa giác. Sau phép chia chúng ta tiến hành sắp xếp lại các đa giác con.



Hình 6.7. Hình ảnh 2 mặt đa giác đan chéo vào nhau

### 3.2. Giải thuật chọn lọc mặt sau (Back-Face Detection)

Sẽ rất đơn giản nếu chúng ta dùng vector pháp tuyến để khử các mặt khuất của một đối tượng 3D đặc và lồi, xem Hình 6.8. Chúng ta sẽ tính góc giữa vector hướng nhìn V và pháp vector N của mặt, nếu góc này là lớn hơn  $90^\circ$  thì mặt là không nhìn thấy (bị khuất), ngược lại thì mặt là nhìn thấy (khả kiến).

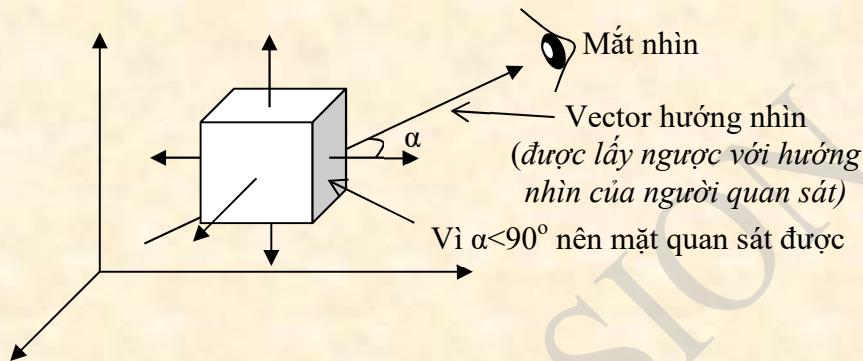
Dấu của tích vô hướng của 2 vector là dương nếu góc giữa chúng nhỏ hơn hay bằng  $90^\circ$ . Vậy, giải thuật để xét một mặt bị khuất hay không chỉ đơn giản là:

If ( $V.N \geq 0$ )

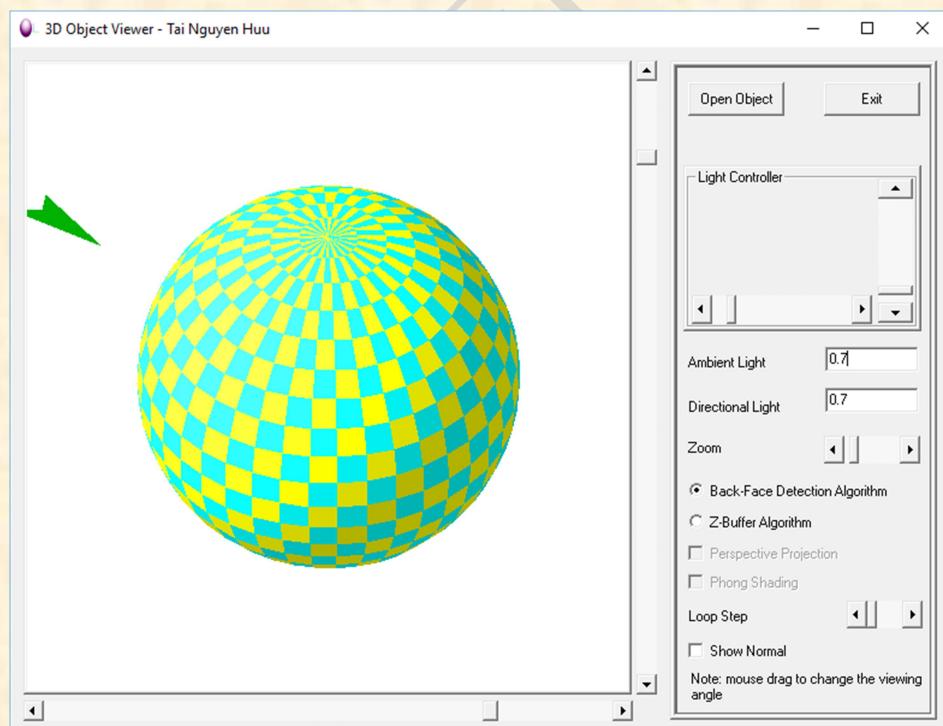
*Mặt quan sát thấy*

Else

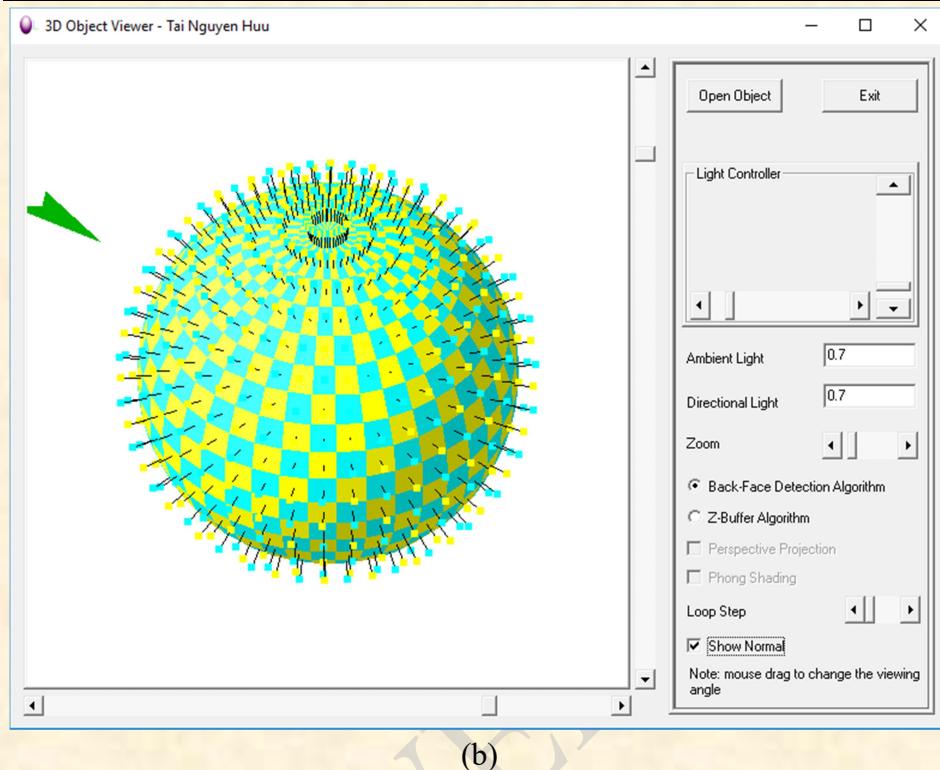
*Mặt không quan sát thấy (hay mặt bị che khuất)*



Hình 6.8. Minh họa cho mô hình chọn lọc mặt sau



(a)



(b)

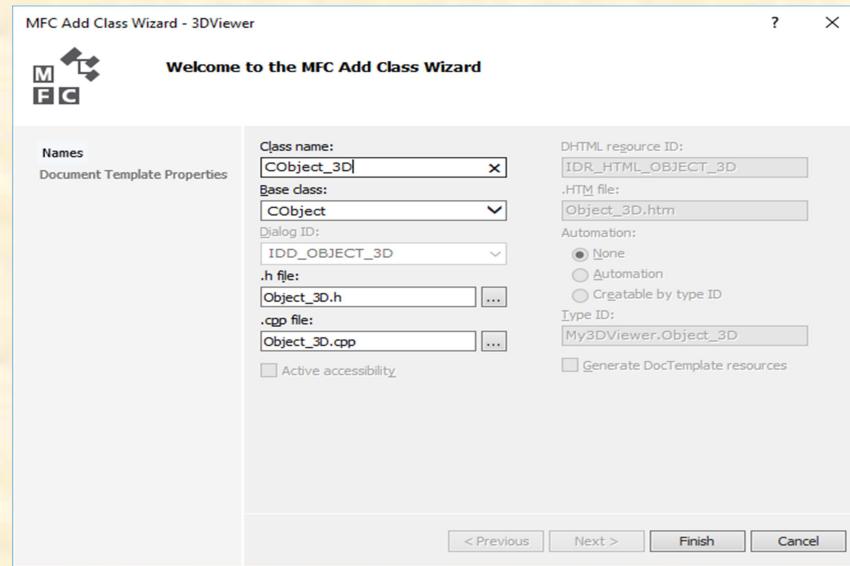
Hình 6.9. Kết quả thực nghiệm xử lý với hình cầu cấu tạo bởi 450 mặt đa giác theo giải thuật chọn lọc mặt sau: (a) Hình thể hiện các mặt quan sát được; (b) Hình thể hiện các mặt quan sát được và vector pháp tuyến của mỗi bề mặt đa giác (có xử lý vân để chiếu sáng)

### 3.3. Cài đặt minh họa cho giải thuật chọn lọc mặt sau

#### Bài thực nghiệm số 4:

Xây dựng chương trình (hay Project) với tên gọi 3DViewer với các bước tương tự như hướng dẫn ở “**Bài thực nghiệm số 3**”, với một số khác biệt:

- ❖ Bước 1: Tạo dự án với tên là 3DViewer (*thay thế cho tên cũ là WireFrameDemo*).
- ❖ Bước 2: Tạo lớp mới có tên là CObject\_3D để lưu trữ và xử lý thông tin của đối tượng 3 chiều theo mô hình các mặt đa giác (*thay thế cho lớp CWireFrame ở bài thực nghiệm số 3*) như Hình 6.10 dưới đây:



Hình 6.10. Tạo lớp CObject\_3D

- ❖ Bước 3: Bổ sung những nội dung chính sau đây vào file header của lớp CObject\_3D, tức file Object\_3D.h:

```
// CObject_3D command target
struct Vertex
{
    double x, y, z;
};

struct Vector
{
    double x, y, z;
};

struct Surface_3D
{
    unsigned int      NumVert;
    int *            ListIndex;
    COLORREF         Color;
    Vector           Normal;
};

class CObject_3D : public CObject
{
public: // properties
    CObject_3D();
}
```

```

virtual ~CObject_3D();

CString Name = L "";
unsigned int NumVertex;
Vertex * ListVertex;
unsigned int NumSurface;
Surface_3D* ListSurface;
Vertex World_Location;
double Zoom;
Vertex * ListVertexInViewCoordinates = NULL;

private:
    double LUT_Cos[360];
    double LUT_Sin[360];

public: // Methods or Functions

    void Create_TetrahedralObjectInfor();
    void TransVertexToViewCoordinates(unsigned int R, unsigned int
DeltaAngle, unsigned int PhiAngle);
    void DrawObject(CDC* p_DC, CRect R);

    static Vector NormalVector(Vertex P1, Vertex P2, Vertex P3);
    static Vector CalVector(Vertex P1, Vertex P2) {
        Vector v;
        v.x = P2.x - P1.x;
        v.y = P2.y - P1.y;
        v.z = P2.z - P1.z;
        return v;
    }
    static double ScalarProductofVectors(Vector V1, Vector V2) {
return (V1.x*V2.x + V1.y*V2.y + V1.z*V2.z); } /* Tính tích vô
hướng của 2 vector */
};


```

- ❖ Bước 4: Bổ sung những nội dung chính sau đây vào file mã nguồn (source file) của lớp CObject\_3D, tức file Object\_3D.cpp :

```

const double PI = 3.14159265358979323846;

// CObject_3D

```

```

CObject_3D::CObject_3D()
{
    for (unsigned int i = 0; i<360; i++)
    {
        double Angle = i*PI / 180;
        this->LUT_Cos[i] = cos(Angle);
        this->LUT_Sin[i] = sin(Angle);
    }
}

CObject_3D::~CObject_3D()
{
}

// CObject_3D member functions

void CObject_3D::Create_TetrahedralObjectInfor()
{
    this->Name = L"Tetrahedral Object";
    this->NumVertex = 4;
    this->ListVertex = new Vertex[this->NumVertex];

    ListVertex[0].x = 1;
    ListVertex[0].y = 0;
    ListVertex[0].z = -1;

    ListVertex[1].x = -1;
    ListVertex[1].y = 1;
    ListVertex[1].z = -1;

    ListVertex[2].x = -1;
    ListVertex[2].y = -1;
    ListVertex[2].z = -1;

    ListVertex[3].x = 0;
    ListVertex[3].y = 0;
    ListVertex[3].z = 1;

    this->NumSurface = 4;
    this->ListSurface = new Surface_3D[this->NumSurface];
}

Surface_3D* p;

```

```
int * pIndex;

// Surface 0
p = &ListSurface[0];
p->NumVert = 3;
p->Color = RGB(255, 0, 0);

p->ListIndex = new int[p->NumVert];

pIndex = p->ListIndex;

pIndex[0] = 0;
pIndex[1] = 2;
pIndex[2] = 1;

// Surface 1
p = &ListSurface[1];
p->NumVert = 3;
p->Color = RGB(0, 255, 0);

p->ListIndex = new int[p->NumVert];

pIndex = p->ListIndex;

pIndex[0] = 0;
pIndex[1] = 1;
pIndex[2] = 3;

// Surface 2
p = &ListSurface[2];
p->NumVert = 3;
p->Color = RGB(0, 0, 255);

p->ListIndex = new int[p->NumVert];
pIndex = p->ListIndex;

pIndex[0] = 0;
pIndex[1] = 3;
pIndex[2] = 2;

// Surface 3
p = &ListSurface[3];
```

```

p->NumVert = 3;
p->Color = RGB(255, 255, 0);

p->ListIndex = new int[p->NumVert];
pIndex = p->ListIndex;

pIndex[0] = 1;
pIndex[1] = 2;
pIndex[2] = 3;

for (unsigned int i = 0; i<NumSurface; i++)
{
    p = &ListSurface[i];
    pIndex = p->ListIndex;

    p->Normal = this->NormalVector(ListVertex[pIndex[0]],
ListVertex[pIndex[1]], ListVertex[pIndex[2]]);
};

this->World_Location.x = 0;
this->World_Location.y = 0;
this->World_Location.z = 0;
this->Zoom = 100;
}

void CObject_3D::TransVertexToViewCoordinates(unsigned int R,
unsigned int DeltaAngle, unsigned int PhiAngle)
{
    if (ListVertexInViewCoordinates == NULL)
    {
        this->ListVertexInViewCoordinates = new Vertex[this-
>NumVertex];
    }

    double x, y, z;

    for (unsigned int i = 0; i<this->NumVertex; i++)
    {
        x = ListVertex[i].x*this->Zoom + this->World_Location.x;
        y = ListVertex[i].y*this->Zoom + this->World_Location.y;
        z = ListVertex[i].z*this->Zoom + this->World_Location.z;
    }
}

```

```

ListVertexInViewCoordinates[i].x = -x*LUT_Sin[DeltaAngle] +
y*LUT_Cos[DeltaAngle];
ListVertexInViewCoordinates[i].y = -x*LUT_Cos[DeltaAngle] *
LUT_Sin[PhiAngle] - y*LUT_Sin[DeltaAngle] * LUT_Sin[PhiAngle] +
z*LUT_Cos[PhiAngle];
ListVertexInViewCoordinates[i].z = -x*LUT_Cos[DeltaAngle] *
LUT_Cos[PhiAngle] - y*LUT_Sin[DeltaAngle] * LUT_Cos[PhiAngle] -
z*LUT_Sin[PhiAngle] + R;
};

Surface_3D* p;
int * pIndex;

for (unsigned int i = 0; i<NumSurface; i++)
{
    p = &ListSurface[i];
    pIndex = p->ListIndex;

    /* Trong hệ tọa độ cục bộ Normal được tính từ 3 điểm theo
    thứ tự (0,1,2). Nhưng chuyển sang hệ quan sát thì đổi ngược thành
    (2,1,0) do tính chất trực tiếp và gián tiếp */
    p->Normal =
NormalVector(ListVertexInViewCoordinates[pIndex[2]],
ListVertexInViewCoordinates[pIndex[1]],
ListVertexInViewCoordinates[pIndex[0]]);
}

void CObject_3D::DrawObject(CDC *p_DC, CRect R)
{
    long X_C = R.CenterPoint().x;
    long Y_C = R.CenterPoint().y;

    for (unsigned int i = 0; i<this->NumSurface; i++)
    {
        Surface_3D* p = &ListSurface[i];
        Vector View_VT = { 0,0,-1 };
        if (ScalarProductofVectors(p->Normal, View_VT)>0) /* Tích vô
hướng giữa 2 vector */
        {
            int * pIndex = p->ListIndex;
            CPoint* ListPoint;
        }
    }
}

```

```

ListPoint = new CPoint[p->NumVert];

for (unsigned int k = 0; k<p->NumVert; k++)
{
    unsigned int Index = pIndex[k];
    ListPoint[k].x =
int(ListVertexInViewCoordinates[Index].x + 0.5) + X_C;
    ListPoint[k].y =
int(ListVertexInViewCoordinates[Index].y + 0.5) + Y_C; // Đảo
chiều trục Y rồi tính tiến
}

CPen Pen(PS_SOLID, 1, RGB(0, 0, 0));
CBrush Brush(p->Color);

CPen* OldPen = p_DC->SelectObject(&Pen);
CBrush* OldBrush = p_DC->SelectObject(&Brush);
int OldFillMode = p_DC->SetPolyFillMode(ALTERNATE);

p_DC->Polygon(ListPoint, p->NumVert);

delete[] ListPoint;
p_DC->SelectObject(OldPen);
p_DC->SelectObject(OldBrush);
if (OldFillMode != ALTERNATE)
{ p_DC->SetPolyFillMode(OldFillMode); }
}
}
}

Vector CObject_3D::NormalVector(Vertex P1, Vertex P2, Vertex P3)
{
    Vector c, a, b;
    b.x = P1.x - P2.x;
    b.y = P1.y - P2.y;
    b.z = P1.z - P2.z;

    a.x = P3.x - P2.x;
    a.y = P3.y - P2.y;
    a.z = P3.z - P2.z;

    c.x = a.y*b.z - a.z*b.y;
}

```

```

c.y = a.z*b.x - a.x*b.z;
c.z = a.x*b.y - a.y*b.x;

return c;
}

```

- ❖ Bước 5: Lập trình xử lý các sự kiện chuột nhằm thể hiện hình ảnh của đối tượng 3 chiều dưới góc nhìn của người quan sát:

- + Các khai báo **cần bổ sung** trong file 3DViewerDlg.h là:

```

1. #include "Object_3D.h" /* Thêm khai báo để có thể sử dụng lớp
CObject_3D */

2. sau từ khóa “protected:” trong phần khai báo lớp
CMy3DViewerDlg cần bổ sung thêm các khai báo:

CObject_3D* Object_3D; /* Khai báo biến con trả đối tượng thuộc
Lớp CObject_3D để Lưu trữ và xử Lý với đối tượng 3 chiều theo mô
hình các mặt đa giác */

int R=50, DeltaAngle=30, PhiAngle=35; /* Khai báo các thông số:
khoảng cách (R), góc θ (DeltaAngle), và góc φ (PhiAngle) */

bool MouseDown=false; // Báo hiệu phím trái chuột có được bấm
xuống?
CPoint Position_MouseDown; /* Lưu trữ vị trí tọa độ (Position)
trên cửa sổ mà tại đó người sử dụng đã bấm giữ nút chuột trái */
int MouseDown_DeltaAngle, MouseDown_PhiAngle; /* Lưu giữ giá
trị các góc θ và φ tại thời điểm bắt đầu bấm nút trái chuột, phục
vụ cho việc tính toán giá trị các góc θ và φ mới khi di chuyển
chuột (mouse move) */

```

- + Add một hàm có dạng **void ViewObject()** vào lớp C3DViewerDlg nhằm cho phép thể hiện hình ảnh của đối tượng lên cửa sổ chương trình, với mã lệnh như sau:

```

void C3DViewerDlg::ViewObject()
{
    CDC *p_DC = this->GetDC(); /*Lấy device context của cửa sổ

```

```

chương trình */
CRect Rect;
this->GetClientRect(&Rect); /* Lấy tọa độ của vùng không gian
bên trong cửa sổ phục vụ cho việc vẽ đối tượng */
Object_3D->TransVertexToViewCoordinates(R, DeltaAngle,
PhiAngle);
/* Chuyển số đo của đối tượng từ hệ tọa độ cục bộ sang hệ quan
sát */
p_DC->FillSolidRect(Rect, RGB(255, 255, 255)); /* Xóa trắng cửa
sổ bằng màu trắng */
Object_3D->DrawObject(p_DC, Rect); // Vẽ đối tượng lên cửa sổ
}

```

- + Thêm đoạn mã lệnh sau vào cuối hàm đáp ứng sự kiện `CWireFrameDemoDlg::OnInitDialog()` như sau:

```

BOOL C3DViewerDlg::OnInitDialog()
{
    ... /* Các câu lệnh mặc định có sẵn trong hàm OnInitDialog() do
trình Wizard tạo ra */
    // TODO: Add extra initialization here
    Object_3D = new CObject_3D();
    Object_3D->Create_TetrahedralObjectInfor(); /* Tạo thông tin mô
tả về hình tứ diện cho biến đối tượng Object_3D */
    ViewObject(); // Thể hiện đối tượng lên trên cửa sổ chương
trình
    return TRUE; // return TRUE unless you set the focus to a
control
}

```

- + Thêm đoạn mã lệnh sau vào cuối hàm đáp ứng sự kiện `C3DViewerDlg::OnPaint()` như sau:

```

void C3DViewerDlg::OnPaint()
{
    ... /* Các câu lệnh mặc định có sẵn trong hàm OnPaint() do
trình Wizard tạo ra */

    ViewObject(); // Câu lệnh thêm vào cuối nhằm vẽ đối tượng lên
cửa sổ
}

```

- + Add thêm 3 hàm xử lý sự kiện chuột trên cửa sổ chính gồm:
  - WM\_LBUTTONDOWN: Bấm nút trái chuột.
  - WM\_LBUTTONUP: Thả nút trái chuột.
  - WM\_MOUSEMOVE: Di chuyển chuột.
  - MOUSEWHEEL: Lăn bánh xe (giữa) chuột.

