2-Dimensional Shape Categorization Using Polar Coordinate Representations

by Daniel Russo, Boston College, Class of 2002

Submitted for the completion of MC397 (Honors Thesis)

under the supervision of Associate Professor Peter Kugel

Introduction

In my experience with computer vision, the one thing that struck me most of all was the seemingly exclusive use of rectangular (x, y) coordinates for shape representations in template matching. It occurred to me that replacing such traditional representations with polar coordinates – a system I had seen used all-too-little since it was introduced to me in high school – would offer a number of advantages, especially when shapes need to be rotated to be properly compared. For my thesis, I decided to implement this idea, first in a one-on-one comparison situation, then for grouping like shapes within a scene. In doing so, I was able to come up with some surprisingly efficient algorithms, yielding promising results.

Polar Coordinates

The use of polar coordinates in this project was inspired by the fact that rectangular coordinates are not very convenient for comparisons involving rotation. When rectangular coordinates x and y are transformed through angle ϕ about the origin, the new coordinates x' and y' are determined by the following equations:

$$x' = x \cos\phi - y\sin\phi$$

$$y' = x\sin\phi + y\cos\phi$$

For each increment of the rotation, the new coordinates need to be calculated. This process can be simplified somewhat by saving values of the sine and cosine functions to a table, though doing so limits the possible angle increments to those listed in the table.

Changing over to a polar coordinate system, x and y are replaced with radius r and angle θ, which are calculated as follows:

$r = \text{sqrt}(x^2 + y^2)$

$\theta = \tan^{-1}(y/x)$ if x != 0, π/2 if x = 0 and y >0, -π/2 if x = 0 and y >0, 0 if x = y = 0[*]

These values only need to be calculated once, when the shape is first read from the file. After that, the rotational transformations used in comparisons only involve adding the transformation angle, ϕ, to the value of θ, producing a new value θ' by the following equation:

$\theta' = \theta + \phi$

As the number of comparisons increases, the running time of the polar coordinate method becomes increasingly attractive. Polar coordinates can also be simplified in a way that rectangular coordinates cannot, as I will later demonstrate.

Stage 1: Comparing Two Single-Shape BMP Files

The first program I wrote for this project was the simplest possible incarnation of the idea. Two separate BMP files, each containing a single shape, are compared to each other, producing a floating-point result between 0 and 1, with 1 representing a perfect match and 0 representing a perfect mismatch. Whether this comparison is done with or without accounting for rotation is left for the user to decide.

The program starts by identifying the shapes in each image via a recursive fill algorithm. This algorithm creates a list of points, and the center of these points is calculated by averaging their x and y values. The points are then translated so that the center becomes the origin. Finally, the points are converted into their polar equivalents using the equations described in the previous section. Shapes are stored as a doubly-linked list of polar coordinates, sorted by their θ values, which range from -π to π. This list, along with important attributes such as the shape's color and the average value of r for the points within the shape, is stored in a class known as Shape2D.

The points in the two shapes are compared based on certain specified error values of r and θ. The error in r specifies the maximum number of pixel widths that two points can be separated by in their radius value while still being recorded as matching each other. The theta error, on the other hand, does not represent the maximum allowable difference in θ on its own, but is instead a multiplier whose value is divided by 2πr* to obtain the allowable interval. Both error values were set to 1.0 throughout most of the testing at this stage, as this was found to be the value that produced the most accurate results. Also, in the case of shapes of unequal size, the r values of the smaller shape were multiplied by the ratio of the larger shape's average r value to its own average r value, and the errors were used with these new r values in mind. (The r error, for example, was multiplied by the same ratio)

When shapes A and B are compared based on their orientations, the matching function first tries to find a match for the first point in A by searching through the list of points in B until it either finds a match (a point where r and θ are both within the allowable margin of error), or it leaves the allowable θ interval. This process is repeated

---

* Actually, in this last case the value of θ is undeterminable. Setting it equal to 0 was just a matter of personal preference.

* This value of r is actually an average of the r values from the two shapes being compared.

until all of the points in A have been checked. A result is produced by dividing the number of points in A that were successfully matched by the total number of points in A. The process is then repeated on shape B, producing a second result. The two results are then multiplied together to produce the sameness value that is retuned by the program. The reason why the process is repeated is to prevent cases where one shape is small subset of another shape from being recorded as a perfect match, as well as to distinguish between a small shape within a large shape and two large shapes with a small overlap. For both stages of this project, I used 0.9 as my sameness threshold, with all shapes above the threshold being considered a match and all those below being considered a mismatch.

When comparing two shapes regardless of their orientation, the same basic process is applied, but is repeated with an increasing offset being added to the value of θ for shape A. This value is corrected for higher values of the offset so that the θ value always stays within the interval $(-\pi, \pi)$. Once the best match from these trials is found, its θ offset is used to make the comparison to points in B. This entire process has a running time of $O(n^3)$.

Standardizing Shapes

In order to facilitate the shape comparison process, I decided to institute a process I call "standardizing", which involves simplifying the shape representations in several ways.

One of these simplifications consisted of normalizing the radius values of all the points in each shape, so that they all fell between 0 (the center of the object) and 1 (the farthest point from the center). Previously, I had been using a multiplier based on the average value of r in both shapes to compare objects with different sizes.