Xem lại cách thực hiện ở bài thực nghiệm số 3, sau đó viết các lệnh xử lý như sau:

```
void C3DViewerDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    MouseDown = true;
    Position_MouseDown = point;
    MouseDown_DeltaAngle = DeltaAngle;
    MouseDown_PhiAngle = PhiAngle;

    //CDialogEx::OnLButtonDown(nFlags, point);
}

void C3DViewerDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
    MouseDown = false;

    //CDialogEx::OnLButtonUp(nFlags, point);
}

void C3DViewerDlg::OnMouseMove(UINT nFlags, CPoint point)
{
    if (MouseDown)
    {
        int Del_X = point.x - Position_MouseDown.x;
        int Del_Y = point.y - Position_MouseDown.y;

        DeltaAngle = MouseDown_DeltaAngle - Del_X;
        PhiAngle = MouseDown_PhiAngle + Del_Y;
        DeltaAngle = DeltaAngle % 360;
        PhiAngle = PhiAngle % 360;

        if (DeltaAngle < 0) DeltaAngle += 360;
        if (PhiAngle < 0) PhiAngle += 360;
    }
}
```

```

        ViewObject();
    }

    //CDialogEx::OnMouseMove(nFlags, point);
}

BOOL C3DViewerDlg::OnMouseWheel(UINT nFlags, short zDelta, CPoint pt)
{
    Object_3D->Zoom += 0.01*Object_3D->Zoom*(zDelta / WHEEL_DELTA);
    if (Object_3D->Zoom > 1000) Object_3D->Zoom = 1000;

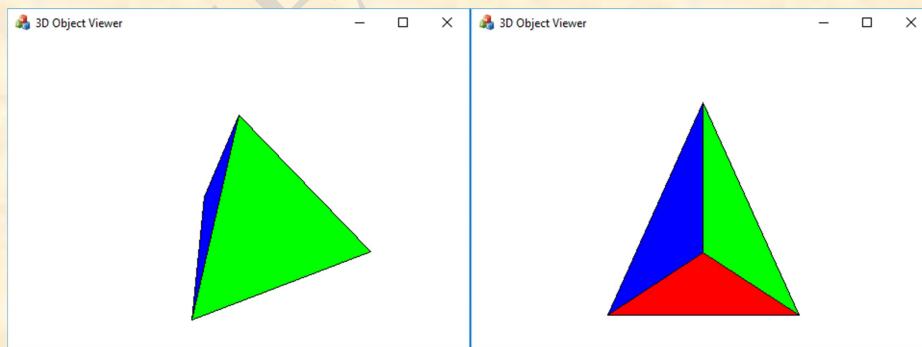
    if (Object_3D->Zoom < 0.001) Object_3D->Zoom = 0.001;

    ViewObject();

    return CDialogEx::OnMouseWheel(nFlags, zDelta, pt);
}

```

- ❖ Biên dịch và chạy thực nghiệm. Cửa sổ chính của chương trình xuất hiện hình ảnh một hình tứ diện với mỗi mặt có một màu sắc riêng biệt. Các thao tác rê chuột trái (gồm bấm giữ nút trái chuột và di chuyển) sẽ kéo theo sự thay đổi góc thể hiện hình ảnh trên cửa sổ, lăn bánh xe chuột để phóng to hay thu nhỏ đối tượng. Dưới đây là vài hình ảnh minh họa kết quả chạy thực nghiệm của chương trình:



Hình 6.11. Hình ảnh thực nghiệm cài đặt giải thuật chọn lọc mặt sau

- ❖ Nâng cấp chương trình:

Để chương trình có thể trình diễn với các hình ảnh đối tượng phong phú hơn, thay cho tình trạng hiện tại chỉ có thể trình diễn duy nhất một

hình tú diện. Chúng ta cần nâng cấp chương trình như sau:

- ❖ Bước 1: Thêm hàm đọc file dữ liệu mô tả đối tượng 3 chiều theo mô hình các mặt đa giác. Với file dữ liệu dạng text được định dạng với hình thức sau đây:

Dữ liệu	Giải thích
Hughes_500	+ Hàng đầu tiên: Tên đối tượng.
675 1270	+ Hàng tiếp theo: Hai số N (số đỉnh) và M (số mặt).
-0.010000 13.150000	+ N hàng tiếp theo mỗi hàng chứa tọa độ của một điểm gồm 3 số đo x, y, z.
3.670000	+ M hàng tiếp theo mỗi hàng chứa thông tin của một mặt đa giác gồm: K, là số lượng đỉnh của mặt đa giác; tiếp theo là một dãy với K số nguyên là chỉ mục nhằm tham chiếu vào danh sách đỉnh để có được tọa độ của K đỉnh của đa giác; kế đến là bộ 3 giá trị cho biết cường độ của 3 thành phần màu đơn sắc R, G, B tạo nên màu của mặt đa giác.
-0.620000 3.690000	+ Hàng tiếp theo (cũng là hàng sau cùng) lưu trữ 4 giá trị trong đó 3 giá trị đầu tiên cho biết vị trí tọa độ của đối tượng trong không gian (hay không gian ảo) gồm: World_Location.x,
3.670000	World_Location.y,
-0.620000 -3.740000	World_Location.z. Giá trị tiếp theo là hệ số thu phóng (Zoom) nhằm có được kích thước thực của đối tượng.
3.670000	
0.580000 -3.740000	
3.670000	
0.580000 3.690000 3.670000	
...	
-0.010000 13.150000	
0.000000	
3 0 5 6 255 255 0	
3 0 6 1 255 255 0	
...	
3 673 674 624 255 255 0	
0.000000 0.000000 0.000000	
10.00000	

- + Nội dung hàm đọc dữ liệu “**ReadObj**” thuộc lớp CObject\_3D như sau:

```

CString readLine(FILE *file) {

    CString s = L"";

    char ch = getc(file);
    int count = 0;

    while ((ch != '\n') && (ch != EOF))
    {
        s += ch;
        ch = getc(file);
    }

    return s;
}

bool CObject_3D::ReadObj(CString FileName, CObject_3D *pObj)
{
    FILE *stream;

    USES_CONVERSION;
    const char* Name = T2A((LPCTSTR)FileName);

    errno_t err = fopen_s(&stream, Name, "r");

    if (err !=0)
    {

        MessageBox(NULL,_T("Error!!! Can't open file for
read"),_T("Error!"),MB_ICONEXCLAMATION + MB_OK);
        MessageBox(NULL, (CString)FileName, _T("Error!"),
MB_ICONEXCLAMATION + MB_OK);
        return false;
    }
    else
    {
        CString Name = readLine(stream);

        unsigned int NumVertex,NumSurface;

        pObj->Name = Name;
    }
}

```

```

fscanf_s( stream, "%d %d", &NumVertex,&NumSurface);

pObj->NumVertex=NumVertex;
pObj->NumSurface=NumSurface;

pObj->ListVertex = new Vertex[NumVertex];
if (pObj->ListVertex==NULL)
{
    MessageBox(NULL, _T("Can't create memory"),
_T("Error!"), MB_OK);
    return false;
}

Vertex * pV=pObj->ListVertex;
for (unsigned int i=0; i<NumVertex; i++)
{
    float x,y,z;
    x=0;y=0;z=0;
    fscanf_s(stream, "%f %f %f", &x, &y, &z);

    pV[i].x=x;
    pV[i].y=y;
    pV[i].z=z;

}

pObj->ListSurface = new Surface_3D[NumSurface];
if (pObj->ListSurface==NULL)
{
    MessageBox(NULL, _T("Can't create memory"),
_T("Error!"), MB_OK);
    delete[] (pObj->ListVertex);
    return false;
}

Surface_3D* pS=pObj->ListSurface;
unsigned int NumVert,index;
for (unsigned int i=0; i<NumSurface; i++)
{
    fscanf_s(stream, "%d ", &NumVert);

    pS[i].NumVert=NumVert;
}

```

```

pS[i].ListIndex= new int[NumVert];
int * pIndex=pS[i].ListIndex;
for (unsigned int k=0; k<NumVert;k++)
{
    fscanf_s(stream, "%d ", &index);

    pIndex[k]=index;
};

unsigned int R,G,B;
fscanf_s(stream, "%d %d %d", &R, &G, &B);
pS[i].Color=RGB(R,G,B);

}

float Xw,Yw,Zw,Zoom;

fscanf_s(stream, "%f %f %f %f", &Xw, &Yw, &Zw, &Zoom);

pObj->World_Location.x=Xw;
pObj->World_Location.y=Yw;
pObj->World_Location.z=Zw;

pObj->Zoom=Zoom;

fclose( stream );

return true;
}
}
}

```

- ❖ Bước 2: Thêm một nút bấm (**button control**) với nội dung caption là “Open File” rồi viết hàm đáp ứng sự kiện click chuột lên nút như sau:

```

void C3DViewerDlg::OnBnClickedButton1()
{
    CFileDialog m_ldFile(TRUE);

    m_ldFile.m_ofn.lpstrInitialDir = _T(".");
    m_ldFile.m_ofn.lpstrFilter = TEXT("Object 3D")

```

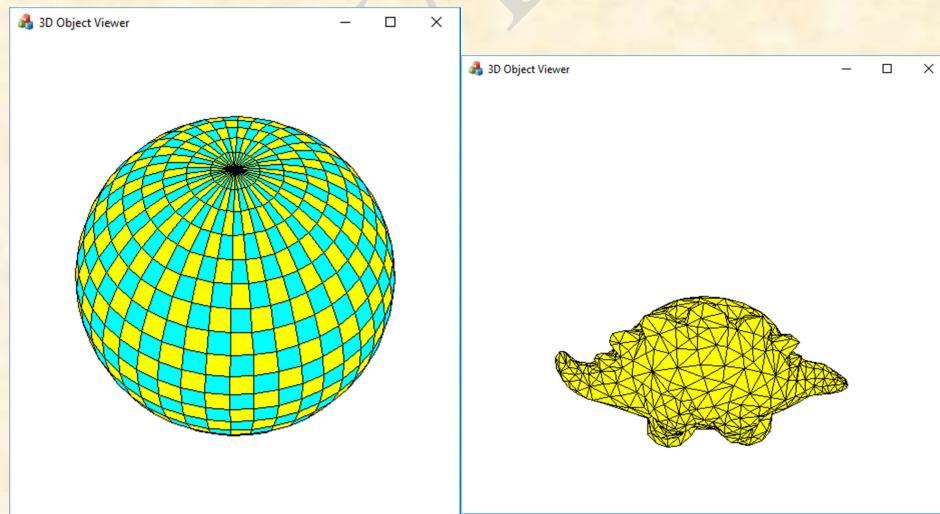
```
(*.Obj)\0*.Obj\0");
m_ldFile.m_ofn.nFilterIndex = 1;

if (m_ldFile.DoModal() == IDOK)
{
    CObject_3D *pObj;
    pObj = new CObject_3D();

    if (CObject_3D::ReadObj(m_ldFile.GetPathName(), pObj))
    {
        delete Object_3D;
        Object_3D = pObj;

        ViewObject();
    }
}
}
```

- ❖ Biên dịch và chạy thực nghiệm. Tiếp đến bấm nút Open file để mở file dữ liệu đối tượng 3 chiều “hình cầu” và “khủng long” (Dinasaur) cho chúng ta hình ảnh dưới đây:



Hình 6.12. Hình ảnh thực nghiệm cài đặt giải thuật chọn lọc mặt sau, với các đối tượng hình cầu, khủng long (Dinasaur) từ các file dữ liệu mô tả

Kết quả thực hiện chương trình cho thấy, với các đối tượng phức tạp có nhiều mặt cần xử lý, thì khi chúng ta rê chuột để thay đổi vị trí quan

sát sẽ khiến cho hình ảnh bị nhấp nháy. Nguyên nhân của hiện tượng này là do chúng ta thực hiện vẽ đối tượng trên “Screen DC” như đã được bàn luận ở chương 2 phần kỹ thuật vẽ đồ họa hậu trường (Off-Screen). Để khắc phục hiện tượng này, chúng ta cần thực hiện các thao tác vẽ đối tượng trên MemDC thay cho DC. Nội dung cần thay đổi trong hàm ViewObject là:

```
void C3DViewerDlg::ViewObject()
{
    CDC *p_DC = this->GetDC(); /* Lấy device context của cửa sổ
chương trình */

    CRect Rect;
    this->GetClientRect(&Rect); /* Lấy tọa độ của vùng không gian
bên trong cửa sổ phục vụ cho việc vẽ đối tượng */

    Rect.right -= 100; /* Giảm bớt chiều rộng để tránh vẽ đè lên
nút Open File */

    CMemDC *pMemDC = new CMemDC(*p_DC, Rect); // Tạo MemDC
    CDC* pDC = &(pMemDC->GetDC());

    Object_3D->TransVertexToViewCoordinates(R, DeltaAngle,
PhiAngle); /* Chuyển số đo của đối tượng từ hệ tọa độ cục bộ sang
hệ quan sát */

    pDC->FillSolidRect(Rect, RGB(255, 255, 255)); /* Xóa trắng cửa
sổ bằng màu trắng */

    Object_3D->DrawObject(pDC, Rect); // Vẽ đối tượng lên MemDC
    pMemDC->~CMemDC(); /* Cập nhật hình ảnh đối tượng lên cửa sổ,
và xóa đối tượng MemDC */

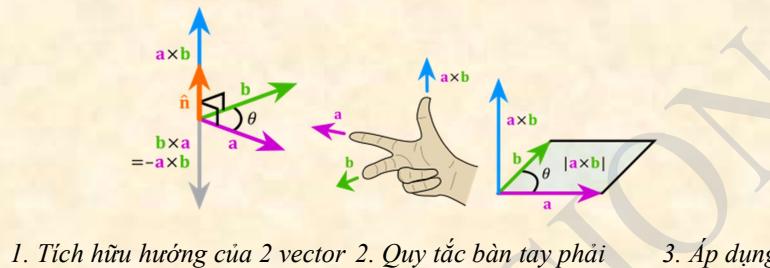
    ReleaseDC(p_DC);
}
```

*Chạy lại chương trình với cách thực hiện đồ họa trên MemDC sẽ cho hình ảnh chuyển động được đáp ứng nhanh chóng và không còn bị hiện tượng nhấp nháy.*

Rõ ràng, giải thuật rất đơn giản và độ phức tạp tính toán không cao. Song, khi sử dụng phải luôn đảm bảo rằng đối tượng có đặt tính là “**đặc và lồi**”, nếu đối tượng không thoả mãn điều kiện đó thì chúng ta phải áp dụng một giải thuật khác hay có những sửa đổi cần thiết để tránh sự thể hiện sai lạc.

❖ **Hướng dẫn thêm:**

Thông thường, vector pháp tuyến được tự động tính toán dựa vào danh sách các đỉnh của mặt đa giác. Vì thế, khi lưu trữ danh sách các đỉnh của đa giác người ta thường thống nhất theo một chiều và thông thường là theo chiều dương (*ngược chiều kim đồng hồ*) khi chúng ta nhìn từ phía ngoài vào mặt đa giác.



1. Tích hữu hướng của 2 vector 2. Quy tắc bàn tay phải 3. Áp dụng

Hình 6.13. Hình minh họa cách xác định tích hữu hướng của hai vector và cách áp dụng

Công thức để tính tích 2 vector  $\vec{b} \times \vec{c} = \vec{a}$  như sau:

$$a_x = b_y c_z - b_z c_y$$

$$a_y = b_z c_x - b_x c_z$$

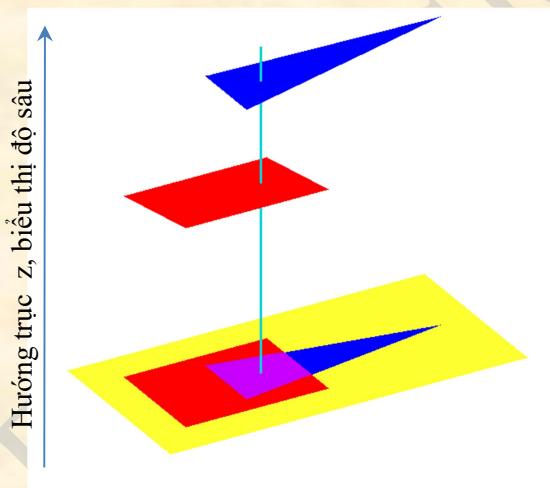
$$a_z = b_x c_y - b_y c_x$$

### 3.4. Giải thuật vùng đệm độ sâu (Z-Buffer)

Bằng cách tính giá trị độ sâu (là giá trị Z trong hệ tọa độ quan sát) của mỗi điểm trong tất cả các mặt đa giác, tại mỗi điểm trên mặt phẳng chiếu có thể có ảnh của nhiều điểm trên nhiều mặt đa giác khác nhau, song hình vẽ chỉ được thể hiện hình ảnh của điểm có độ sâu thấp nhất (tức là điểm ở gần nhất). Với cách thực hiện này, giải thuật có thể khử được tất cả các trường hợp mà các giải thuật khác mắc phải.

Giới hạn của phương pháp này là đòi hỏi nhiều bộ nhớ và thực hiện nhiều tính toán. Z\_Buffer là một bộ đệm dùng để lưu độ sâu cho mỗi pixel trên hình ảnh của vật thể, thông thường chúng ta tổ chức nó là một ma trận hình chữ nhật. Nếu dùng 1 byte để biểu diễn độ sâu của một

pixel, thì một vật thể có hình ảnh trên mặt phẳng chiếu là  $100 \times 100$  sẽ cần 10.000 byte dùng để làm Z-Buffer, và khi đó vùng đệm độ sâu sẽ cho phép chúng ta phân biệt được 256 mức sâu khác nhau, điều này có nghĩa là nếu có 257 pixel ở 257 độ sâu khác nhau thì khi đó buộc chúng ta phải quy 2 pixel nào đó về cùng một độ sâu. Nếu chúng ta dùng 4 byte để biểu diễn độ sâu của một pixel, thì khi đó vùng đệm độ sâu sẽ cho phép chúng ta phân biệt được  $4294967296 (2^{32})$  mức sâu khác nhau, song lúc đó sẽ phải cần 40.000 byte cho một bộ đệm có kích thước  $100 \times 100$ . Do tính chất 2 mặt này nên tùy vào tình huống và yêu cầu mà chúng ta có thể tăng hay giảm số byte để lưu giữ độ sâu của 1 pixel. Thông thường, người ta dùng 4 byte để lưu giữ độ sâu của một điểm, khi đó thì độ chính xác rất cao.



Hình 6.14. Minh họa hình chiếu của 2 mặt phẳng lên mặt phẳng chiếu và phần chồng lấp (overlap) giữa chúng

Một câu hỏi có thể đặt ra là làm sao có thể tính độ sâu của mỗi điểm trong đa giác. Ở đây, có 2 phương pháp: Phương pháp trực tiếp và phương pháp gián tiếp.

➤ **Phương pháp trực tiếp:** Sẽ tính độ sâu của mỗi điểm dựa vào phương trình mặt phẳng chứa đa giác. Với phương pháp này, chúng ta cần duyệt qua tất cả các điểm của đa giác (tất nhiên chỉ hữu hạn điểm), bằng cách cho các thành phần x và y, nếu cặp giá trị  $(x,y)$  thỏa mãn trong miền giới hạn của đa giác, thì chúng ta sẽ tìm thành phần thứ 3 là z

bằng cách thay thế x và y vào phương trình mặt phẳng để tính ra thành phần z. Về mặt toán học, phương pháp trực tiếp rõ ràng là rất khoa học, song khi áp dụng ta sẽ gặp phải một số vướng mắc đó là:

Cần phải tính bao nhiêu điểm để hình ảnh thể hiện của đa giác lên mặt phẳng chiếu đủ mịn và cũng không bị tình trạng quá mịn (tức là vẽ rất nhiều điểm chồng chất lên nhau không cần thiết mà lại gây ra tình trạng chập chờn và tăng độ phức tạp tính toán. Cũng nên nhớ rằng, khi thể hiện một đa giác lên mặt phẳng chiếu thì ảnh của nó có thể được phóng to hay thu nhỏ).

- **Phương pháp gián tiếp**: Chúng ta sẽ tính độ sâu của một điểm gián tiếp thông qua độ sâu của các điểm lân cận. Để thực hiện chúng ta tiến hành theo các bước sau:

Gọi G là một mặt đa giác được biểu diễn bởi tập các điểm  $P_1, P_2, \dots, P_n$  và  $G'$  là hình chiếu của G xuống mặt phẳng chiếu với tập các đỉnh  $P'_1, P'_2, \dots, P'_n$ .

Để thể hiện hình ảnh của G lên mặt phẳng chiếu thì rõ ràng là chúng ta phải tiến hành tô đa giác  $G'$ . Song như giải thuật đã phát biểu, chúng ta cần xác định xem mỗi điểm  $M'$  bất kỳ thuộc  $G'$  là ảnh của điểm  $M$  nào trên G và dựa vào độ sâu của  $M$  để so sánh với độ sâu đã có trong z-buffer, từ đó quyết định là có vẽ điểm  $M'$  hay không. Nếu chúng ta gán thêm cho các điểm ảnh một thành phần nữa, đó là giá trị độ sâu của điểm tạo ảnh (tức là điểm đã tạo ra điểm ảnh sau phép chiếu) thì lúc này chúng ta không cần thiết phải xác định  $M$  để tính độ sâu, mà chúng ta có thể tính được giá trị độ sâu này qua hình thức nội suy tuyến tính:

- Nếu  $M'$  nằm trên đoạn thẳng  $P'Q'$  với tỷ lệ chia là:  $P'M'/P'Q' = t$
- Và biết được độ sâu của  $P'$  và  $Q'$  lần lượt là  $z(P')$  và  $z(Q')$  thì độ sâu mà điểm ảnh  $M'$  nhận được sẽ là:

$$z(M') = z(P') + (z(Q') - z(P'))t \quad (6.1)$$

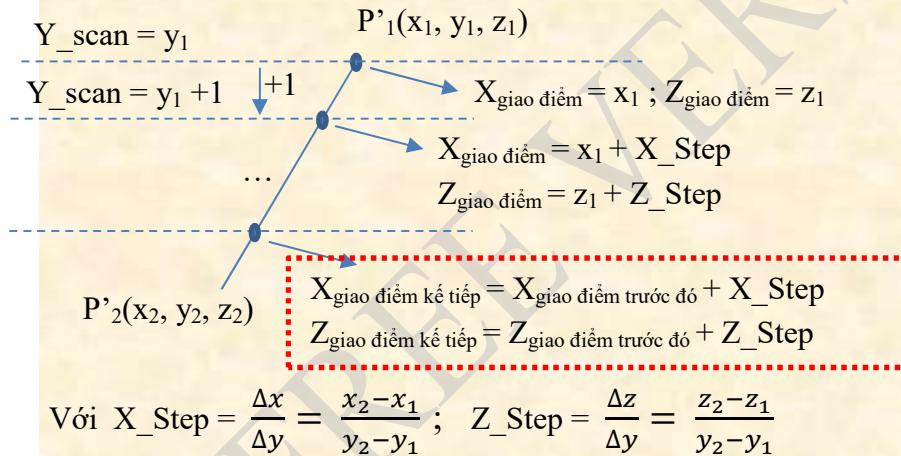
Chúng ta có thể sử dụng công thức trên với các phép chiếu bảo toàn tính chất tuyến tính.

Từ đó, chúng ta có thể xác định quy trình vẽ đa giác  $G'$  là ảnh của

G như sau:

- Gán thêm cho mỗi đỉnh của đa giác  $G'$  một thành phần  $z$  có giá trị bằng độ sâu của điểm tạo ảnh. Có nghĩa là  $P'_1$  sẽ chứa thêm giá trị  $z(P_1)$ ,  $P'_2$  sẽ chứa thêm giá trị  $z(P_2)$ , hay một cách tổng quát  $P'_i$  sẽ chứa thêm giá trị  $z(P_i)$  với  $i = 1..n$ .
- Tiến hành tô đa giác  $G'$  theo giải thuật tô đa giác theo dòng quét (Scanline Algorithm), có bổ sung thêm thông tin độ sâu tại mỗi đỉnh của đa giác. Tại mỗi giao điểm của đường quét với cạnh của đa giác: Ngoài việc phải nội suy  $x_{\text{giao\_điểm}}$  chúng ta cần nội suy giá trị độ sâu ( $z_{\text{giao\_điểm}}$ ) theo cùng một cách như đã được trình bày trong “Hình 3.22. Minh họa cơ chế nội suy giao điểm trong giải thuật tô đa giác theo dòng quét (Scanline Algorithm)”.

Hình ảnh dưới đây minh họa cơ chế nội suy giao điểm và độ sâu tại giao điểm của dòng quét với cạnh đa giác:



Hình 6.15. Minh họa cơ chế nội suy giao điểm trong giải thuật tô đa giác theo dòng quét (Scanline Algorithm) có tính đến nội suy độ sâu của tạo ảnh, để có thể áp dụng vào giải thuật vùng đệm độ sâu. Ở đây  $z_1$  chính là giá trị độ sâu của điểm  $P_1$ , và  $P_1$  là tạo ảnh của  $P'$

### 3.5. Cài đặt minh họa cho giải thuật “vùng đệm độ sâu”

Từ những phân tích trên, chúng ta có thể tiến hành khai báo các cấu trúc dữ liệu cần thiết và cài đặt cho giải thuật.

**Bài thực nghiệm số 5:**

Nâng cấp “**bài thực nghiệm số 4**” để có thể áp dụng giải thuật vùng đệm độ sâu nhằm khử mặt khuất của đối tượng 3 chiều. Cụ thể:

- Bổ sung thêm các khai báo cấu trúc dữ liệu cần thiết so với những khai báo trước đây. Cụ thể:

```

struct Point_Z /* Cấu trúc dữ liệu lưu trữ hình chiếu Q(x,y) của
một đỉnh bất kỳ P trong mặt đa giác lên mặt phẳng chiếu, nên
thông tin gồm 2 thành phần tọa độ x và y. Song, để phục vụ cho
tác vụ nội suy độ sâu của giải thuật z-buffer thì cần thêm thông
tin độ sâu của P (của tạo ảnh), đó chính là lý do có thêm thành
phần thứ 3 là z */
{
    int x, y; /* Tọa độ ảnh chiếu của một đỉnh lên mặt phẳng được
nguyễn hóa nhằm xử lý trên mặt phẳng thiết bị đồ họa */
    double z; // Độ sâu của tạo ảnh, phục vụ quá trình nội suy độ
sâu */
};

struct EdgeInfType /* Cấu trúc dữ liệu chứa các thông tin về một
cạnh bất kỳ của đa giác phục vụ cho việc nội suy giao điểm
và độ sâu tại giao điểm với dòng quét (scan-Line) một cách hiệu
quả */
{
    int Y_Top;
    double X_Intersection;
    double Z_Intersection;
    double Xstep;
    double Zstep;
    int Y_Bottom;
    int X_Bottom;
    double Z_Bottom;
};
struct IntersectionType // Thông tin tại một giao điểm
{
    int X;
    double Z;
    int EdgeIndex;
};

```

➤ Bổ sung thêm các hàm xử lý:

```

void CObject_3D::DrawObject_ZBufferAlgorithm(BYTE *pBits, int
BytePerLine, CRect Rect, double * Z_Buffer)
/* Ở đây *pBits Là một con trỏ trả đến một mảng dữ liệu ảnh dạng
DIB chế độ 24-bit với số điểm ảnh bằng (Width * Height), và số
byte trên một dòng ảnh Là BytePerLine, từ những thông tin này
chúng ta sẽ dễ dàng tính ra được địa chỉ ô nhớ chứa thông tin của
một điểm có tọa độ (x,y) sẽ Là Address = y * BytePerLine + x * 3.
Các thao tác vẽ hình đối tượng được thực hiện trên ảnh DIB này,
sau đó nếu muốn hiển thị hình ảnh lên của sổ nào thì cần thao tác
sao chép (hàm BitBlt) nội dung của DIB lên DC của cửa sổ đó. Cách
làm này giúp nâng cao tốc độ thực thi. Tham khảo các tài liệu số
7, 8 và 9 để nắm chi tiết hơn vấn đề. */
{
    long X_C = Rect.CenterPoint().x;
    long Y_C = Rect.CenterPoint().y;

    for (unsigned int i = 0; i<this->NumSurface; i++)
    {
        Surface_3D* p = &ListSurface[i];

        int * pIndex = p->ListIndex;
        Point_Z* ListPoint;

        ListPoint = new Point_Z[p->NumVert];

        for (unsigned int k = 0; k<p->NumVert; k++)
        {
            unsigned int Index = pIndex[k];
            ListPoint[k].x =
int(ListVertexInViewCoordinates[Index].x+0.5) + X_C;
// Đảo chiều trục Y rồi tính tiến đến giữa cửa sổ theo vector
(X_C,Y_C)
            ListPoint[k].y = -int(ListVertexInViewCoordinates[Index].y+0.5)
+ Y_C;
            ListPoint[k].z = ListVertexInViewCoordinates[Index].z;
        }

        BYTE R = GetRValue(p->Color);
        BYTE G = GetGValue(p->Color);
        BYTE B = GetBValue(p->Color);
    }
}

```

```

    this->FillPoly_Z(ListPoint, p->NumVert, pBits, BytePerLine,
R, G, B, Z_Buffer, Rect.Width(), Rect.Height());

    delete[] ListPoint;
}

}

```

Trong đó, hàm FillPoly\_Z và các hàm con phục vụ cho nó được xây dựng như sau:

```

void CObject_3D::FillPoly_Z(Point_Z P[], int Num_Vertex, BYTE
*pBits, int BytePerLine, BYTE Rcolor, BYTE Gcolor, BYTE Bcolor,
double Z_Buffer[], int Width, int Height)
{
    IntersectionType *ListIntersection;

    int i, j;
    int Y_Min = P[0].y, Y_Max = Y_Min;

    // Tìm Min Max
    for (i = 1; i < Num_Vertex; i++)
    {
        if (Y_Min>P[i].y)
            Y_Min = P[i].y;
        else
            if (Y_Max<P[i].y)
                Y_Max = P[i].y;
    }

    if (Y_Min == Y_Max) // Đa giác cho ảnh chiếu là một đoạn thẳng
    {
        if ((Y_Min < 0) || (Height <= Y_Min)) return;

        int X_left = P[0].x, X_right = X_left;
        double Z_left = P[0].z, Z_right = Z_left;
        for (i = 1; i < Num_Vertex; i++)
        {
            if (X_left > P[i].x)
            {
                X_left = P[i].x; Z_left = P[i].z;
            }
            else if (X_right < P[i].x)

```

```

    {
        X_right = P[i].x; Z_right = P[i].z;
    }
}

ListIntersection = new IntersectionType[2];
ListIntersection[0].X = X_left; ListIntersection[0].Z =
Z_left;
ListIntersection[1].X = X_right; ListIntersection[1].Z =
Z_right;
FillLines(Y_Min, ListIntersection, 2, pBits, BytePerLine,
Rcolor, Gcolor, Bcolor, Z_Buffer, Width, Height);
delete[] ListIntersection;
return;
}

EdgeInfType * List_Edge = new EdgeInfType[Num_Vertex];

/* Trích xuất danh sách các cạnh của đa giác nhằm phục vụ cho
tiến trình tìm giao điểm với Scan_Line được hiệu quả */
int cnt = 0, IndexTop, IndexBottom;
for (i = 0; i<Num_Vertex; i++)
{
    if (i == Num_Vertex - 1)
        j = 0;
    else
        j = i+1;

    if (P[i].y <= P[j].y)
    {
        IndexTop = i; IndexBottom = j;
    }
    else
    {
        IndexTop = j; IndexBottom = i;
    }

    List_Edge[cnt].Y_Top = P[IndexTop].y;
    List_Edge[cnt].Y_Bottom = P[IndexBottom].y;

    List_Edge[cnt].X_Intersection = P[IndexTop].x;
    List_Edge[cnt].Z_Intersection = P[IndexTop].z;
}

```

```

List_Edge[cnt].X_Bottom = P[IndexBottom].x;
List_Edge[cnt].Z_Bottom = P[IndexBottom].z;

double DX = P[IndexBottom].x - P[IndexTop].x, DY =
P[IndexBottom].y - P[IndexTop].y, DZ = P[IndexBottom].z -
P[IndexTop].z;

if ((DY == 0) && (DX == 0)) /* Cạnh đa giác có độ dài zero
sẽ không đưa vào danh sách cạnh để xử lý */
    continue;

if (DY != 0)
{
    List_Edge[cnt].Xstep = DX / DY;
    List_Edge[cnt].Zstep = DZ / DY;
}
else
{
    List_Edge[cnt].Xstep = 0;
    List_Edge[cnt].Zstep = 0;
    if (List_Edge[cnt].X_Intersection < P[IndexBottom].x) /* Lấy giao điểm phía bên phải */
    {
        List_Edge[cnt].X_Intersection = P[IndexBottom].x;
        List_Edge[cnt].Z_Intersection = P[IndexBottom].z;
    }
}
cnt++;
}

int NumEdge = cnt;
ListIntersection = new IntersectionType[NumEdge];

for (int Yscan = Y_Min; Yscan <= Y_Max; Yscan++)
{
    cnt = 0;
    for (i = 0; i<NumEdge; i++)
    {
        if ((List_Edge[i].Y_Top <= Yscan) && (Yscan <=
List_Edge[i].Y_Bottom))
        {

```

```

        // chú ý làm tròn số thực về số nguyên
        ListIntersection[cnt].X =
int(List_Edge[i].X_Intersection + 0.5);
        // giữ nguyên số thực, không làm tròn về nguyên
        ListIntersection[cnt].Z = List_Edge[i].Z_Intersection;
        ListIntersection[cnt].EdgeIndex = i;
        cnt++;
        List_Edge[i].X_Intersection += List_Edge[i].Xstep;
        List_Edge[i].Z_Intersection += List_Edge[i].Zstep;
    }
}

// Bỏ qua nếu Y_Scan nằm ngoài phạm vi [0, Width-1]
if ((Yscan < 0) || (Yscan >= Height))
    continue; // Tiếp tục vòng lặp Yscan với dòng tiếp theo

// Sắp xếp các giao điểm
for (i = 0; i<cnt - 1; i++)
{
    long Min = ListIntersection[i].X, ChiSo = i;
    for (j = i + 1; j<cnt; j++)
        if (Min > ListIntersection[j].X)
    {
        Min = ListIntersection[j].X; ChiSo = j;
    }
    if (ChiSo != i)
    {
        IntersectionType tg = ListIntersection[i];
        ListIntersection[i] = ListIntersection[ChiSo];
        ListIntersection[ChiSo] = tg;
    }
}
// Xem xét với các giao điểm kép
int cnt2 = 0, k = 0;
while (k < cnt - 1)
{
    if (ListIntersection[k].X == ListIntersection[k + 1].X)
    {
        int Edge1 = ListIntersection[k].EdgeIndex, Edge2 =
ListIntersection[k + 1].EdgeIndex;

        if ((List_Edge[Edge1].Y_Top != List_Edge[Edge2].Y_Top)

```

```

&& (List_Edge[Edge1].Y_Bottom != List_Edge[Edge2].Y_Bottom) /*  

không nằm về cùng một phía */  

    k++; // bỏ qua giao điểm  

}  

if (cnt2 != k)  

    ListIntersection[cnt2] = ListIntersection[k];  

k++; cnt2++;  

}  

if ((k <= cnt-1)) //còn điểm cuối cùng  

{  

    if (cnt2 != k)  

        ListIntersection[cnt2] = ListIntersection[k];  

    cnt2++;  

}  

if ((cnt2 % 2) != 0)  

    cnt2--; // Bỏ bớt giao điểm sau cùng  

// Tô màu các đoạn  

FillLines(Yscan, ListIntersection, cnt2, pBits, BytePerLine,  

Rcolor, Gcolor, Bcolor, Z_Buffer, Width, Height);
}  

delete[] List_Edge;  

delete[] ListIntersection;
}

```

```

int ForwardIndex(int Num_Vertex, int CurrentIndex)
{
    if (CurrentIndex != (Num_Vertex - 1))
        return CurrentIndex + 1;
    else
        return 0;
}

int BackwardIndex(int Num_Vertex, int CurrentIndex)
{