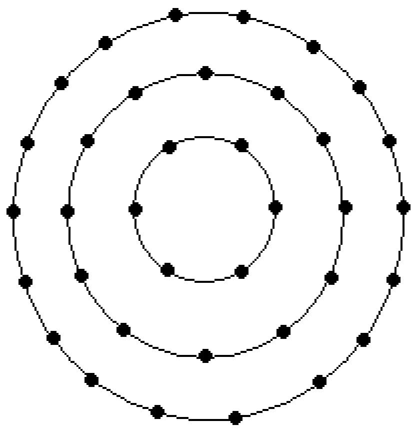
A second improvement involved discarding the existing point list in favor of one in which the points are stored at regular intervals of r and θ. In a standardized shape (represented by the structure StdShape2D in the program), points are stored in evenly-spaced "rings", each of which has a set radius. Each of the points along a ring can occur at any one of (int)$2\pi r$ evenly-spaced positions. Once the number of rings, x, has been determined, the program decides whether to put a point at each position of the standardized shape by checking whether there is a point in the original Shape2D representation that falls within the error ranges for r and θ, which are determined by the function as follows:

r error = $1/x$

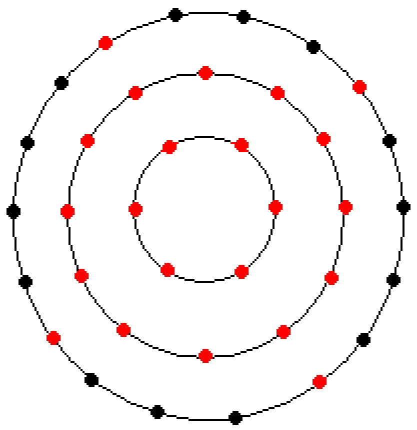
θ error = 2π/(int)$2\pi r$

Note that the θ error cannot be simplified because its denominator must be simplified to an integer before the division can take place.

For example, the possible positions for points in the first three rings are shown in the following diagram:



In the next diagram, the dots highlighted in red are those that might be chosen by the program to represent a square:



Finally, the third revision I implemented here allows the user to specify the maximum number of points that can be used to represent an object. If a Shape2D instance has fewer than the maximum number of points, the standardized shape is simply calculated by using the maximum value of r in the original representation as the value of x (after converting it to an integer, of course). If a shape exceeds the maximum number of points, the value of x is determined by finding the largest value of x that would have a total number of possible point positions that is less than or equal to the specified maximum number of points. This step prevents reduces delays caused by comparing shapes with high point totals, since point totals directly determine the running time of the algorithm.

Stage 2: Grouping Shapes from a Scene in a BMP File

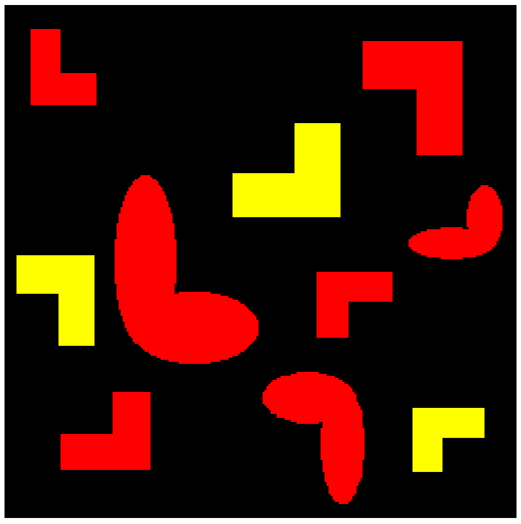
During the second stage of the project, which I will be referring to in the examples that follow, the focus was shifted from comparing just two individual shapes to comparing many shapes in a single image, in an effort to group together shapes that bear a strong resemblance to each other in both shape and color. These groupings, called "concepts", are stored in order of decreasing popularity, so that the program can recognize common objects faster than rare ones.

The use of standardized shapes makes the comparisons for this process significantly faster than they were in the first stage. When comparing two shapes at a given orientation, the program now only needs to visit each point on each shape once, following the points around each of the rings in order. The number of different rotational orientations that need to be tested is (int)2πx - the maximum number of points on the outer ring of a shape, taken from whichever of the two shapes has the greater number of rings. The error values for r and θ are the same as those used in creating standardized shapes, taken from the shape with fewer points, since the error there is greater. The running time for this algorithm is O((n1+n2)*sqrt(n2)), where n1 is the number of points in the smaller shape and n2 is the number of points in the larger one. The points in n1 and n2 are each checked once in the two inner loops, and the number of rotational positions checked by the outer loop is on the order of sqrt(n2), since it is based on the number of the points in the outermost ring, and both the number of rings and the number of points in each ring are based on x, making n2 the sum of the points in all the rings, of the order x^2.

Experiments

In order to test the versatility of the algorithm, I created a series of images designed to illustrate a variety of sorting abilities.