```

```

if (CurrentIndex != 0)
    return CurrentIndex - 1;
else
    return (Num_Vertex - 1);
}

// Tô màu các đoạn
void FillLines(int Yscan, IntersectionType *ListIntersection, int
Num_Inter, BYTE *pBits, int BytePerLine, BYTE Rcolor, BYTE
Gcolor, BYTE Bcolor, double Z_Buffer[], int Width, int Height)

/* Ở đây *pBits là một con trỏ trả đến một mảng dữ liệu ảnh dạng
DIB chế độ 24-bit với số điểm ảnh bằng (Width * Height), và số
byte trên một dòng ảnh là BytePerLine, từ những thông tin này
chúng ta sẽ dễ dàng tính ra được địa chỉ ô nhớ chứa thông tin của
một điểm có tọa độ (x,y) sẽ là Address = y * BytePerLine + x * 3.
*/
{
    for (int k = 0; k < Num_Inter - 1; k += 2)
    {
        // Xét đoạn thẳng đi từ ListIntersection[k] đến
        ListIntersection[k + 1]
        double Dx = ListIntersection[k + 1].X - ListIntersection[k].X;
        double ZStep;
        if (Dx == 0)
            ZStep = 0;
        else
            ZStep = (ListIntersection[k + 1].Z - ListIntersection[k].Z) / Dx; /* Chia số thực */

        double Z = ListIntersection[k].Z;
        int X_end = ListIntersection[k + 1].X;

        int Pos = Yscan * BytePerLine;

        for (int x = ListIntersection[k].X; x <= X_end; x++)
        {
            if ((0 <= x) && (x < Width))
            {

                if (Z_Buffer[Yscan * Width + x] > Z) // Vẽ và cập nhật
            }
        }
    }
}

```

độ sâu

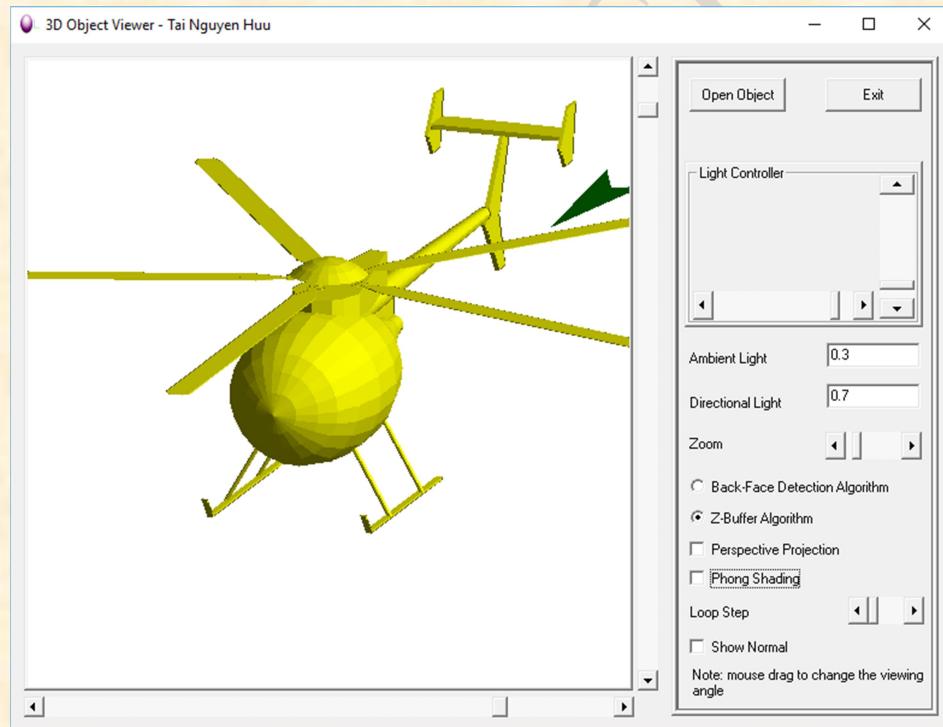
```

    {
        int Pos2 = Pos + x * 3;
        pBits[Pos2] = Bcolor; pBits[Pos2 + 1] = Gcolor;
        pBits[Pos2 + 2] = Rcolor;
        Z_Buffer[Yscan*Width + x] = Z;
    }
}

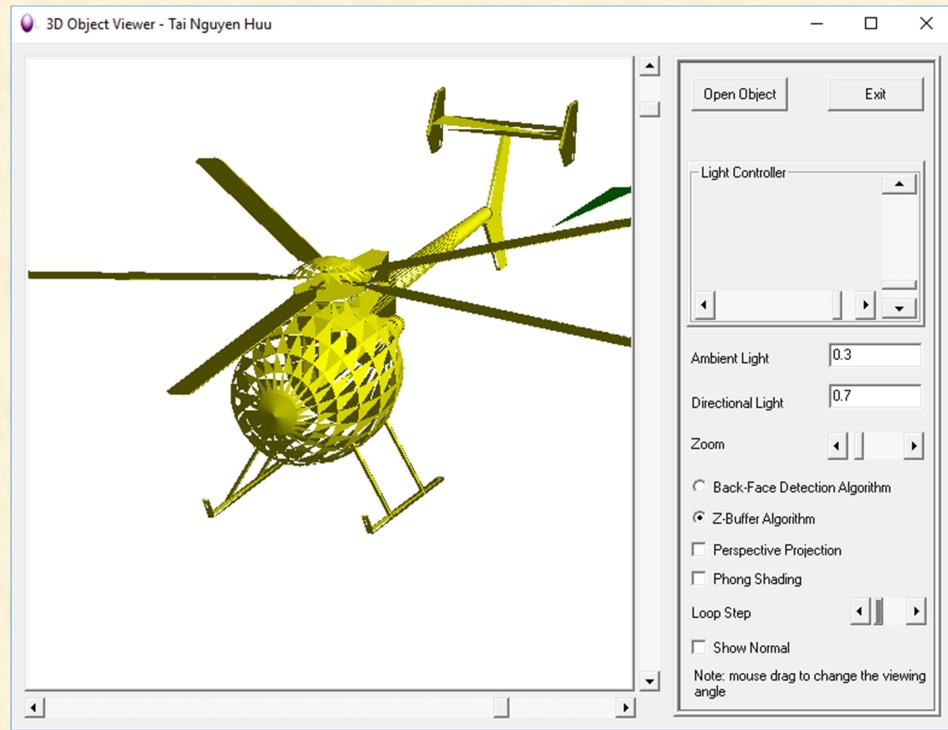
Z += ZStep;
}
}
}

```

❖ Chạy thực nghiệm cho hình ảnh minh họa dưới đây:



Hình 6.16. Mô hình trực thăng (Hughes 500) được xử lý mặt khuất theo giải thuật vùng đệm độ sâu, có xử lý thêm ván đè chiếu sáng nhằm tạo ra hình ảnh trung thực



Hình 6.17. Mô hình trực thăng (Hughes 500) với một số mặt được lược bỏ để có thể quan sát được phần bên trong của đối tượng. Xử lý mặt khuất theo giải thuật vùng đậm độ sâu

#### 4. BÀI TẬP CUỐI CHƯƠNG

1. Cài đặt chương trình cho phép biểu diễn và quan sát vật thể 3D theo mô hình "các mặt đa giác". Trong đó, sử dụng giải thuật Depth-Sorting để xử lý vấn đề mặt khuất.
2. Xây dựng chương trình mô phỏng thế giới thực 3 chiều với nhiều đối tượng chuyển động theo thời gian, và người sử dụng có được nhiều lựa chọn giải thuật khử mặt khuất khác nhau, đồng thời có thể chuyển đổi qua lại giữa 2 phép chiếu song song và phối cảnh.

## Chương 7

# CÁC MÔ HÌNH CHIẾU SÁNG

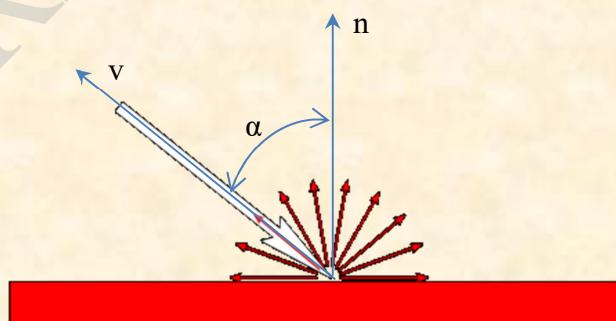
Khi biểu diễn các đối tượng 3 chiều, một yếu tố không thể bỏ qua để tăng tính thực của đối tượng đó là tạo bóng sáng cho vật thể. Để thực hiện được điều này, chúng ta cần phải lần lượt tìm hiểu các dạng nguồn sáng có trong tự nhiên, cũng như các tính chất đặc trưng khác nhau của mỗi loại nguồn sáng. Từ đó, đưa ra các giải pháp kỹ thuật khác nhau nhằm thể hiện sự tác động của các nguồn sáng khác nhau lên hình ảnh của đối tượng.

### 1. NGUỒN SÁNG XUNG QUANH

Nguồn ánh sáng xung quanh (Ambient Light) là mức sáng trung bình, tồn tại trong một vùng không gian. Một không gian lý tưởng là không gian mà tại đó mọi vật đều được cung cấp một lượng ánh sáng lên bề mặt là như nhau, từ mọi phía ở mọi nơi. Thông thường, ánh sáng xung quanh được xác định với một mức cụ thể gọi là mức sáng xung quanh của vùng không gian mà vật thể đó cư ngụ, sau đó chúng ta cộng với cường độ sáng có được từ các nguồn sáng khác để có được cường độ sáng cuối cùng lên một điểm hay một mặt của vật thể.

### 2. NGUỒN SÁNG ĐỊNH HƯỚNG

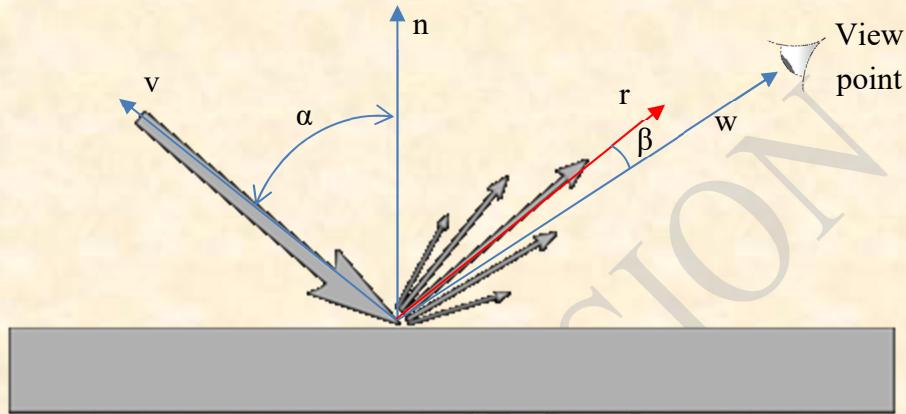
#### 2.1. Khái niệm



Hình 7.1. Sự khuếch tán của ánh sáng trên các bề mặt

Nguồn sáng định hướng (Directional Light) giống như những gì mà

mặt trời cung cấp cho chúng ta. Nó bao gồm một tập các tia sáng song song, bất kể cường độ của chúng có giống nhau hay không. Khi ánh sáng định hướng chiếu đến bề mặt của vật thể, sẽ sinh ra hai hình thức đáp ứng gồm: Khuếch tán; và phản chiếu (hay phản xạ). Hình 7.1 và Hình 7.2 lần lượt minh họa cho hai hình thức đáp ứng.



Hình 7.2. Sự phản xạ của ánh sáng trên các bề mặt

## 2.2. Tính toán mô phỏng

- ❖ Đối với hiện tượng khuếch tán: Nếu gọi  $\alpha$  là góc giữa tia tới, ngược hướng với tia ánh sáng tới nhằm thuận tiện trong tính toán, với vector pháp tuyến của bề mặt thì  $\cos(\alpha)$  phụ thuộc vào tia tới  $v$  và vector pháp tuyến  $n$  của bề mặt theo công thức:

$$\cos(\alpha) = \frac{\vec{v} \cdot \vec{n}}{|\vec{v}| \cdot |\vec{n}|} \quad (7.1)$$

Trong công thức trên  $\cos(\alpha)$  bằng tích vô hướng của  $v$  và  $n$  chia cho tích độ lớn của chúng. Nếu chúng ta đã chuẩn hóa độ lớn của các vector  $v$  và  $n$  về độ lớn bằng 1 từ trước thì chúng ta có thể tính giá trị trên một cách hiệu quả hơn như sau:

$$\cos(\alpha) = \text{tích vô hướng } (v, n) = v.x * n.x + v.y * n.y + v.z * n.z$$

Vì góc tới  $\alpha$  có giá trị giao động trong khoảng từ 0 đến  $90^\circ$ , nên  $\cos(\alpha)$  có giá trị giao động từ 0 đến 1. Từ đó, trong lĩnh vực đồ họa máy tính người ta thường sử dụng hàm  $\cos(\alpha)$  như là hàm phản ánh mối quan

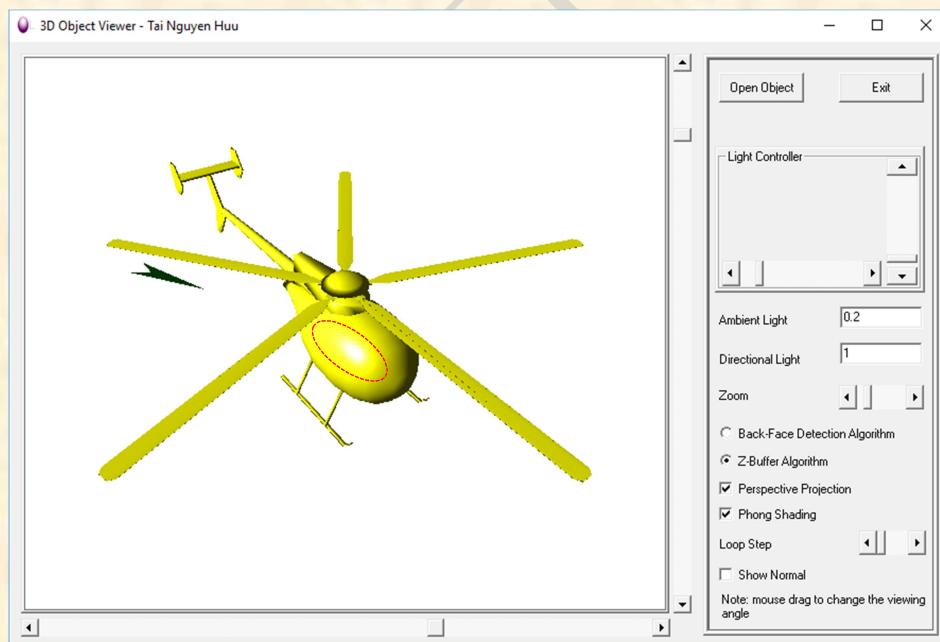
hệ của cường độ ánh sáng tới (*ánh sáng định hướng*) và ánh sáng khuếch tán, như sau:

$$\text{Cường độ khuếch tán} = \text{cường độ định hướng} * \cos(\alpha) \quad (7.2)$$

Qua các công thức (7.1) và (7.2), chúng ta có thể tính được cường độ của ánh sáng khuếch tán trên bề mặt, khi biết được cường độ của ánh sáng định hướng cũng như vector pháp tuyến của bề mặt và tia sáng tới.

- ❖ Đối với hiện tượng phản chiếu (hay phản xạ): Thông thường, chúng ta chỉ quan sát được tia sáng phản chiếu từ bề mặt vật thể trong một phạm vi rất hẹp xung quanh tia phản xạ  $r$  (xem Hình 2.2). Nếu gọi  $\beta$  là góc tạo bởi tia phản xạ  $r$  và tia hướng nhìn  $w$ , thì cường độ ánh sáng phản chiếu mà mắt người quan sát có thể cảm thụ được cho bởi công thức:

$$\text{Cường độ phản chiếu} = \text{Cường độ định hướng} * \cos(\beta)^{10} \quad (7.3)$$



Hình 7.3. Mô phỏng hiện tượng phản chiếu trên bề mặt đối tượng với phần sáng trắng được đánh dấu bởi vòng màu đỏ

Trong công thức (7.3), giá trị  $\text{Cos}(\beta)$  được biến đổi qua hàm mũ (cơ số mũ là 10) nhằm làm suy giảm (hay triệt tiêu) giá trị của *cường độ ánh sáng phản chiếu* một cách nhanh chóng khi góc  $\beta$  được mở rộng. Hay nói cách khác là người quan sát sẽ nhận được lượng ánh sáng phản chiếu gần như bằng không nếu hướng quan sát của họ tạo với tia phản xạ  $r$  một góc lớn, giúp tạo cho hình ảnh có bóng sáng phản chiếu hẹp gần giống với hình ảnh quan sát trong thế giới thực.

### 2.3. Cài đặt giải thuật

Dưới đây là phần trình bày các hàm phục vụ cho việc vẽ đối tượng 3D đặc biệt, theo giải thuật chọn lọc mặt sau có tính đến vấn đề chiếu sáng của nguồn sáng xung quanh và nguồn sáng định hướng.

#### Bài thực nghiệm số 6:

```
void CObject_3D::DrawObject(Vector VLight, double Ambient_Light,
double Directional_Light, CDC *p_DC, CRect R)
{
    long X_C=R.CenterPoint().x;
    long Y_C=R.CenterPoint().y;

    for (unsigned int i = 0; i<this->NumSurface; i++)
    {
        Surface_3D* p=&ListSurface[i];
        Vector View_VT={0,0,-1};
        if (ScalarProductofVectors(p->Normal,View_VT)>0) /* Nếu tích
vô hướng >0 */
        {
            double Inensity = Ambient_Light +
                Directional_Light* Cosine(VLight,p-
                >Normal);

            int * pIndex=p->ListIndex;
            CPoint* ListPoint;

            ListPoint = new CPoint[p->NumVert];
            for (unsigned int k=0; k<p->NumVert; k++)
        }
    }
}
```

```

{
    unsigned int Index=pIndex[k];
    ListPoint[k].x =
int(ListVertexInViewCoordinates[Index].x + 0.5) + X_C;
    ListPoint[k].y = -
int(ListVertexInViewCoordinates[Index].y + 0.5) + Y_C; // Đảo
chiều trục Y rồi tịnh tiến
}

CPen Pen(PS_NULL, 1, RGB(0, 0, 0));

BYTE R=GetRValue(p->Color);
BYTE G=GetGValue(p->Color);
BYTE B=GetBValue(p->Color);
double H,S,L;

CMyColor::RGBToHSL(R,G,B,&H,&S,&L);

L=L*Ientity;
if (L>1) L=1;

CMyColor::HSLToRGB(H,S,L,&R,&G,&B);

CBrush Brush(RGB(R,G,B));

CPen* OldPen=p_DC->SelectObject(&Pen);
CBrush* OldBrush=p_DC->SelectObject(&Brush);
int OldFillMode=p_DC->SetPolyFillMode(ALTERNATE);

p_DC->Polygon(ListPoint, p->NumVert);

delete[] ListPoint;

p_DC->SelectObject(OldPen);
p_DC->SelectObject(OldBrush);
if (OldFillMode!=ALTERNATE)
    {p_DC->SetPolyFillMode(OldFillMode);}
}
}
}

```

### 3. NGUỒN SÁNG ĐIỂM

Nguồn sáng định hướng là tương đương với nguồn sáng điểm đặt ở vô tận. Nhưng khi nguồn sáng điểm được mang đến gần đối tượng, thì các tia sáng từ nó phát ra không còn song song nữa mà được toả ra theo mọi hướng dưới dạng hình cầu. Vì thế, các tia sáng sẽ rời xuống các điểm trên bề mặt dưới các góc khác nhau. Giả sử vector pháp tuyến của mặt là  $n = (x_n, y_n, z_n)$ , điểm đang xét có tọa độ là  $(x_0, y_0, z_0)$  và nguồn sáng điểm có tọa độ là  $(plx, ply, plz)$  thì ánh sáng sẽ rời đến điểm đang sét theo vector  $(x_0 - plx, y_0 - ply, z_0 - plz)$ , hay tia tới:

$$\mathbf{a} = (plx - x_0, ply - y_0, plz - z_0).$$

Và từ đó cường độ sáng tại điểm đang xét sẽ phụ thuộc vào  $\cos(\theta)$  giữa  $n$  và  $a$  như đã trình bày trong phần nguồn sáng định hướng.

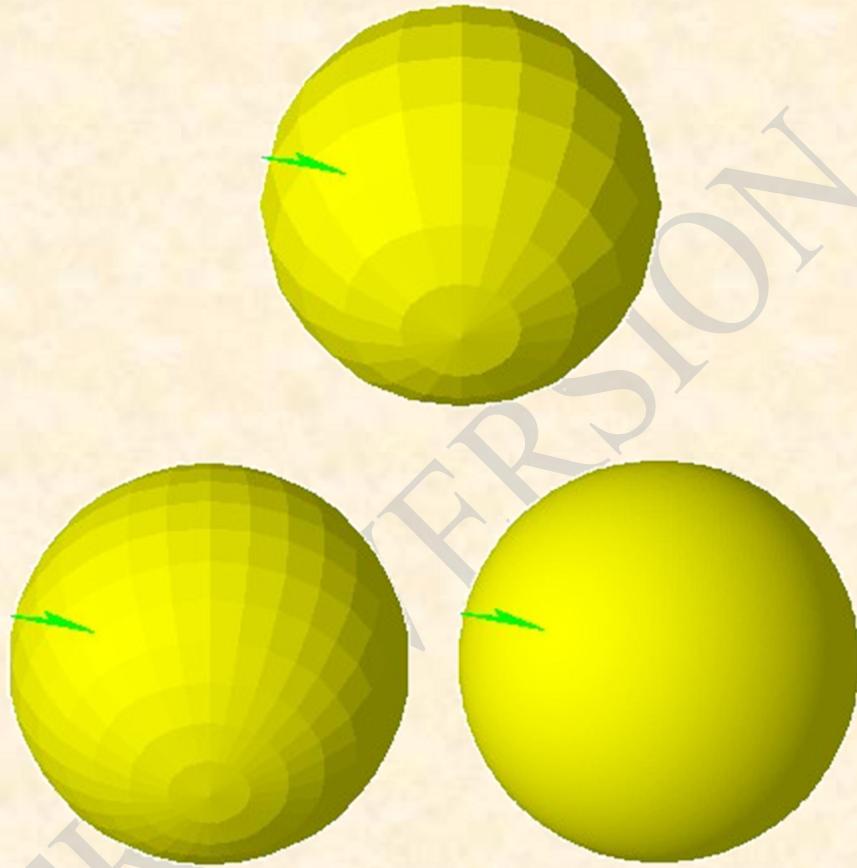
Vậy với nguồn sáng điểm, chúng ta cần tính tia tới cho mọi điểm trên bề mặt, từ đó kết hợp với vector pháp tuyến của mặt để tính được cường độ sáng tại điểm đó, nếu tính toán trực tiếp thì có thể mất khá nhiều thời gian do phải tính vector  $a$  và tính  $\cos(\theta)$  thông qua công thức (7.1) với tất cả các điểm trên mặt.

## 4. MÔ HÌNH BÓNG PHONG

### 4.1. Khái niệm

Mô hình bóng Phong là một phương pháp vẽ bóng, tạo cho đối tượng 3D có hình dáng cong có một cái nhìn có tính thực hơn. Phương pháp này đặt cơ sở trên thực tế sau: Đối với các đối tượng 3D có bề mặt cong thì người ta thường xấp xỉ bề mặt cong của đối tượng bằng nhiều mặt đa giác phẳng, ví dụ như một mặt cầu có thể xấp xỉ bởi một tập các mặt đa giác phẳng có kích thước nhỏ sắp xếp lại, khi số đa giác xấp xỉ tăng lên (có nghĩa là diện tích mặt đa giác nhỏ lại) thì tính thực của mặt cầu sẽ tăng, sẽ cho chúng ta cảm giác mặt cầu trông tròn trịa hơn, mịn và cong hơn. Tuy nhiên, khi số đa giác xấp xỉ một mặt cong tăng thì khối lượng tính toán và lưu trữ cũng tăng theo tỷ lệ thuận theo số mặt, điều đó dẫn đến tốc độ thực hiện sẽ trở nên chậm chạp hơn. Chúng ta hãy thử với một ví dụ sau: Để mô phỏng một mặt cầu, người ta xấp xỉ nó bởi 200 mặt thì cho chúng ta một cảm giác hơi gồ ghề, nhưng với 450 mặt thì chúng ta

thấy nó mịn và tròn trịa hơn, song khi số mặt là 16.200 thì cho chúng ta cảm giác hình cầu rất tròn và mịn (xem Hình 7.4). Tuy hình ảnh mặt cầu với 16,200 mặt đa giác thì mịn hơn so với 200 mặt, song lượng tính toán phải thực hiện trên mỗi đa giác cũng tăng lên gấp  $16.200/200 = 81$  lần.

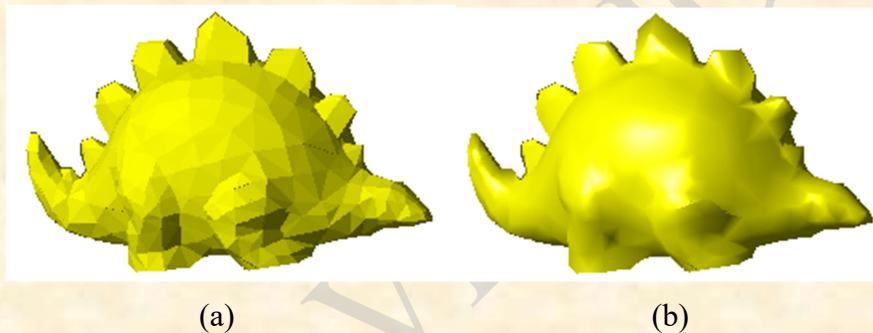


Hình 7.4. Các hình cầu được số hóa theo mô hình các mặt đa giác với số mặt lần lượt là (a) 200 mặt, (b) 450 mặt, (c) 16.200 mặt

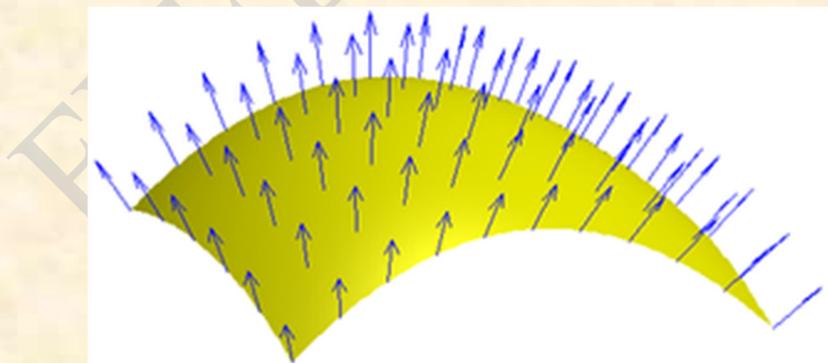
Vấn đề vẫn còn nảy sinh một khi chúng ta phóng lớn hay thu nhỏ vật thể. Nếu chúng ta phóng lớn thì rõ ràng là các đa giác cũng được phóng lớn theo cùng tỷ lệ, dẫn đến hình ảnh về các mặt đa giác lại hiện rõ và gây ra cảm giác gồ ghề không tròn mịn. Ngược lại, khi chúng ta thu nhỏ nếu số đa giác xấp xỉ lớn thì sẽ dẫn đến tình trạng các đa giác quá nhỏ, chồng chất lên nhau không cần thiết.

## 4.2. Tính toán mô phỏng

Để giải quyết vấn đề trên, chúng ta có thể tiến hành theo phương pháp tô bóng Phong. Mô hình bóng Phong tạo cho đối tượng một cái nhìn giống như là nó có nhiều mặt đa giác bằng cách vẽ mỗi mặt không chỉ với một cường độ sáng mà vẽ với nhiều cường độ sáng khác nhau trên các vùng khác nhau, làm cho mặt phẳng nom như bị cong. Thực chất, chúng ta cảm nhận được độ cong của các mặt cong do hiệu ứng ánh sáng khi chiếu lên bề mặt, tại các điểm trên mặt cong sẽ có pháp vector khác nhau nên chúng sẽ đón nhận và phản xạ ánh sáng khác nhau, từ đó chúng ta sẽ cảm nhận được các mức độ sáng khác nhau trên cùng một mặt cong.



Hình 7.5. Minh họa kết quả xử lý tô bóng (a) Tô bóng thường, (b) Tô bóng theo giải thuật Phong

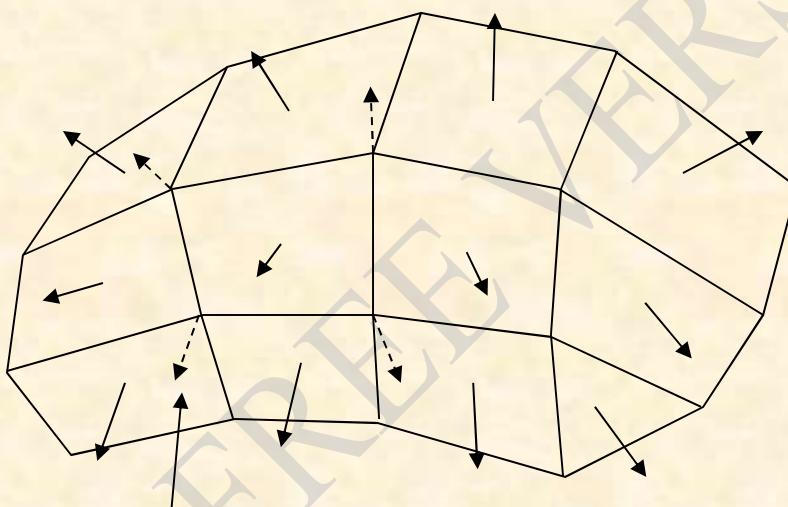


Hình 7.6. Vector pháp tuyến của các điểm trên một mặt cong

Thường thì mỗi mặt đa giác có một vector pháp tuyến, và như phân trên đã trình bày, vector pháp tuyến đó được dùng để tính cường độ của

ánh sáng phản xạ trên bề mặt của đa giác, rồi suy ra cường độ sáng của mặt. Tuy nhiên, mô hình Phong lại xem một đa giác không chỉ có một vector pháp tuyến, mà mỗi đỉnh của mặt đa giác lại có một vector pháp tuyến với giá trị các vector là không giống nhau. Từ vector pháp tuyến của các đỉnh chúng ta sẽ nội suy ra được vector pháp tuyến của từng điểm trên mặt đa giác, dựa vào vector pháp tuyến của từng điểm mà tính được cường độ sáng của mỗi điểm. Như thế, các điểm trên cùng một mặt của đa giác phẳng sẽ có cường độ sáng khác nhau, *do pháp vector nội suy được khác nhau nên tính ra độ sáng khác nhau*, từ đó cho hình ảnh thị giác là một mặt cong chứ không phải là một mặt phẳng.

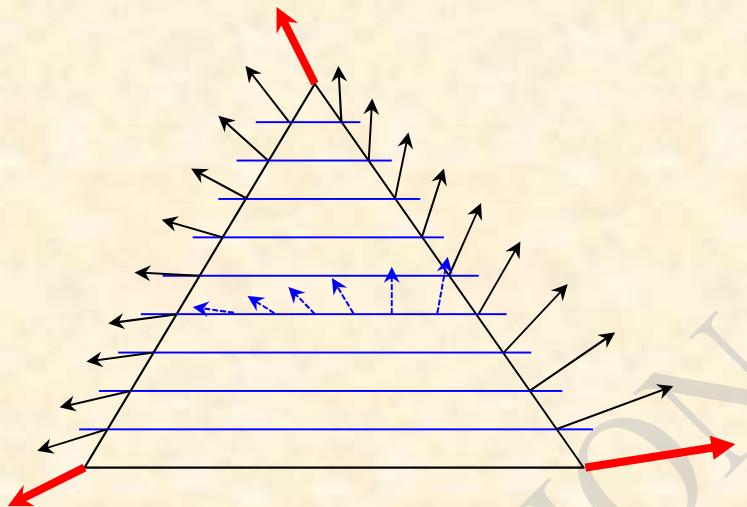
Thực chất, mỗi mặt đa giác chỉ có một vector pháp tuyến, song phương pháp Phong tính toán vector trung bình tại mỗi đỉnh của đa giác bằng cách: Lấy trung bình cộng các vector pháp tuyến của các đa giác có chứa đỉnh đang xét.



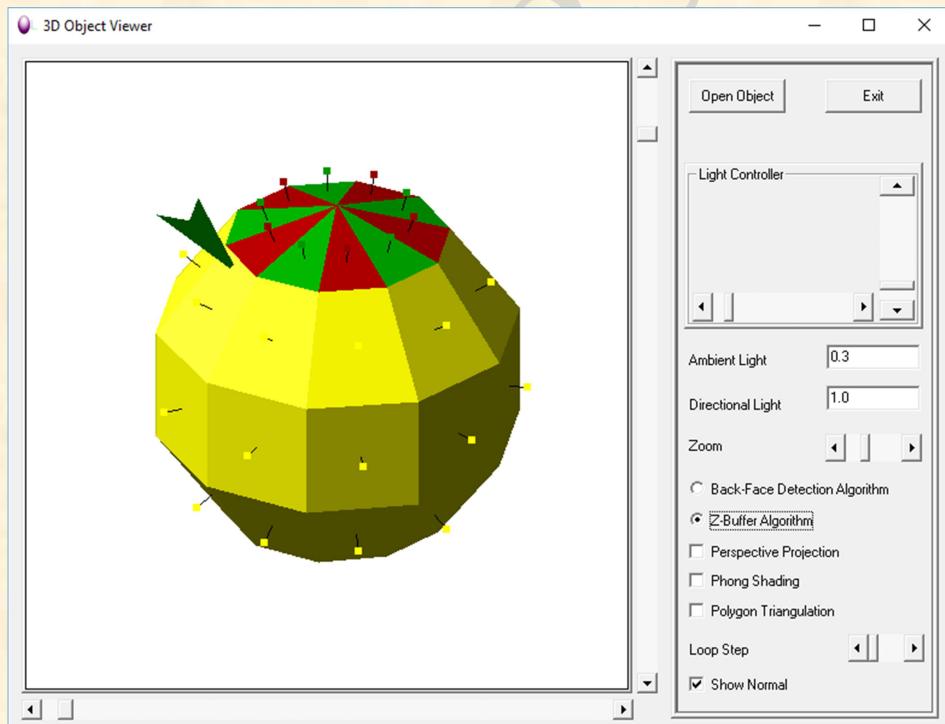
Vector pháp tuyến tại đỉnh, bằng trung bình cộng của các vector pháp tuyến bề mặt lân cận

Hình 7.7. Minh họa vector pháp tuyến tại các đỉnh của đa giác

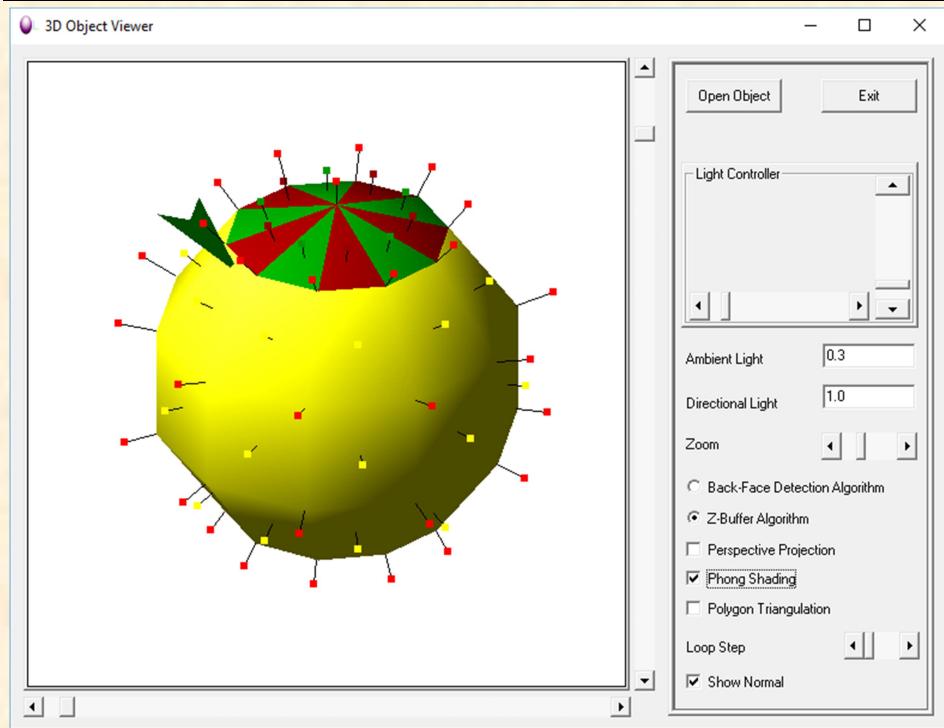
Việc nội suy vector pháp tuyến của từng điểm trên mặt đa giác được thực hiện tương tự như việc nội suy độ sâu trong giải thuật “vùng đệm độ sâu” đã được trình bày trong chương trước.



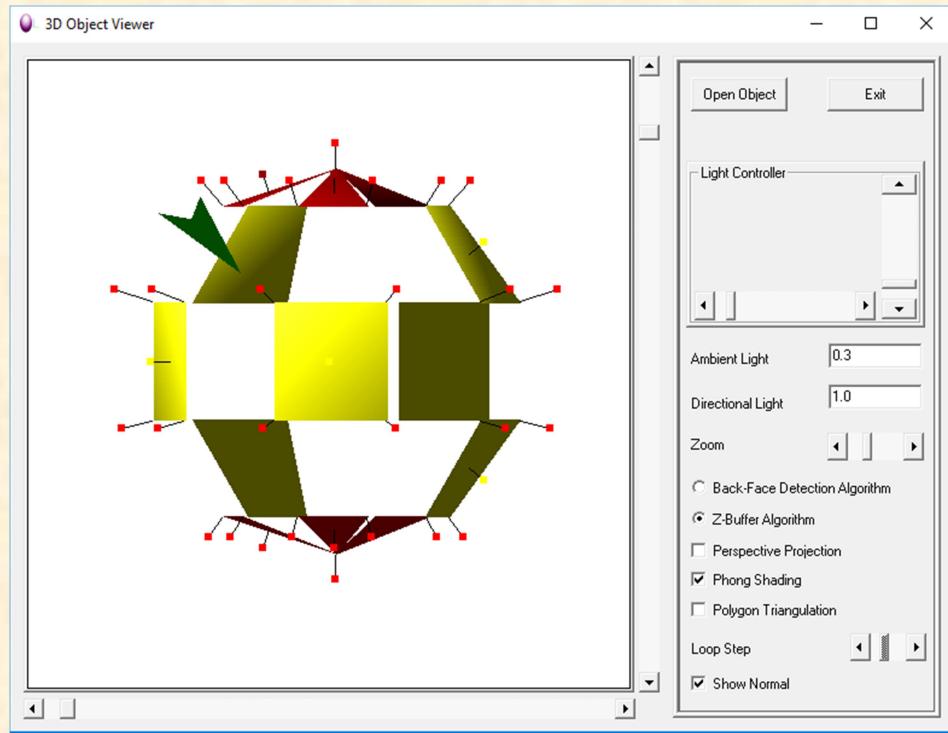
Hình 7.8. Minh họa các bước nội suy vector pháp tuyến cho từng điểm trên mặt đa giác



Hình 7.9. Hình ảnh thực nghiệm minh họa hình cầu 50 mặt không sử dụng phương pháp tô bóng cong (với chỉ một vector pháp tuyến cho mỗi bề mặt)



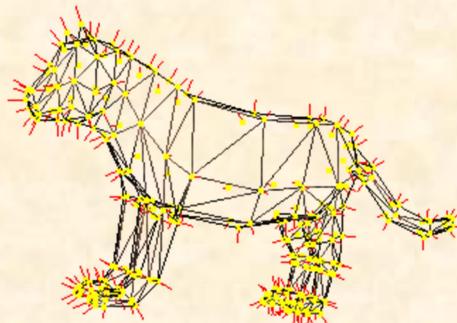
Hình 7.10. Hình ảnh thực nghiệm minh họa hình cầu 50 mặt sử dụng phương pháp tô bóng Phong, với các vector pháp tuyến tại các đỉnh của bề mặt (hay đa giác) không cùng phuong với vector pháp tuyến bề mặt mà có xu hướng ngả ra bên ngoài như mặt cong



Hình 7.11. Minh họa như Hình 7.10, nhưng được giản lược bớt một số mặt nhằm giúp quan sát rõ hơn các vector pháp tuyến tại đỉnh và bề mặt.