The first of these images contains a general assortment of objects, some of which are rounded "L" shapes, and the rest of which are squared-off "L" shapes, occurring in two distinctly different colors:

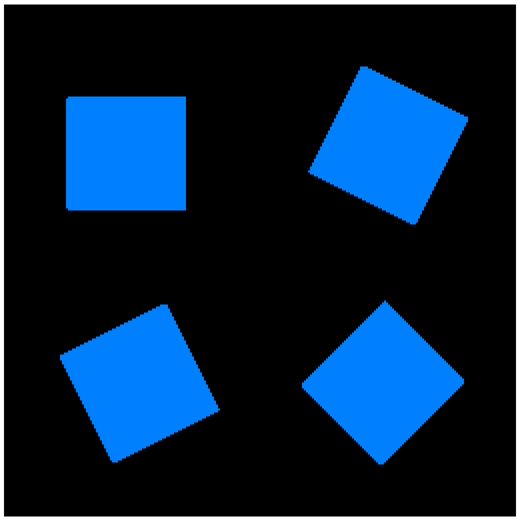


When given the image, the algorithm produced the following output, displaying the centers of shapes in each concept as rectangular coordinates, with (0, 0) representing the upper left corner of the bitmap and (255, 255) representing the lower right corner:

```
Concepts found:
Color: Red=255, Green=0, Blue=0
Center at (25.5385,34.1923)
Center at (54.2903,216.065)
Center at (170.808,145.538)
Center at (208.73,40.5421)
Instances: 4
Color: Red=255, Green=0, Blue=0
Center at (82.2759,141.342)
Center at (159.845, 208.786)
Center at (229.99,112.24)
Instances: 3
Color: Red=255, Green=255, Blue=0
Center at (29.0645,142.71)
Center at (145.555,86.8355)
Center at (218.132,213.132)
Instances: 3
```

The first concept shown here contains the red squared-off “L” shapes, the second represents the rounded “L” shapes, and the third represents the yellow squared-off “L” shapes.

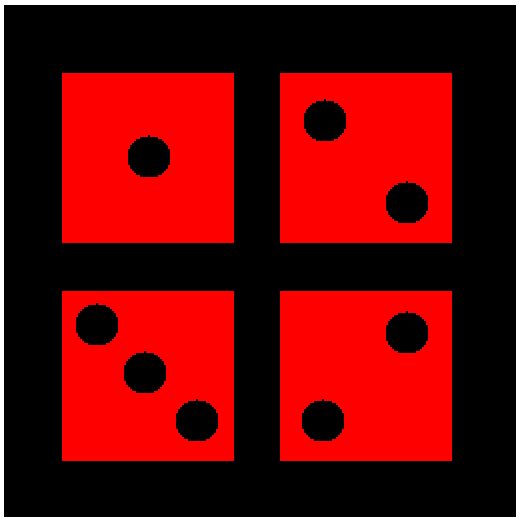
The second image was used to test the algorithm’s ability to detect arbitrary rotations of the same shape, in this case a square:



The program produced the following output:

```
Concepts found:
Color: Red=0, Green=128, Blue=255
Center at (67.2016,188.903)
Center at (60.5173,74)
Center at (189,188.988)
Center at (191.798,69.9029)
Instances: 4
```

In the third image, holes in the shapes are introduced, demonstrating why it was important for the algorithm to use all the points in the shape for its comparisons, instead of just those on the outer edge:

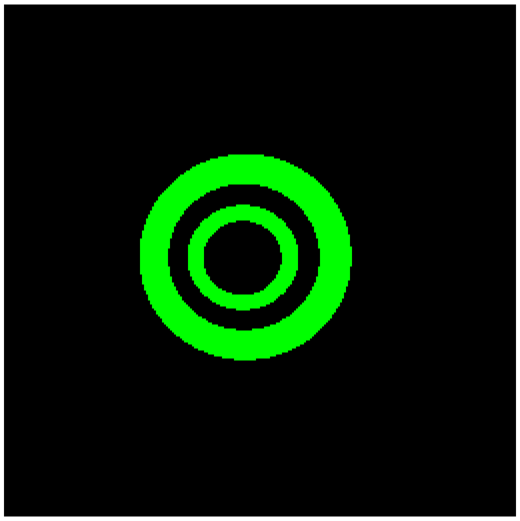


The algorithm grouped the shapes as follows:

```
Concepts found:
Color: Red=255, Green==0, Blue=0
Center at (71.4746,76.0269)
Instances: 1
Concepts found:
Color: Red=255, Green==0, Blue=0
Center at (180.5,75.7894)
Center at (180.553,184.95)
Instances: 2
Concepts found:
Color: Red=255, Green==0, Blue=0
Center at (71.6412,185.259)
Instances: 1
```

The first concept here contains the square with one hole, the second contains the squares with two holes, and the third contains the square with three holes.