Hình ảnh cho thấy các vector pháp tuyến tại đỉnh của bề mặt có xu hướng nghiêng ra bốn phía bên ngoài như của một mặt cong

Hiện nay, trong các tiện ích tạo mô hình, người ta cho phép chúng ta nhập và điều chỉnh hướng cho các vector nút tại các đỉnh của đa giác thay vì tính nội suy từ các vector pháp tuyến. Như hình minh họa sau:



Hình 7.12. Minh họa đối tượng với các vector nút

### 4.3. Cài đặt giải thuật

Kết thừa những gì đã xây dựng được ở chương 6. Dưới đây là phần trình bày các cấu trúc và hàm phục vụ cho việc vẽ đối tượng 3D theo giải thuật z-buffer, cùng với việc xử lý vấn đề chiếu sáng của nguồn sáng xung quanh và nguồn sáng định hướng theo phương pháp tô bóng Phong.

#### Bài thực nghiệm số 7:

- ❖ Bước 1: Phần khai báo các cấu trúc và các hàm xử lý:

```

struct RGB_Color
{
    BYTE B, G, R;
};

struct Point_Z_Normal
{
    int x, y;
    double z;
    Vector Normal;
};

class CObject_3D : public CObject
{
public:
    ... // Các khai báo đã có ở phần chương 6

    Vector * ListVertexNormal = NULL; /* Vector pháp tuyến tại
    mỗi đỉnh */
    Vector * ListVertexNormalInViewCoordinates = NULL; /* Vector
    pháp tuyến tại mỗi đỉnh được chuyển sang hệ quan sát */

    void DrawObject_ZBufferAlgorithmAndPhongShading(Vector VLight,
    double Ambient_Light, double Directional_Light, BYTE *pBits, int
    BytePerLine, CRect R, double * Z_Buffer, int ScanSubfaceStep);
    /* Vẽ đối tượng theo giải thuật z-buffer kết hợp với tô bóng
    Phong */

    void FillPoly_Z_Phong(Vector VLight, double Ambient_Light,
    double Directional_Light, Point_Z_Normal P[], int Num_Vertex,
    BYTE *pBits, int BytePerLine, BYTE Rcolor, BYTE Gcolor, BYTE
    Bcolor, double Z_Buffer[], int Width, int Height); /* Tô màu một
    
```

*đa giác là hình chiếu của một bề mặt đa giác trong không gian 3 chiều được chiếu lên mặt phẳng chiếu (2D) theo giải thuật z-buffer kết hợp với tô bóng Phong \*/*

```

void TransVertexNormalToViewCoordinates(unsigned int R,
unsigned int DeltaAngle, unsigned int PhiAngle);
static Vector NormalVector(Vertex P1, Vertex P2, Vertex P3);
static Vector CalVector(Vertex P1, Vertex P2) {
    Vector v;
    v.x = P2.x - P1.x;
    v.y = P2.y - P1.y;
    v.z = P2.z - P1.z;
    return v;
}
static Vector Vector3D(double x, double y, double z)
{
    Vector v; v.x = x; v.y = y; v.z = z; return v; }
static Vector SubtractVector(Vector v1, Vector v2)
{
    return Vector3D(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z); }
static Vector AddVector(Vector v1, Vector v2)
{
    return Vector3D(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z); }
static Vector DivVector(Vector v1, Vector v2)
{
    return Vector3D(v1.x / v2.x, v1.y / v2.y, v1.z / v2.z); }
static Vector DivVector(Vector v1, double x)
{
    return Vector3D(v1.x / x, v1.y / x, v1.z / x); }
static Vector MulVector(Vector v1, Vector v2)
{
    return Vector3D(v1.x * v2.x, v1.y * v2.y, v1.z * v2.z); }
static Vector MulVector(Vector v1, double x)
{
    return Vector3D(v1.x * x, v1.y * x, v1.z * x); }

static RGB_Color CalColorInPoint(Vector Normal, Vector VLight,
double Ambient_Light, double Directional_Light, double Hue,
double Saturation, double Lightness);
};
```

- ❖ Bước 2: Xây dựng mã cho các hàm xử lý:

```

void
CObject_3D::DrawObject_ZBufferAlgorithmAndPhongShading(Vector
VLight, double Ambient_Light, double Directional_Light, BYTE
*pBits, int BytePerLine, CRect Rect, double * Z_Buffer, int
ScanSubfaceStep)
{
    long X_C = Rect.CenterPoint().x;
```

```

long Y_C = Rect.CenterPoint().y;

for (unsigned int i = 0; i<this->NumSurface; i += ScanSubfaceStep)
{
    Surface_3D* p = &ListSurface[i];
    int * pIndex = p->ListIndex;
    Point_Z_Normal *ListPoint;

    ListPoint = new Point_Z_Normal[p->NumVert];

    for (unsigned int k = 0; k<p->NumVert; k++)
    {
        unsigned int Index = pIndex[k];
        ListPoint[k].x = int(ListVertexInViewCoordinates[Index].x
+ 0.5) + X_C;
        ListPoint[k].y = -
int(ListVertexInViewCoordinates[Index].y + 0.5) + Y_C; // Đảo
chiều trực Y rồi tính tiến
        ListPoint[k].z = ListVertexInViewCoordinates[Index].z;
        ListPoint[k].Normal =
ListVertexNormalInViewCoordinates[Index];

    }

    BYTE R = GetRValue(p->Color);
    BYTE G = GetGValue(p->Color);
    BYTE B = GetBValue(p->Color);

    this->FillPoly_Z_Phong(VLight, Ambient_Light,
Directional_Light, ListPoint, p->NumVert, pBits, BytePerLine, R,
G, B, Z_Buffer, Rect.Width(), Rect.Height());

    delete[] ListPoint;
}
}

/* Trong đó FillPoly_Z_Phong cần các khai báo cấu trúc cục bộ và
các hàm như sau: */

struct EdgeInfType2 {
    int Y_Top;

```

```

    double X_Intersection;
    double Z_Intersection;
    Vector Normal_Intersection;
    double Xstep;
    double Zstep;
    Vector Normalstep;
    int Y_Bottom;
    int X_Bottom;
    double Z_Bottom;
    Vector Normal_Bottom;
};

struct IntersectionType2 {
    int X;
    double Z;
    Vector Normal;
    int EdgeIndex;
};

RGB_Color CObject_3D::CalColorInPoint(Vector Normal, Vector
VLight, double Ambient_Light, double Directional_Light, double
Hue, double Saturation, double Lightness)
{
    // Ambient Light
    // Directional Lights
    // Reflection_Light = 2(Normal . VLight)Normal - VLight
    double x = CObject_3D::ScalarProductofVectors(Normal, VLight);
    Vector Reflection_Light =
CObject_3D::SubtractVector(CObject_3D::MulVector(Normal, 2*x),
VLight);
    Vector DirectView = CObject_3D::Vector3D(0, 0, -1);

    double Cos_Reflection = CObject_3D::Cosine(Reflection_Light,
DirectView);
    if (Cos_Reflection < 0) Cos_Reflection = 0;

    double Cos = Cosine(VLight, Normal);

    double Inentity = Ambient_Light + Directional_Light*Cos +
Directional_Light*pow(Cos_Reflection, 15);

    Lightness = Lightness*Inentity;
}

```

```

    if (Lightness>1) Lightness = 1;

    RGB_Color Color;
    CMyColor::HSLToRGB(Hue, Saturation, Lightness, Color.R,
    Color.G, Color.B);
    return Color;
}

// Tô màu các đoạn
void FillLines2(Vector VLight, double Ambient_Light, double
Directional_Light, int Yscan, IntersectionType2
*ListIntersection, int Num_Inter, BYTE *pBits, int BytePerLine,
double Hue, double Saturation, double Lightness, double
Z_Buffer[], int Width, int Height)
{
    for (int k = 0; k < Num_Inter - 1; k += 2)
    {
        // Xét đoạn thẳng đi từ ListIntersection[k] đến
        ListIntersection[k + 1]
        double Dx = ListIntersection[k + 1].X -
        ListIntersection[k].X;
        double ZStep;
        Vector NormalStep;
        if (Dx == 0)
        {
            ZStep = 0;
            NormalStep.x = 0; NormalStep.y = 0; NormalStep.z = 0;
        }
        else
        {
            ZStep = (ListIntersection[k + 1].Z -
            ListIntersection[k].Z) / Dx; //Chia số thực
            Vector V = CObject_3D::SubtractVector(ListIntersection[k
            + 1].Normal, ListIntersection[k].Normal);
            NormalStep = CObject_3D::DivVector(V, Dx);

        }

        double Z = ListIntersection[k].Z;
        int X_end = ListIntersection[k + 1].X;
        Vector Norm = ListIntersection[k].Normal;
    }
}

```

```

int Pos = Yscan*BytePerLine;

for (int x = ListIntersection[k].X; x <= X_end; x++)
{
    if ((0 <= x) && (x < Width))
    {
        if (Z_Buffer[Yscan*Width + x] > Z) // Vẽ và cập nhật
độ sâu
        {
            int Pos2 = Pos + x * 3;
            RGB_Color *Color = (RGB_Color *)pBits[Pos2];

            *Color = CObject_3D::CalColorInPoint(Norm, VLight,
Ambient_Light, Directional_Light, Hue, Saturation, Lightness);

            Z_Buffer[Yscan*Width + x] = Z;
        }
    }

    Z += ZStep;
    Norm.x += NormalStep.x;
    Norm.y += NormalStep.y;
    Norm.z += NormalStep.z;
}

void CObject_3D::FillPoly_Z_Phong(Vector VLight, double
Ambient_Light, double Directional_Light, Point_Z_Normal P[], int
Num_Vertex, BYTE *pBits, int BytePerLine, BYTE Rcolor, BYTE
Gcolor, BYTE Bcolor, double Z_Buffer[], int Width, int Height)
{
    // Xử lý với đầu vào là những đa giác lồi
    double Hue, Saturation, Lightness;
    CMycolor::RGBToHSL(Rcolor, Gcolor, Bcolor, Hue, Saturation,
Lightness);

    IntersectionType2 *ListIntersection;
    int i, j;
    int Y_Min = P[0].y, Y_Max = Y_Min;
}

```

```

// Tim Min Max
for (i = 1; i < Num_Vertex; i++)
{
    if (Y_Min>P[i].y)
        Y_Min = P[i].y;
    else
        if (Y_Max<P[i].y)
            Y_Max = P[i].y;
}

if (Y_Min == Y_Max) // Đa giác cho ánh chiếu là một đoạn thẳng
{
    if ((Y_Min < 0) || (Height <= Y_Min)) return;

    int X_left = P[0].x, X_right = X_left;
    double Z_left = P[0].z, Z_right = Z_left;
    Vector Norm_left = P[0].Normal, Norm_right = Norm_left;
    for (i = 1; i < Num_Vertex; i++)
    {
        if (X_left > P[i].x)
        {
            X_left = P[i].x; Z_left = P[i].z; Norm_left =
P[i].Normal;
        }
        else if (X_right < P[i].x)
        {
            X_right == P[i].x; Z_right = P[i].z; Norm_right =
P[i].Normal;
        }
    }

    ListIntersection = new IntersectionType2[2];
    ListIntersection[0].X = X_left; ListIntersection[0].Z =
Z_left; ListIntersection[0].Normal = Norm_left;
    ListIntersection[1].X = X_right; ListIntersection[1].Z =
Z_right; ListIntersection[1].Normal = Norm_right;
    FillLines2(VLight, Ambient_Light, Directional_Light, Y_Min,
ListIntersection, 2, pBits, BytePerLine, Hue, Saturation,
Lightness, Z_Buffer, Width, Height);

    delete[] ListIntersection;
    return;
}

```

}

```

EdgeInfType2 * List_Edge = new EdgeInfType2[Num_Vertex];
/* Trích xuất danh sách các cạnh của đa giác nhằm phục vụ cho
tiến trình tìm giao điểm với Scan_Line được hiệu quả */
int cnt = 0, IndexTop, IndexBottom;
for (i = 0; i<Num_Vertex; i++)
{
    if (i == Num_Vertex - 1)
        j = 0;
    else
        j = i + 1;

    if (P[i].y <= P[j].y)
    {
        IndexTop = i; IndexBottom = j;
    }
    else
    {
        IndexTop = j; IndexBottom = i;
    }

    List_Edge[cnt].Y_Top = P[IndexTop].y;
    List_Edge[cnt].Y_Bottom = P[IndexBottom].y;

    List_Edge[cnt].X_Intersection = P[IndexTop].x;
    List_Edge[cnt].Z_Intersection = P[IndexTop].z;
    List_Edge[cnt].Normal_Intersection = P[IndexTop].Normal;

    List_Edge[cnt].X_Bottom = P[IndexBottom].x;
    List_Edge[cnt].Z_Bottom = P[IndexBottom].z;
    List_Edge[cnt].Normal_Bottom = P[IndexBottom].Normal;

    double DX = P[IndexBottom].x - P[IndexTop].x, DY =
P[IndexBottom].y - P[IndexTop].y, DZ = P[IndexBottom].z -
P[IndexTop].z;
    Vector D_Normal =
CObject_3D::SubtractVector(P[IndexBottom].Normal,
P[IndexTop].Normal);

    if ((DY == 0) && (DX == 0)) /* Cạnh đa giác có độ dài zero
sẽ không đưa vào danh sách cạnh để xử lý */

```

```

        continue;

    if (DY != 0)
    {
        List_Edge[cnt].Xstep = DX / DY;
        List_Edge[cnt].Zstep = DZ / DY;
        List_Edge[cnt].Normalstep =
CObject_3D::DivVector(D_Normal, DY);
    }
    else
    {
        List_Edge[cnt].Xstep = 0;
        List_Edge[cnt].Zstep = 0;
        List_Edge[cnt].Normalstep = CObject_3D::Vector3D(0, 0,
0);
        if (List_Edge[cnt].X_Intersection < P[IndexBottom].x) /*  

Lấy giao điểm phía bên phải */
        {
            List_Edge[cnt].X_Intersection = P[IndexBottom].x;
            List_Edge[cnt].Z_Intersection = P[IndexBottom].z;
            List_Edge[cnt].Normal_Intersection =
P[IndexBottom].Normal;
        }
    }

    cnt++;
}

int NumEdge = cnt;
ListIntersection = new IntersectionType2[NumEdge];

for (int Yscan = Y_Min; Yscan <= Y_Max; Yscan++)
{
    cnt = 0;
    for (i = 0; i<NumEdge; i++)
    {
        if ((List_Edge[i].Y_Top <= Yscan) && (Yscan <=
List_Edge[i].Y_Bottom))
        {
            ListIntersection[cnt].X =
int(List_Edge[i].X_Intersection + 0.5); /* chú ý làm tròn số thực  

về số nguyên */
        }
    }
}

```

```

        ListIntersection[cnt].Z = List_Edge[i].Z_Intersection;
/* giữ nguyên số thực, không làm tròn về nguyên */
        ListIntersection[cnt].Normal =
List_Edge[i].Normal_Intersection;
        ListIntersection[cnt].EdgeIndex = i;

        cnt++;
        List_Edge[i].X_Intersection += List_Edge[i].Xstep;
        List_Edge[i].Z_Intersection += List_Edge[i].Zstep;
        List_Edge[i].Normal_Intersection =
CObject_3D::AddVector(List_Edge[i].Normal_Intersection,
List_Edge[i].Normalstep);
    }
}
// Bỏ qua nếu Y_Scan nằm ngoài phạm vi [0, Width-1]
if ((Yscan < 0) || (Yscan >= Height))
    continue; // Tiếp tục vòng Lặp Yscan với dòng tiếp theo

// Sắp xếp các giao điểm
for (i = 0; i<cnt - 1; i++)
{
    long Min = ListIntersection[i].X, ChiSo = i;
    for (j = i + 1; j<cnt; j++)
        if (Min > ListIntersection[j].X)
    {
        Min = ListIntersection[j].X; ChiSo = j;
    }
    if (ChiSo != i)
    {
        IntersectionType2 tg = ListIntersection[i];
ListIntersection[i] = ListIntersection[ChiSo];
ListIntersection[ChiSo] = tg;
    }
}
// Xem xét với các giao điểm kép
int cnt2 = 0, k = 0;
while (k < cnt - 1)
{
    if (ListIntersection[k].X == ListIntersection[k + 1].X)
    {
        int Edge1 = ListIntersection[k].EdgeIndex, Edge2 =
ListIntersection[k + 1].EdgeIndex;
    }
}