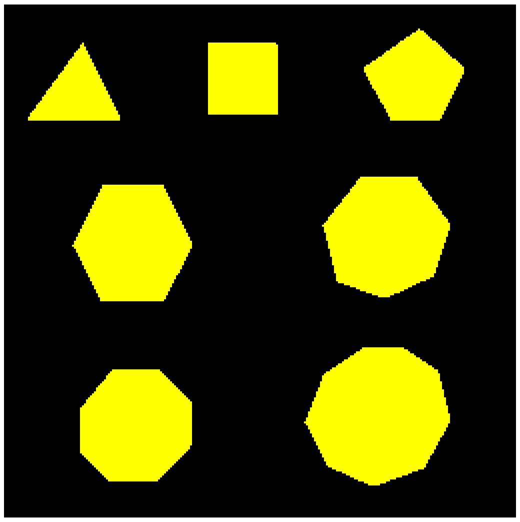
A fourth image shows how the program's ability to deal with holes allows it to recognize similar concentric shapes:



The algorithm recognized both shapes as representing a single concept, though the centers of the two shapes are difficult to distinguish from each other:

```
Concepts found:
Color: Red=0, Green=255, Blue=0
Center at (121.04,125.8)
Center at (119.218,125.771)
Instances: 2
```

The fifth and final image tests one of the algorithm's limitations as it depicts shapes with increasing numbers of sides:



The following output was produced:

```
Concepts found:
Color: Red=255, Green=255, Blue=0
Center at (35.9425,44.4441)
Instances: 1
Color: Red=255, Green=255, Blue=0
Center at (118.986,36.5139)
Instances: 1
Color: Red=255, Green=255, Blue=0
Center at (118.986,36.5139)
Instances: 1
Color: Red=255, Green=255, Blue=0
Center at (63981,118.76)
Instances: 1
Color: Red=255, Green=255, Blue=0
Center at (190.577,114.799)
Center at (65.3313,209.754)
Center at (187.02,204.288)
Instances: 3
```

The first four concepts represent shapes with increasing numbers of sides, from the triangle to the hexagon. The fifth concept contains the heptagon, octagon, and nonagon, which the algorithm could not distinguish from each other. In a way, this is similar to the human eye, which often cannot distinguish between shapes with more than six sides without counting the sides. (Recognition of the octagon, such as in a stop sign, is a special case, since such recognition is based partially on the shape's orientation)

Future Work

As I continue to develop this algorithm, I hope to increase its versatility in a number of ways.

The first of these would be to give it the ability to process multiple files while keeping track of the concepts it recognized from previous files. A master list of concepts could then be stored, allowing the algorithm to learn in a more permanent manner.

I would also like to add to the program a smoothing algorithm, so that I can use actual photographs instead of having to manufacture my own images. This would go a long way towards moving this project from the realm of the theoretical to that of the practical.

On a less cosmetic level, I would most like to expand on the method by which concepts are stored. If the program were given a series of images that represent the same scene moving over a period of time, it should be able to recognize objects as they move through three dimensions, possibly being distorted or partially obscured. This could be done by creating multiple instances by which a concept could be identified, since a shape will not change much between two consecutive frames of motion, but it can change considerably over long periods of time. Keeping the most recent objects used in a convenient location would also be useful for this task.

I hope to make these and many other additions to the program over time. I see in it a potential that could eventually lead to a wide variety of applications for computer vision and human-computer interaction. However, there is still much work left to be done before specific applications can be considered.

The following pages contain the final version of the source code used in the program.

```
/*    Bitmap.h

      Contains definition of Bitmap class, used for storing BMP
      file information.
*/

#ifndef _BITMAPH_
#define _BITMAPH_

class Bitmap {
      protected:
            int x, y;   //width and height of bitmap, respectively
            char *header;     //header information from BMP file
            unsigned char ***color; //color values of all coordinates
in bitmap

      public:
            Bitmap(const char *filename);
            Bitmap(int new_x,int new_y,unsigned char ***new_color);
            long getX(void) { return x; }
            long getY(void) { return y; }
            unsigned char *** getColor(void) { return color; }
            void saveToFile(char *filename);    //not yet created
};

#endif
```

```
/*    Bitmap.cpp

      Function definitions for Bitmap class.
*/

#include <stdio.h>
#include <iostream.h>
#include "bitmap.h"
#include "misc.h"

/*    Bitmap::Bitmap(const char *filename)

      Constructor for Bitmap class; reads file specified
      by user.

      Parameters:

        filename - name of file to be read
*/

Bitmap::Bitmap(const char *filename)
{
      FILE *bmpfile = fopen(filename,"r");
      int i, j;

      header = new char[54];

      for (i=0;i<54;i++)
            fscanf(bmpfile,"%c",header+i);

      x = BMP_WIDTH;
      y = BMP_HEIGHT;

      color = new unsigned char **[x];

      for (i=0;i<x;i++) {
            color[i] = new unsigned char *[y];
            for (j=0;j<y;j++) {
                  color[i][j] = new unsigned char[3];
            }
      }

      for (j=y-1;j>=0;j--) {
            for (i=0;i<=x-1;i++) {
                  fscanf(bmpfile,"%c",&(color[i][j][B]));
                  fscanf(bmpfile,"%c",&(color[i][j][G]));
                  fscanf(bmpfile,"%c",&(color[i][j][R]));
            }
      }

      fclose(bmpfile);
}
```

```
/*    Scene2D.h

      Contains definition of Scene 2D class, a data representation
      of the scene being analyzed by the program.
*/

#ifndef _SCENE2DH_
#define _SCENE2DH_

#include "Bitmap.h"
#include "View2D.h"

class Scene2D {
      protected:
            int max_x, max_y; //width and height of image, respectively
            Bitmap *source;   //Bitmap object used to create Scene2D
                              //instance
            unsigned char ***color; //Color values of all coordinates
                                    //in scene

      public:
            Scene2D(char *bitmap);
            View2D * makeView2DPtr(int x1,int y1,int x,int y,int res);
            View2D * makeView2DPtr(int res) { return
makeView2DPtr(0,0,max_x,max_y,res); }
                  //view encompasses whole scene by default
            View2D * makeView2DPtr(void) { return
makeView2DPtr(0,0,max_x,max_y,1); }
                  //blur is set to 1 by default
};


#endif
```

```
/*    Scene2D.cpp

      Function definitions for the Scene2D class.
*/


#include <stdio.h>
#include <iostream.h>
#include "Scene2D.h"
#include "Bitmap.h"
#include "misc.h"

/*    Scene2D::Scene2D(char *bitmap)

      Constructor for Scene2D class; takes a filename,
      uses it to create a Bitmap object, then reads from
      the Bitmap object.

      Parameters:

        bitmap - filename of image to be analyzed
*/

Scene2D::Scene2D(char *bitmap)
{
      source = new Bitmap(bitmap);

      max_x = source->getX();
      max_y = source->getY();
      color = source->getColor();
}

/*    View2D * Scene2D::makeView2DPtr(int x1,int y1,int x,int y,int
res)

      Create a View2D pointer (see View2D.h) from the current scene.

      Parameters:

        x1, y1 - starting coordinates of new view
        x, y - width and height of new view
        res - number of vertical and horizontal pixels in the scene
        that will be combined into one pixel in the new view. (Note: In
        the current version of the program, this value is always set to
        1)

      Return Value:

        A View2D pointer, representing a (possibly scaled down) portion
        of the current scene.
*/
View2D * Scene2D::makeView2DPtr(int x1,int y1,int x,int y,int res)
{
      unsigned char ***new_color;
      int i, j, k, l;
      int color_sum[3];
```

```
        if (x1<0) {
              x -= (int)((0 - x1)/res);
              x1 = 0;
        }
        if (y1<0) {
              y -= (int)((0 - y1)/res);
              y1 = 0;
        }
        if (x1+x*res>max_x) {
              x = (int)(max_x/res);
        }
        if (y1+y*res>max_y) {
              y = (int)(max_y/res);
        }

        new_color = new unsigned char **[x];

        for (i=0;i<x;i++) {
              new_color[i] = new unsigned char *[y];
              for (j=0;j<y;j++) {
                    new_color[i][j] = new unsigned char[3];
              }
        }

        for (i=x1;i<x*res;i+=res) {
              for (j=y1;j<y*res;j+=res) {
                    color_sum[R] = 0;
                    color_sum[G] = 0;
                    color_sum[B] = 0;
                    for (k=i;k<i+res;k++) {
                          for (l=j;l<j+res;l++) {
                                color_sum[R] += color[k][l][R];
                                color_sum[G] += color[k][l][G];
                                color_sum[B] += color[k][l][B];
                          }
                    }
                    new_color[i][j][R] = color_sum[R]/(res*res);
                    new_color[i][j][G] = color_sum[G]/(res*res);
                    new_color[i][j][B] = color_sum[B]/(res*res);
              }
        }

        return new View2D(new_color,x1,y1,x,y,res);