```

```

        if ((List_Edge[Edge1].Y_Top != List_Edge[Edge2].Y_Top)
&& (List_Edge[Edge1].Y_Bottom != List_Edge[Edge2].Y_Bottom)) /*
không nằm về cùng một phía */
    k++; // bỏ qua giao điểm
}

if (cnt2 != k)
    ListIntersection[cnt2] = ListIntersection[k];

k++; cnt2++;
}

if ((k <= cnt - 1)) // còn điểm cuối cùng
{
    if (cnt2 != k)
        ListIntersection[cnt2] = ListIntersection[k];

    cnt2++;
}

if ((cnt2 % 2) != 0)
    cnt2--; // Bỏ bớt giao điểm sau cùng

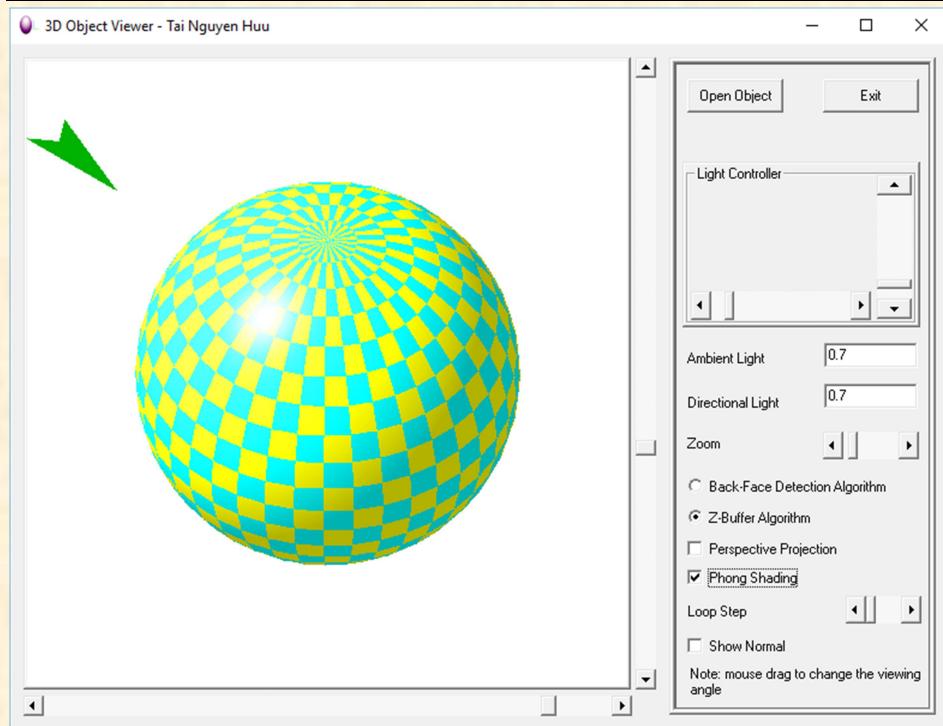
// Tô màu các đoạn
FillLines2(VLight, Ambient_Light, Directional_Light, Yscan,
ListIntersection, cnt2, pBits, BytePerLine, Hue, Saturation,
Lightness, Z_Buffer, Width, Height);
}

delete[] List_Edge;
delete[] ListIntersection;
}

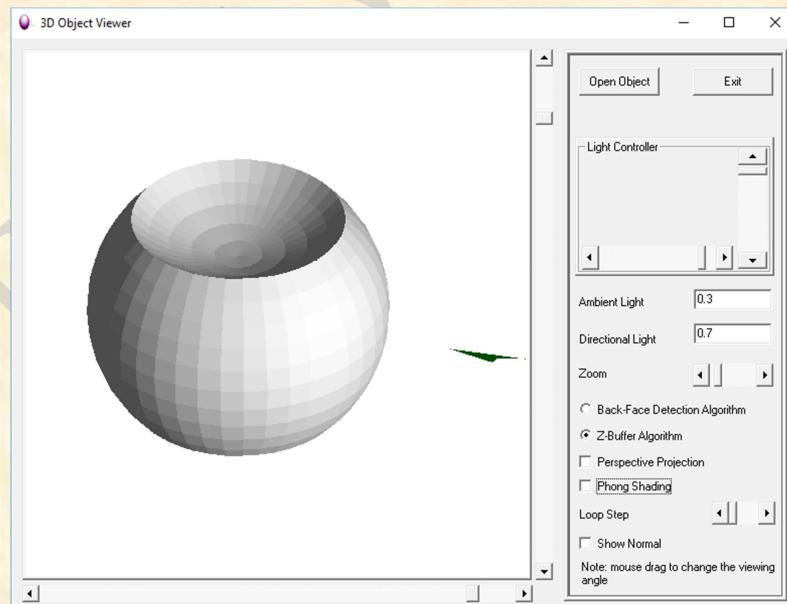
```

❖ Kết quả thực nghiệm và so sánh đánh giá:

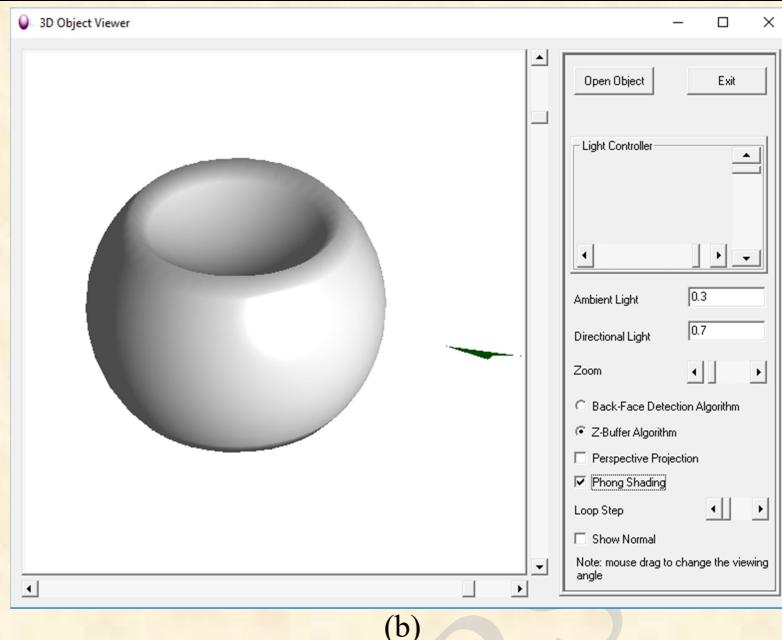
Chạy chương trình cài đặt trên và mở file đối tượng hình quả cầu màu vàng xanh như Hình 7.13 dưới đây, rồi so sánh với Hình 6.9 để thấy sự khác biệt do quá trình xử lý ánh sáng theo mô hình tô bóng Phong mang lại.



Hình 7.13. Kết quả thực nghiệm cài đặt với giải thuật z-buffer kết hợp phương pháp tô bóng Phong (so sánh với Hình 6.9 để thấy sự khác biệt)



(a)



(b)

Hình 7.14. Kết quả thực nghiệm so sánh giữa phương pháp tô bóng thường (a) và tô bóng Phong trên cùng một hình cầu lõm màu xám có 800 mặt đa giác

#### 5. 4. BÀI TẬP CUỐI CHƯƠNG

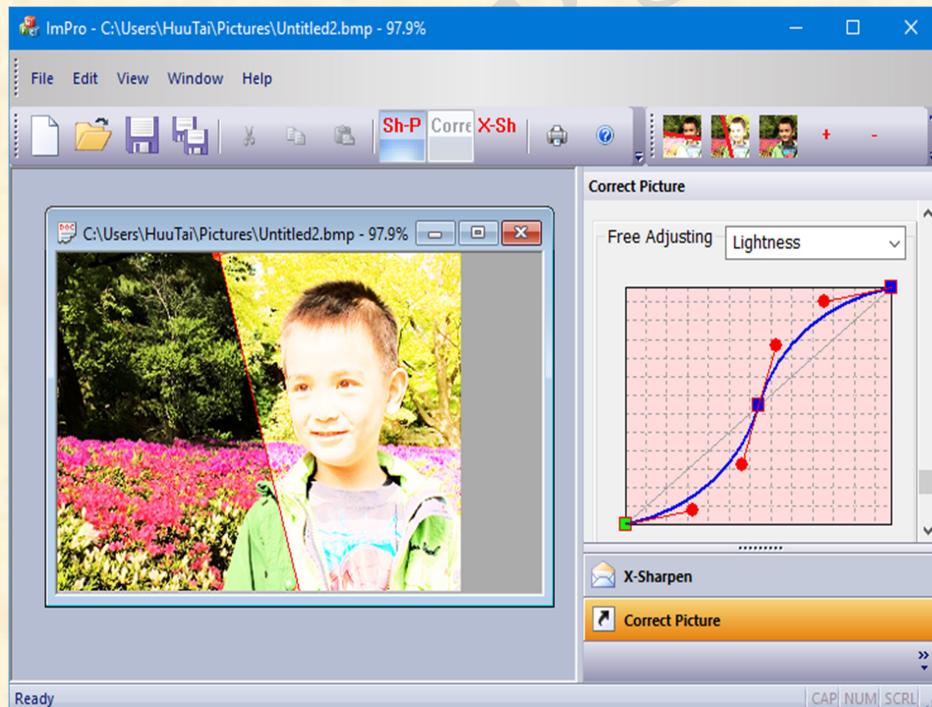
1. Xây dựng một chương trình cho phép quan sát vật thể 3D đặc lòi. Chương trình cho phép thay đổi vị trí quan sát, cho phép thể hiện tác động của các nguồn sáng xung quanh và định hướng lên đối tượng.
2. Nâng cao vấn đề 1, sao cho có thể thay đổi cường độ của các nguồn sáng, cũng như thay đổi hướng chiếu của nguồn sáng định hướng.
3. Hãy xây dựng chương trình với các chức năng như trong vấn đề 1, song sử dụng phương pháp tô bóng Phong.
4. Hãy tổng hợp các kiến thức đã biết để xây dựng một chương trình mô phỏng thế giới thực, trong đó có nhiều đối tượng khác nhau vận động.

## Chương 8

# CÁC PHƯƠNG PHÁP DỤNG ĐƯỜNG CONG VÀ MẶT CONG

Đường cong và mặt cong là một công cụ quan trọng trong nhiều lĩnh vực ứng dụng cũng như khoa học. Việc nghiên cứu và phát triển các phương pháp cho phép dễ dàng tạo nên các đường cong và mặt cong, cũng như kiểm soát chúng một cách hiệu quả sẽ mở ra nhiều cơ hội ứng dụng, nhất là trong lĩnh vực thiết kế xây dựng hay lĩnh vực Game 3 chiều và thực tại ảo.

Hình ảnh dưới đây minh họa việc áp dụng đường cong Bezier bậc 3 như một hàm ánh xạ để hiệu chỉnh chất lượng hình ảnh trong lĩnh vực xử lý ảnh (Digital Image Processing), *một cách hiệu quả và dễ dàng*.



*Hình 8.1. Thay đổi chất lượng hình ảnh với hàm ánh xạ có dạng đường cong Bezier bậc 3, cung cấp khả năng điều chỉnh hình dạng và độ cong của hàm ánh xạ một cách uyển chuyển và đơn giản*

Trong chương này chúng ta sẽ xem xét hai phương pháp tạo ra các đường cong và mặt cong tron (tức không gồ ghề hay gấp khúc tại điểm nối), đó là phương pháp Bezier và B-Spline. Ở đây, chúng ta chỉ bàn đến các phương pháp xây dựng đường và mặt cong dựa trên một số đặc điểm dữ liệu mô tả về đường cong đó, các dữ liệu mô tả của đường cong hay mặt cong chính là các điểm kiểm soát. Khi cần, chúng ta sẽ tiến hành nội suy lại đường cong hay mặt cong từ các điểm kiểm soát của nó theo một quy trình giải thuật nhất định.

Với một đường cong bất kỳ chúng ta có thể tiếp cận theo 2 cách:

- (1) Lấy một số điểm trên đường cong làm mẫu rồi tìm một hàm toán học phù hợp và tinh chỉnh hàm này sao cho nó đi qua các điểm mẫu và khớp với đường cong ban đầu. Khi đó, chúng ta có được công thức của đường cong và dùng nó để phát sinh lại đường cong mong muốn.
- (2) Cách thứ hai là dùng một tập các điểm kiểm soát cùng một giải thuật để xây dựng nên một đường cong của riêng nó dựa trên các điểm này. Có khả năng là đường cong được tạo ra không khớp với đường cong ban đầu, lúc này chúng ta sẽ di chuyển một vài điểm kiểm soát và khi đó giải thuật phát sinh một đường cong mới dựa trên các điểm kiểm soát mới thay đổi này. Tiến trình thay đổi các điểm kiểm soát được lặp đi lặp lại cho đến khi đường cong được tạo ra khớp với đường cong mong muốn (*hay đường cong mẫu*). Lúc này, chúng ta sẽ lưu trữ tập điểm kiểm soát này để phát sinh lại đường cong khi cần theo một giải thuật cố định.

Trong chương này sẽ trình bày các phương pháp xây dựng đường cong và mặt cong theo cách tiếp cận thứ hai. Qua cách tiếp cận này, người ta đã xây dựng nên 2 loại đường cong và mặt cong khác nhau, loại thứ nhất được gọi là các đường cong hoặc mặt cong Bezier, và loại thứ hai là các đường cong hoặc mặt cong B-Spline.

## **1. ĐƯỜNG CONG BEZIER VÀ MẶT CONG BEZIER**

Phản tiếp theo dưới đây chúng ta sẽ xem xét phương pháp để định nghĩa một đường cong tham số  $P(t)$  dựa trên một tập các điểm. Trước tiên, chúng ta xem xét ý tưởng của giải thuật Casteljau để định nghĩa các

đường cong loại này, sau đó sẽ đưa ra dạng toán học của đường cong Bezier qua công thức Bernstein.

### 1.1. Giải thuật De Casteljau

Để xây dựng đường cong  $P(t)$ , giải thuật này dựa trên một dãy các điểm kiểm soát cho trước rồi tạo ra các giá trị  $P(t)$  tương ứng với các giá trị  $t$  khác nhau. Khi chúng ta thay đổi các điểm kiểm soát thì sẽ kéo theo sự thay đổi dạng của đường cong. Phương pháp tạo đường cong ở đây dựa trên một dãy các thao tác nội suy tuyến tính (hay còn được gọi với một cái tên khác là nội suy khoảng giữa).

Ví dụ, với 3 điểm  $P_0, P_1, P_2$  chúng ta có thể xây dựng một đường cong Parabol nội suy từ 3 điểm này bằng cách: chọn một giá trị  $t$  nào đó nằm giữa 0 và 1 rồi chia đoạn  $P_0P_1$  theo tỷ lệ  $t$ , chúng ta được điểm chia cấp 1 là  $P_0^1(t)$  (hay điểm chia  $P_0^1$  phụ thuộc vào tham số  $t$ ). Tương tự, chúng ta chia tiếp  $P_1P_2$  theo cùng tỷ lệ  $t$  sẽ thu được  $P_1^1(t)$ . Với đoạn thẳng nối 2 điểm chia cấp 1 là  $P_0^1(t)$  và  $P_1^1(t)$ , chúng ta lại tiếp tục chia đoạn này theo cùng tỷ lệ  $t$ , kết quả sẽ thu được điểm chia cấp 2 là  $P_0^2(t)$ .

Với cách làm này, khi lấy  $t$  ở những giá trị khác nhau giữa 0 và 1 thì sẽ được tập điểm  $P_0^2(t)$ , và tập điểm này chính là đường cong  $P(t)$ .

Biểu diễn bằng phương trình:

$$P_0^1(t) = (1-t).P_0 + t.P_1$$

$$P_1^1(t) = (1-t).P_1 + t.P_2$$

$$P_0^2(t) = (1-t).P_0^1(t) + t.P_1^1(t)$$

Thay  $P_0^1(t)$  và  $P_1^1(t)$  bằng về phái của hai phương trình trên chúng ta thu được:

$$P(t) = P_0^2(t) = (1-t)^2.P_0 + 2t(1-t).P_1 + t^2.P_2$$

Đây là một đường cong bậc 2 theo  $t$ , với các tham số  $P_0, P_1, P_2$ . Do đó,  $P(t)$  là một Parabol.

Tổng quát chúng ta có giải thuật Casteljau cho  $(L+1)$  điểm:

Giả sử tập điểm là  $P_0, P_1, \dots, P_L$

Với mỗi giá trị  $t$  cho trước chúng ta tạo ra điểm  $P_i^r(t)$  ở thế hệ thứ  $r$  từ thế hệ thứ  $(r-1)$  trước đó theo phương trình:

$$P_i^r(t) = (1-t).P_i^{r-1}(t) + t.P_{i+1}^{r-1}(t)$$

Với  $r = 1, 2, \dots, L$ ;  $I = 0, 1, 2, \dots, L-r$

Thé hệ cuối cùng  $P_0^L(t)$  được gọi là đường cong Bezier, và các điểm  $P_0, P_1, \dots, P_L$  được gọi là các điểm kiểm soát hay đa giác Bezier.

### 1.2. Dạng Bernstein của các đường cong và mặt cong Bezier

Đường cong Bezier dựa trên  $(L+1)$  điểm kiểm soát  $P_0, P_1, \dots, P_L$  được cho bởi công thức:

$$P(t) = \sum_{k=0}^L P_k B_k^L(t)$$

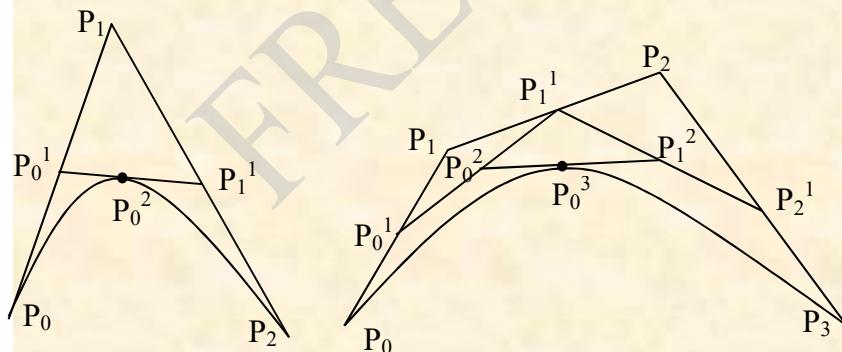
$P(t)$  ứng với một giá trị  $t$  nào đó nằm giữa 0 và 1 là một điểm trong 2D hay 3D.

$B_k^L(t)$  được gọi là đa thức BERNSTEIN và được cho bởi công thức:

$$B_k^L(t) = C_L^k \times (1-t)^{L-k} \times t^k = \frac{L!}{k!(L-k)!} \times (1-t)^{L-k} \times t^k, \text{ với } k \leq L$$

Mỗi đa thức Bernstein có bậc là  $L$ .

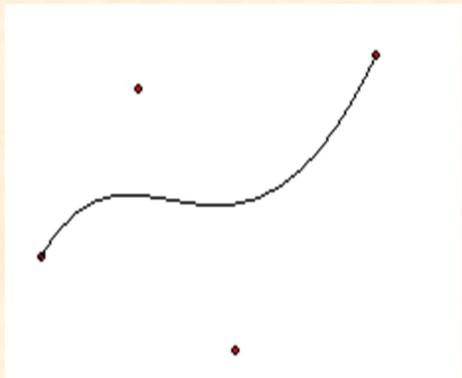
Thông thường chúng ta còn gọi các  $B_k^L(t)$  là các hàm trộn. Bởi chúng ta có thể hiểu vector  $P(t)$  như là một sự pha trộn của các vector kiểm soát  $P_0, P_1, \dots, P_L$  với một tỷ lệ nhất định theo công thức Bernstein ở các giá trị  $t$  khác nhau giữa 0 và 1.



Đường cong Bezier bậc 2

Đường cong Bezier bậc 3

Hình 8.2. Minh họa việc nội suy đường cong Bezier



Hình 8.3. Đường cong Bezier bậc 3 được vẽ bởi chương trình Paint của Microsoft. Các điểm tròn (màu đỏ) chính là các điểm kiểm soát của nó

- ❖ Đối với mặt BEZIER chúng ta có phương trình sau:

$$P(u, v) = \sum_{i=0}^M \sum_{k=0}^L P_{i,k} B_i^M(u) B_k^L(v)$$

Trong trường hợp này, khối đa diện kiểm soát mặt cong sẽ có  $(M+1) \times (L+1)$  đỉnh.

### 1.3. Dạng biểu diễn ma trận của đường Bezier

Để thích hợp cho các tính toán trên máy tính (việc tính toán trở nên hợp lý và nhanh chóng) người ta thường biểu diễn công thức dưới dạng ma trận:

$$B^L(t) = [B_0^L(t), B_1^L(t), B_2^L(t), \dots, B_L^L(t)]$$

$$P = [P_0, P_1, P_2, \dots, P_L]$$

$$P(t) = \sum_{k=0}^L P_k B_k^L(t) = \begin{bmatrix} B_0^L(t) & B_1^L(t) & B_2^L(t) & \cdots & B_L^L(t) \end{bmatrix} \times \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ \vdots \\ P_L \end{bmatrix} = B^L(t) \times P^T$$

Trong đó:  $P^T$  là dạng chuyển vị của  $P$ .