  }
```

```
/*    View2D.h

      Definition of the View2D class, representing a "view", which is a
      (possibly zoomed-in) portion of a Scene2D object.  This class
      can also create a list of shapes contained within the boundaries
      of the view.
*/

#ifndef _VIEW2DH_
#define _VIEW2DH_

#include "Shape2D.h"
#include "misc.h"

class View2D
{
      protected:
            int x1, y1; //top left corner of the view
            int x, y;   //width and height of the view, respectively
            int res;    //zoom level of the view
            unsigned char ***color; //colors of coordinates within the
                                    //view
            bool **mark;      //coordinates of the view that have been
                              //checked for shapes
            void makePoint2DList(int curr_x,int curr_y,Point2DList
*pl_address);

      public:
            View2D(unsigned char ***new_color,int new_x1,int new_y1,int
new_x,int new_y,int new_res);
            Shape2DList makeShape2DList(void);
};

#endif
```

```
/*    View2D.cpp

      Function definitions for the View2D class.
*/

#include <iostream.h>
#include "Scene2D.h"
#include "misc.h"

/*    View2D::View2D(unsigned char ***new_color,int new_x1,int
new_y1,int new_x,int new_y,int new_res)

      Constructor for View2D; sets member variables and initializes
      mark list.

      Parameters:

        new_color - the color information for the view
        new_x1,new_y1 - the top-left corner of the view
        new_x,new_y - the width and height of the view, respectively
        new_res - the zoom level of the view
*/

View2D::View2D(unsigned char ***new_color,int new_x1,int new_y1,int
new_x,int new_y,int new_res)
{
      int i, j;

      color = new_color;
      x1 = new_x1;
      y1 = new_y1;
      x = new_x;
      y = new_y;
      res = new_res;

      mark = new bool *[x];

      for (i=0;i<x;i++) {
            mark[i] = new bool[y];
            for (j=0;j<y;j++)
                  mark[i][j] = false;
      }
}

/*    Shape2DList View2D::makeShape2DList(void)

      Create a Shape2DList object by identifying and listing shapes
      in the view.

      Retrun Value:

        The newly-created shape list
*/

Shape2DList View2D::makeShape2DList(void)
{
      int i, j;
```

```
	Shape2DList sl = NULL, new_sl, sl_counter;
	Point2DList pl = NULL, pl_killer;

	for (i=0;i<x;i++) {		//shape-finding loop
		for (j=0;j<y;j++) {
			if ((!mark[i][j]) && !isBlack(color[i][j])) {
	//Black pixels are assumed to be background
				makePoint2DList(i,j,&pl);
				new_sl = new Shape2DListData;
				new_sl->s = new Shape2D(pl);
				if (sl==NULL) {
					sl = new_sl;
					sl->next = NULL;
				}
				else if (new_sl->s->getColorValue() >
sl->s->getColorValue()) {
					new_sl->next = sl;
					sl = new_sl;
				}
				else if (sl->next==NULL) {
					sl->next = new_sl;
					new_sl->next = NULL;
				}
				else {
					sl_counter = sl;
					while (sl_counter->next!=NULL) {
						if (new_sl->s->getColorValue() >
sl_counter->next->s->getColorValue()) {
							break;
						}
						sl_counter = sl_counter->next;
					}
					new_sl->next = sl_counter->next;
					sl_counter->next = new_sl;
				}

			}
			while (pl!=NULL) {
				pl_killer = pl;
				pl = pl->next;
				delete pl_killer;
			}
		}
	}

	return sl;
}

/*	void View2D::makePoint2DList(int curr_x,int curr_y,Point2DList
*pl_address)

	Creates a Point2DList of rectangular points in a shape, using
	a recursive fill agorithm.

	Parameters:

	  curr_x,curr_y - x and y view coordinates of the current point
```

```
        pl_address - a pointer to the point list being created
*/

void View2D::makePoint2DList(int curr_x,int curr_y,Point2DList
*pl_address)
{
      Point2DList new_pl;

      mark[curr_x][curr_y] = true;

      new_pl = new Point2DListData;
      new_pl->color = new unsigned char[3];
      new_pl->color[R] = color[curr_x][curr_y][R];
      new_pl->color[G] = color[curr_x][curr_y][G];
      new_pl->color[B] = color[curr_x][curr_y][B];
      new_pl->x = curr_x;
      new_pl->y = curr_y;
      new_pl->next = *pl_address;
      *pl_address = new_pl;

      if (curr_x>0) {
            if(isSameColor(color[curr_x][curr_y],color[curr_x-
1][curr_y]) && !mark[curr_x-1][curr_y])
                  makePoint2DList(curr_x-1,curr_y,pl_address);
      }
      if (curr_x<x) {

      if(isSameColor(color[curr_x][curr_y],color[curr_x+1][curr_y]) &&
!mark[curr_x+1][curr_y])
                  makePoint2DList(curr_x+1,curr_y,pl_address);
      }
      if (curr_y>0) {
            if(isSameColor(color[curr_x][curr_y],color[curr_x][curr_y-
1]) && !mark[curr_x][curr_y-1])
                  makePoint2DList(curr_x,curr_y-1,pl_address);
      }
      if (curr_y<y) {

      if(isSameColor(color[curr_x][curr_y],color[curr_x][curr_y+1]) &&
!mark[curr_x][curr_y+1])
                  makePoint2DList(curr_x,curr_y+1,pl_address);
      }
}
```

```
/*    Shape2D.h

      Definition of Shape2D class, used for storing polar equivalents
      of points forming shapes in the original image.  Also contains
      definitions of Ring, RingList, CenterList, StdShape2D,
      Polar2DList, and Shape2DList structures explained below.
*/

#ifndef _SHAPE2DH_
#define _SHAPE2DH_

#include <iostream.h>
#include <math.h>
#include "misc.h"

//Standardized Shape structures

/*    Ring

      A list of polar points with equal radius values
*/

struct RingData {
      float theta;      //theta value of a point in the ring
      RingData *next;   // the next counterclockwise point in the ring
};
typedef RingData *Ring;

/*    RingList

      A list of Rings with different radii
*/

struct RingListData {