Mỗi  $B_k^L(t)$ , với  $k=0,1,\dots,L$  lại có thể xem là tích của  $(t^0, t^1, t^2, \dots, t^L)$  với các hệ số tương ứng của các  $t^i$   $i=0..L$ . Vector  $(t^0, t^1, t^2, \dots, t^L)$  được gọi là

cơ sở luỹ thừa (Power basic). Dưới dạng đa thức có thể biểu diễn  $B_k^L(t)$  như sau:

$$\begin{aligned} B_k^L(t) &= a_{0k}t^0 + a_{1k}t^1 + a_{2k}t^2 + \dots + a_{Lk}t^L \\ &= (t^0, t^1, t^2, \dots, t^L)(a_{0k}, a_{1k}, a_{2k}, \dots, a_{Lk})^T \end{aligned}$$

Do đó,  $P(t)$  có thể biểu diễn lại như sau:

$$P(t) = \text{Pow}^L(t) \cdot \text{Bez}^L \cdot P^T$$

Trong đó:

- $\text{Pow}^L(t) = (t^0, t^1, t^2, \dots, t^L)$
- $\text{Bez}^L$  là ma trận biểu diễn mảng  $B^L(t)$ , trong đó mỗi cột  $i$  của ma trận tương ứng với các hệ số  $(a_{0i}, a_{1i}, a_{2i}, \dots, a_{Li})$  của đa thức  $B_i^L(t)$ .

Ví dụ, ma trận  $\text{Bez}^3$  cho các đường Bezier bậc 3:

$$\text{Bez}^3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

Để tạo ra một đường cong Bezier từ một dãy các điểm kiểm soát, người ta áp dụng phương pháp Polyline gần đúng bằng cách lấy mẫu hàm  $P(t)$  ở các giá trị cách đều nhau của tham số  $t$ . Ví dụ như  $t_i = i/N$  với  $i = 0, 1, 2, \dots, N$ . Khi đó, chúng ta sẽ được các điểm  $P(t_i)$  từ công thức, nối các điểm này lại bằng các đoạn thẳng chúng ta sẽ được đường cong Bezier gần đúng. Để tính  $P(t_i)$ , có thể áp dụng dạng ma trận của  $P(t)$  như vừa trình bày ở trên, khi đó chỉ có thành phần  $\text{Pow}^L(t)$  là thay đổi, còn tích:

$$\text{Bez}^L \cdot P^T \quad \text{với } P = (P_0, P_1, \dots, P_L) \text{ là không đổi.}$$

#### 1.4. Các tính chất của đường cong Bezier

Đường cong Bezier có một số tính chất quan trọng sau:

##### 1.4.1. Nội suy được các điểm đầu và điểm cuối

Thật vậy:

Chúng ta có  $P(t) = \sum_{k=0}^L P_k B_k^L(t)$

Khi lấy giá trị t tiến đến 0 thì chúng ta có:

$$\begin{aligned} \lim_{t \rightarrow 0^+}(P(t)) &= \lim_{t \rightarrow 0^+}\left(\sum_{k=0}^L P_k B_k^L(t)\right) = \sum_{k=0}^L \left[\lim_{t \rightarrow 0^+}(P_k B_k^L(t))\right] \\ &= \sum_{k=0}^L \left[P_k \lim_{t \rightarrow 0^+}(B_k^L(t))\right] = \sum_{k=0}^L P_k \left[\lim_{t \rightarrow 0^+}(B_k^L(t))\right] \end{aligned}$$

Mà ta thấy:

$$\begin{aligned} \lim_{t \rightarrow 0^+}(B_k^L(t)) &= \lim_{t \rightarrow 0^+}(C_L^k (1-t)^{L-k} t^k) = C_L^k \times \lim_{t \rightarrow 0^+}((1-t)^{L-k}) \times \lim_{t \rightarrow 0^+}(t^k) \\ &= C_L^k \times 1 \times 0 = 0 \end{aligned}$$

với  $k \neq 0$ .

Khi  $k = 0$  thì:

$$\begin{aligned} \lim_{t \rightarrow 0^+}(B_0^L(t)) &= \lim_{t \rightarrow 0^+}(C_L^0 (1-t)^{L-0} t^0) = \lim_{t \rightarrow 0^+}\left(\frac{L!}{0! L!} (1-t)^{L-0} t^0\right) \\ &= \lim_{t \rightarrow 0^+}((1-t)^{L-0} t^0) = \lim_{t \rightarrow 0^+}((1-t)^L) \times \lim_{t \rightarrow 0^+}(t^0) = \lim_{t \rightarrow 0^+}((1-t)^L) \lim_{t \rightarrow 0^+}(1) \\ &= \lim_{t \rightarrow 0^+}((1-t)^L) \times 1 = \lim_{t \rightarrow 0^+}((1-t)^L) = 1 \end{aligned}$$

Vậy:  $\lim_{t \rightarrow 0^+}(P(t)) = P_0 \times 1 + P_1 \times 0 + P_2 \times 0 + \dots + P_L \times 0 = P_0$

Tương tự cho  $\lim_{t \rightarrow 1^-}(P(t))$  ta có  $\lim_{t \rightarrow 1^-}(P(t)) = P_L$

Vậy đường cong Bezier bắt đầu tại điểm  $P_0$  và kết thúc tại điểm  $P_L$ .

#### 1.4.2. Tính bát biến Affine

Khi biến đổi Affine một đường cong Bezier, chúng ta không cần biến đổi mọi điểm trong đường cong một cách riêng rẽ mà chỉ cần biến đổi Affine các điểm kiểm soát của đường cong đó, rồi sử dụng công thức Bernstein để tái tạo đường cong Bezier đã được biến đổi. Chúng ta gọi đây là tính bát biến Affine của các đường Bezier.

Chứng minh:

Giả sử ứng với một giá trị t cụ thể,  $P(t)$  là một điểm, và được biến đổi Affine thành  $P'(t)$  thì:

$$P'(t) = P(t) \cdot M + Tr = \sum_{k=0}^L P_k B_k^L(t) \cdot M + Tr$$

Trong đó:

M: Ma trận biến đổi.

Tr: Vector tịnh tiến.

$$\text{Xét đường cong } P(t) = \sum_{k=0}^L (P_k \cdot M + Tr) B_k^L(t) \quad (*)$$

được tạo ra bằng cách biến đổi Affine các vector  $P_k$ , ta sẽ chứng minh đường cong này chính là  $P'(t)$ .

Khai triển (\*) ta có:

$$\begin{aligned} & \sum_{k=0}^L P_k \cdot M \cdot B_k^L(t) + \sum_{k=0}^L Tr \cdot B_k^L(t) \\ &= \sum_{k=0}^L P_k \cdot M \cdot B_k^L(t) + Tr \sum_{k=0}^L B_k^L(t) = \sum_{k=0}^L P_k \cdot M \cdot B_k^L(t) + Tr \\ & \text{do } \sum_{k=0}^L B_k^L(t) = 1 \end{aligned}$$

*Chứng minh*  $\sum_{k=0}^L B_k^L(t) = 1$ :

Thật vậy, chúng ta đã biết khai triển nhị thức Newton  $(a+b)^L$ , ở đây áp dụng với  $a = (1-t)$  và  $b = t$ , lúc này  $\sum_{k=0}^L B_k^L(t) = [(1-t)+t]^L = 1^L = 1$ .

Vì vậy, ta có  $P'(t)$  nằm trên đường cong Bezier tạo ra bởi các điểm kiểm soát  $P'_k$ . Với  $P'_k$  là ảnh của  $P_k$  qua phép biến đổi Affine.

#### 1.4.3. Tính chất của bao lồi

Bao lồi của các điểm kiểm soát là tập đỉnh nhỏ nhất chứa tất cả các điểm kiểm soát đó. Đó cũng chính là tập tất cả các tổ hợp lồi của các điểm kiểm soát.

$$\sum_{k=0}^L \alpha_k P_k \text{ với } \alpha_k \geq 0 \text{ và } \sum_{k=0}^L \alpha_k = 1$$

Do  $P(t)$  là một tổ hợp lồi của các điểm kiểm soát  $\forall t$ , vì không có hệ số Bernstein nào âm và  $\sum_{k=0}^L B_k^L(t) = 1$ , nên mọi điểm trong đường cong Bezier sẽ nằm trong bao lồi của các điểm kiểm soát.

#### 1.4.4. Độ chính xác tuyến tính

Đường cong Bezier có thể trở thành một đường thẳng khi tất cả các điểm kiểm soát nằm trên một đường thẳng. Vì khi đó, bao lồi của chúng là một đường thẳng, nên đường Bezier do bị kẹp vào trong bao lồi cũng trở thành đường thẳng.

#### 1.4.5. Bất biến với những phép biến đổi Affine

Có thể đổi đường cong Bezier trong một khoảng tham số khác với khoảng  $(0,1)$ . Ví dụ như một khoảng  $(a,b)$  nào đó bởi một ánh xạ nào đó, *chẳng hạn: đường cong tham số  $t$  thành đường cong tham số  $u$* , không làm thay đổi đường cong ban đầu (dùng tính bất biến Affine).

#### 1.4.6. Đạo hàm của các đường Bezier

$$P'(t) = (P(t))' = L \sum_{k=0}^L \Delta P_k B_k^{L-1}(t), \quad \Delta P_k = P_{k+1} - P_k$$

là một đường cong Bezier khác được tạo ra từ các vector kiểm soát  $\Delta P_k$ .

### 1.5. Đánh giá đường cong Bezier và sự khác biệt của đường cong Spline

Bằng các điểm kiểm soát, chúng ta có thể tạo ra các dạng đường cong khác nhau bằng cách hiệu chỉnh các điểm kiểm soát cho tới khi tạo ra được một dạng đường cong mong muốn, công việc này được lặp đi lặp lại cho đến khi toàn thể đường cong thỏa mãn yêu cầu.

Tuy nhiên, có một vấn đề đối với đường cong Bezier là tính cục bộ yếu của nó, nghĩa là khi chúng ta thay đổi bất kỳ một điểm kiểm soát nào thì toàn bộ đường cong bị thay đổi theo. Nhưng trong thực tế, thông thường chúng ta mong muốn chỉ thay đổi một ít về dạng đường cong ở gần khu vực đang hiệu chỉnh các điểm kiểm soát.

Tính cục bộ yếu của đường cong Bezier có thể thấy được qua việc tất cả các đa thức  $B_k^L(t)$  đều khác 0 trên khoảng  $[0, 1]$ . Mặt khác, đường cong  $P(t)$  bùn thân nó lại là một tổ hợp tuyến tính của các điểm kiểm soát được gia trọng bởi các hàm  $B_k^L(t)$ , nên suy ra rằng mọi điểm kiểm soát đều có ảnh hưởng đến đường cong ở tất cả các giá trị  $t \in (0; 1)$ . Do đó, hiệu chỉnh bất kỳ điểm kiểm soát nào cũng đều ảnh hưởng đến dạng đường cong trên toàn thể.

Để giải quyết vấn đề này, người ta đã đi đến sử dụng một tập các hàm trộn khác nhau thay vì chỉ một hàm trộn  $B_k^L(t)$  như của đường cong Bezier, các hàm trộn này có giá mang chỉ là một phần của khoảng  $[0, 1]$ , hay nói cách khác là mỗi hàm trộn sẽ chỉ trộn với một số điểm kiểm soát để cho ra một phần của đoạn cong. Tập các đoạn cong do mỗi hàm trộn mang lại sẽ tạo nên một đường cong mà chúng ta mong muốn. Như vậy, hàm trộn chính là một tập các đa thức được định nghĩa trên những khoảng kề nhau, được nối lại với nhau để tạo nên một đường cong liên tục. Đường cong kết quả được gọi là đa thức riêng phần hay từng phần.

Ví dụ, chúng ta định nghĩa hàm  $g(t)$  gồm 3 đa thức  $a(t)$ ,  $b(t)$  và  $c(t)$  như sau:

$$a(t) = 1/2 t^2 \quad \text{có giá mang } [0, 1]$$

$$g(t) = b(t) = 3/4(t - 3/2)^2 \quad \text{có giá mang } [1, 2]$$

$$c(t) = 1/2(3 - t)^2 \quad \text{có giá mang } [2, 3]$$

Giá mang của  $g(t)$  là  $[0, 3]$

Các giá trị của  $t$  ứng với các điểm nối của các đoạn gọi là nút (knot).

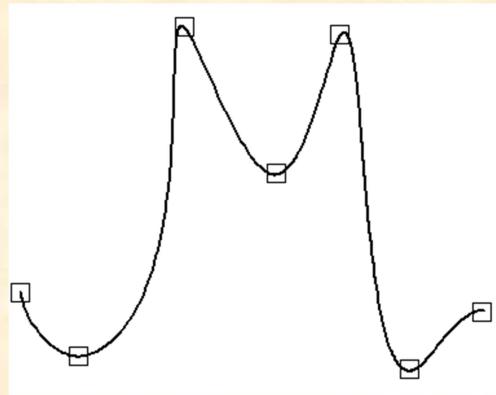
Hơn nữa, tại các điểm nối đường cong  $g(t)$  là trơn, không bị gấp khúc. Chúng ta gọi đó là hàm Spline.

Vậy một hàm Spline cấp  $m$  là đa thức riêng phần cấp  $m$  có đạo hàm cấp  $m-1$  liên tục ở mỗi nút. Dựa trên tính chất của hàm Spline, chúng ta có thể dùng nó như một hàm trộn để tạo ra đường cong  $P(t)$ , dựa trên các điểm kiểm soát  $P_0, P_1, \dots, P_L$ .

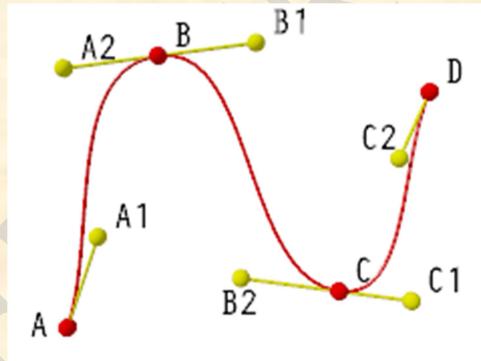
Khi đó:

$$P(t) = \sum_{k=0}^L P_k G_k(t)$$

## 2. ĐƯỜNG CONG SPLINE VÀ B-SPLINE



Hình 8.4. Một đường cong Spline được vẽ bởi chương trình AutoCad



Hình 8.5. Đường cong Multi-Spline với các điểm điều khiển (hay vector tiếp tuyến) giúp điều khiển độ cong

Một hàm Spline cấp m là đa thức riêng phần cấp m có đạo hàm cấp m-1 liên tục ở mỗi nút.

Dựa trên tính chất của hàm Spline, chúng ta có thể dùng nó như một hàm trộn để tạo ra đường cong  $P(t)$ , dựa trên các điểm kiểm soát  $P_0, P_1, \dots, P_L$ .

Khi đó:

$$P(t) = \sum_{k=0}^L P_k G_k(t)$$

với:

$P_k : k = 0, 1, \dots, L$  là các điểm kiểm soát.

$G_k(t) : k = 0, 1, \dots, L$  là các hàm trộn, liên tục trong mỗi đoạn con  $[t_i, t_{i+1}]$  và ở mỗi nút. Mỗi  $R_k(t)$  là một đa thức riêng phần (piecewise polynomial). Các đoạn đường cong riêng phần này gặp nhau ở mỗi nút và tạo cho đường cong trở nên liên tục. Chúng ta gọi những đường cong như vậy là Spline. Cho trước một vector nút, thì có nhiều họ hàm trộn có thể được dùng để tạo ra các đường cong Spline có thể định nghĩa trên vector nút đó. Mỗi họ như vậy được gọi là cơ sở cho các Spline. Trong số các họ hàm này, có một cơ sở cụ thể mà các hàm trộn của nó có giá mang nhỏ nhất mà nhờ vậy nó đem lại khả năng kiểm soát cục bộ lớn nhất. Đó là các B-Spline (B là viết tắt của Basic).

Đối với các hàm B-Spline, mỗi đa thức riêng phần tạo nên nó có một cấp nào đó, người ta gọi là  $m$ , do đó thay vì dùng ký hiệu  $R_k(t)$  cho các hàm riêng phần này bởi  $N_{k,m}(t)$ .

Do đó, đường cong B-Spline có thể biểu diễn bởi công thức sau:

$$P(t) = \sum_{k=0}^L P_k N_{k,m}(t)$$

Trong các hàm B-Spline cấp  $m$  thì hàm B-Spline cấp 2 và cấp 3 là quang trọng nhất, nó được dùng trong hầu hết các hệ thống xử lý đồ họa.

Tóm lại: Để xây dựng các đường cong B-Spline chúng ta cần có:

- Một vector nút  $T = (t_0, t_1, \dots)$ .
- $(L+1)$  điểm kiểm soát  $P_k$ .
- Cấp  $m$  của các hàm B-Spline và công thức cơ bản cho hàm B-Spline  $N_{k,m}(t)$ .

$$N_{k,m}(t) = N_{k, m-1}(t) + N_{k+1, m-1}(t)$$

Đây là một công thức đệ quy với:

$$N_{k,l}(t) = \begin{cases} 1, & \text{nếu } t_k < t_{k+1} \\ 0, & \text{ngược lại} \end{cases}$$

(Hàm hằng 1 trên đoạn  $[t_k, t_{k+1}]$ )

Đối với các mặt B-Spline, chúng ta có công thức biểu diễn tương tự:

$$P(u, v) = \sum_{i=0}^m \sum_{k=0}^L P_{i,k} N_{i,m}(u) N_{k,m}(v)$$

(tương tự như mặt Bezier).

**Ghi chú:** Các đường Bezier là các đường B-Spline.

Sau đây là một công thức cho đường cong Spline bậc 3 có tên là Hermite Spline với công thức sau:

$$P(u) = [u^3 \quad u^2 \quad u \quad 1] \times M_c \times \begin{bmatrix} P_{k-1} \\ P_k \\ P_{k+1} \\ P_{k+2} \end{bmatrix}$$

với  $M_c$  được gọi là ma trận cốt yếu:

$$M_c = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \text{ với } s = (1-t)/2.$$

Hay ở dạng tường minh sẽ là:

$$\begin{aligned} P(u) &= P_{k-1}[-su^3 + 2su^2 - su] + P_k [(2-s)u^3 + (s-3)u^2 + 1] + \\ &\quad P_{k+1} [(s-2)u^3 + (3-2s)u^2 + su] + P_{k+2} [su^3 - su^2] \\ &= P_{k-1} CAR_0(u) + P_k CAR_1(u) + P_{k+1} CAR_2(u) + P_{k+2} CAR_3(u) \end{aligned}$$

và chúng ta gọi  $CAR_k(u)$  với  $k = 0, 1, 2, 3$  là các hàm trộn.

## TÀI LIỆU THAM KHẢO

### Bắt buộc

1. John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley (2014), *Computer Graphics: Principles and Practice*, third edition, Addison-Wesley Publishing Company, USA.

### Không bắt buộc

2. Francis S. Hill (1990), *Computer Graphics*, Jr. Macmillan Publishing Company, New York.
3. John R. Rankin (1989), *Computer Graphics Software Construction*, Prentice Hall of Australia.
4. Harrington (1986), *Computer Graphics: A Programming Approach*, McGraw-Hill Book Company, Singapore.
5. Video Electronics Standards Association (1998), *VESA BIOS Extensions Core Functions Standard – Version 3.0*.
6. [http://www.tftcentral.co.uk/articles/content/pointers\\_gamut.htm](http://www.tftcentral.co.uk/articles/content/pointers_gamut.htm) , 20/06/2016.
7. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd183567\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd183567(v=vs.85).aspx), 20/06/2016.
8. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd145049\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd145049(v=vs.85).aspx), 20/06/2016.
9. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd183494\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd183494(v=vs.85).aspx), 20/06/2016.
10. <https://msdn.microsoft.com/en-us/library/windows/desktop/dd162467>, 20/06/2016.