      Ring ring;
      float r;    //radius of current ring
      int ring_size;    //maximum number of points in current ring
      RingListData *next;           //the next smaller ring in the list
};
typedef RingListData *RingList;

/*    CenterList

      A list of coordinates for centers of shapes, based on their scene
      coordinates.  The centers of shapes are calculated by averaging
      the x and y coordinates of points within a shape.
*/

struct CenterListData {
      float x;
      float y;
      CenterListData *next;
};
typedef CenterListData *CenterList;
```

```
/*  StdShape2D

      A standardized shape, consisting of a RingList containing
      coordinate information, the color of the shape, a list of the
      centers of known instances of the shape, and other statistical
      information about the shape.
*/

struct StdShape2DData {
      RingList rl;      //the shape's points
      int point_total, ring_total;  //total number of points and rings
                                    //in the shape
      unsigned char *color;   //the color of the shape (3-character
                              //array)
      float avg_r, stddev_r;  //mean and standard deviation of points
                              //in shape
      float r_error, theta_error;   //r and theta errors of shape
      CenterList cl;    //list of centers of known similar shapes
};
typedef StdShape2DData *StdShape2D;

//Unstandardized Shape structures

/*    Polar2DList

      A doubly-linked list of 2-dimesional polar coordinates, sorted by
      theta.
*/

struct Polar2DListData
{
      float r;
      float theta;
      Polar2DListData *next_cw, *next_ccw;//next clockwise and
                                          //counterclockwise points in
                                          //list, respectively
};
typedef Polar2DListData *Polar2DList;

/*    Shape2D

      An unstandardized shape, containing points sorted by their theta
      values.
*/

class Shape2D {
      protected:
            Polar2DList pol;  //the shape's points
            int point_total;  //the total number of points in the shape
            unsigned char *color;   //the color of the shape (3-
                                    //character array)
            float avg_r, stddev_r, max_r; //mean, standard deviation,
                                          //and maximum value of r
            int min_x, max_x, min_y, max_y;//minimum and maximum values
            //of the shape's original x and y coordinates, based on
            //their absolute positions in the BMP file
            float center_x, center_y;//the coordinates of the shape's
            //center, based on the coordinates of the points in the BMP
            //file

      public:
            Shape2D(Point2DList pl);
            int getColorValue(void)
            {
                  return color[R] + color[G] + color[B];
            }
            int GetMinX(void)
            {
                  return min_x;
            }
            int GetMaxX(void)
            {
                  return max_x;
            }
            int GetMinY(void)
            {
                  return min_y;
            }
            int GetMaxY(void)
            {
                  return max_y;
            }
            static float sameColor(Shape2D *s1,Shape2D *s2)
            {
                  return ((float)(abs(s1->color[R]-s2->color[R]) +
abs(s1->color[G]-s2->color[G]) + abs(s1->color[B]-s2->color[B])))/765;
            }
            static float sameSize(Shape2D *s1,Shape2D *s2)
            {
                  if (s1->avg_r<s2->avg_r)
                        return s1->avg_r/s2->avg_r;
                  else
                        return s2->avg_r/s1->avg_r;
            }
            static float sameStdDev(Shape2D *s1,Shape2D *s2)
            {
                  if (s1->stddev_r<s2->stddev_r)
                        return s1->stddev_r/s2->stddev_r;
                  else
                        return s2->stddev_r/s1->stddev_r;
            }
            float sameOrientation(Shape2D *s1,Shape2D *s2,float
r_error,float theta_error);
            float sameShape(Shape2D *s1,Shape2D *s2,float r_error,float
theta_error);
            StdShape2D standardize(int point_limit);
            void printInfo(void) {
                  cout << "Color: " << color[R] << "\t" << color[G] <<
"\t" << color[B] << "\n";
                  cout << "Average r: " << avg_r << "\n";
                  cout << "Std. dev. of r: " << stddev_r << "\n";
            }
};
```

```
/*    Shape2DList

      A linked list of Shape2D objects
*/

struct Shape2DListData {
      Shape2D *s;
      Shape2DListData *next;
};
typedef Shape2DListData *Shape2DList;

#endif
```

```
/*	Shape2D.cpp

	Function definitions for the Shape2D class
*/

#include <iostream.h>
#include <math.h>
#include "Shape2D.h"
#include "misc.h"

/*	Shape2D::Shape2D(Point2DList pl)

	Constructor for Shape2D class; converts a list of rectangular
	coordinates to polar coordinates, then calculates the shape's
	other attributes.

	Parameters:

	  pl - a list of recangular coordinates
*/

Shape2D::Shape2D(Point2DList pl)
{
	Point2DList pl_counter;
	Polar2DList new_pol, pol_counter;
	double r_total = 0;
	double stddev_r_total = 0;
	int x_total = 0, y_total = 0;

	color = new unsigned char[3];
	color[R] = pl->color[R];
	color[G] = pl->color[G];
	color[B] = pl->color[B];

	point_total = 0;
	pl_counter = pl;

	while (pl_counter!=NULL) {
		point_total++;
		x_total += pl_counter->x;
		y_total += pl_counter->y;
		pl_counter = pl_counter->next;
	}

	center_x = (float)x_total/(float)point_total;
	center_y = (float)y_total/(float)point_total;

	pol = NULL;
	pl_counter = pl;
	max_r = 0.0f;

	while (pl_counter!=NULL) {	//coordinate transformation loop
		new_pol = new Polar2DListData;
		new_pol->r = (float)sqrt(((float)pl_counter->x-
center_x)*((float)pl_counter->x-center_x) + ((float)pl_counter->y-
center_y)*((float)pl_counter->y-center_y));
		r_total += new_pol->r;
```

```
            if (new_pol->r > max_r) {
                  max_r = new_pol->r;
            }
            if (pl_counter->x != center_x) {
                  if ((float)pl_counter->x-center_x>0.0f) {
                        new_pol->theta =
(float)atan(((float)pl_counter->y-center_y)/((float)pl_counter->x-
center_x));
                  }
                  else if ((float)pl_counter->y-center_y>0.0f) {
                        new_pol->theta =
(float)atan(((float)pl_counter->y-center_y)/((float)pl_counter->x-
center_x)) + PI;
                  }
                  else {
                        new_pol->theta =
(float)atan(((float)pl_counter->y-center_y)/((float)pl_counter->x-
center_x)) - PI;
                  }
            }
            else if (((float)pl_counter->y-center_y)>0)
                  new_pol->theta = PI_OVER_2;
            else
                  new_pol->theta = -PI_OVER_2;

            if (pol==NULL) {
                  new_pol->next_ccw = new_pol;
                  new_pol->next_cw = new_pol;
                  pol = new_pol;
            }
            else if (new_pol->theta<pol->theta) {
                  new_pol->next_ccw = pol;
                  new_pol->next_cw = pol->next_cw;
                  pol->next_cw->next_ccw = new_pol;
                  pol->next_cw = new_pol;
                  pol = new_pol;
            }
            else {
                  pol_counter = pol->next_ccw;
                  while ((pol_counter!=pol) && (pol_counter->theta <
new_pol->theta)) {
                        pol_counter = pol_counter->next_ccw;
                  }
                  new_pol->next_ccw = pol_counter;
                  new_pol->next_cw = pol_counter->next_cw;
                  pol_counter->next_cw->next_ccw = new_pol;
                  pol_counter->next_cw = new_pol;
            }
            pl_counter = pl_counter->next;
      }

      avg_r = (float)(r_total/(double)point_total);
      pol_counter = pol;

      if (avg_r==0.0f) {
            stddev_r = 0.0f;
      }
      else {
            do {  //standard deviation calculation loop
                  stddev_r_total += fabs(pol_counter->r - avg_r);
                  pol_counter = pol_counter->next_ccw;
            } while (pol_counter!=pol);
            stddev_r = (float)(stddev_r_total / (double)point_total) /
avg_r;
      }

      pl_counter = pl;
      min_x = BMP_WIDTH;
      max_x = 0;
      min_y = BMP_WIDTH;
      max_y = 0;

      while (pl_counter!=NULL) {    //min/max x/y finder loop
            if (pl_counter->x<min_x)
                  min_x = pl_counter->x;
            if (pl_counter->x>max_x)
                  max_x = pl_counter->x;
            if (pl_counter->y<min_y)
                  min_y = pl_counter->y;
            if (pl_counter->y>max_y)
                  max_y = pl_counter->y;
            pl_counter = pl_counter->next;
      }
}

/*    Shape2D::sameOrientation(Shape2D *s1,Shape2D *s2,float
r_error,float theta_error)

      Compares two Shape2D objects to see if they have the same
      shape and orientaion.  Not used in final code.

      Parameters:

        s1,s2 - the two shapes being compared
        r_error - the maximum allowable error for the r value of a
        point
        theta_error - multiplier used in calculating maximum allowable
        error for the theta value of a point

      Return Value:

        A decimal value between 0 and 1, where 1 indicates
        a perfect match and 0 indicates a complete mismatch.
*/

float Shape2D::sameOrientation(Shape2D *s1,Shape2D *s2,float
r_error,float theta_error)
{
      Polar2DList pol_counter_1, pol_counter_2;
      float size_ratio;
      float result_1, result_2;
      float match_total;
      bool match_found;
      int i, j;
```

```
	if (s2->avg_r==0.0f)
		return 1.0f;

	size_ratio = s1->avg_r/s2->avg_r;
	match_total = 0;
	pol_counter_1 = s1->pol;

	for (i=0;i<s1->point_total;i++) {
		pol_counter_2 = s2->pol;
		match_found = false;
		for (j=0;(j<s2->point_total) && (pol_counter_2->theta -
pol_counter_1->theta <= theta_error*PI_OVER_2/pol_counter_1->r) &&
!match_found;j++) {
			if ((pol_counter_1->r==0.0f) && (fabs(pol_counter_2-
>r*size_ratio - pol_counter_1->r) <= r_error*size_ratio)) {
				match_total++;
				match_found = true;
			}
			else if ((fabs(pol_counter_2->r*size_ratio -
pol_counter_1->r) <= r_error*size_ratio) && (fabs(pol_counter_2->theta
- pol_counter_1->theta) <= theta_error*PI_OVER_2/pol_counter_1->r)) {
				match_total++;
				match_found = true;
			}
			pol_counter_2 = pol_counter_2->next_ccw;
		}
		pol_counter_1 = pol_counter_1->next_ccw;
	}

	result_1 = match_total/((float)s1->point_total);

	match_total = 0;
	pol_counter_1 = s2->pol;

	for (i=0;i<s2->point_total;i++) {
		pol_counter_2 = s1->pol;
		match_found = false;
		for (j=0;(j<s1->point_total) && (pol_counter_2->theta -
pol_counter_1->theta <= theta_error*PI_OVER_2/pol_counter_1->r) &&
!match_found;j++) {
			if ((pol_counter_1->r==0.0f) && (fabs(pol_counter_2-
>r*size_ratio - pol_counter_1->r) <= r_error*size_ratio)) {
				match_total++;
				match_found = true;
			}
			else if ((fabs(pol_counter_2->r - pol_counter_1->r *
size_ratio) <= r_error/size_ratio) && (fabs(pol_counter_2->theta -
pol_counter_1->theta) <= theta_error*PI_OVER_2/pol_counter_1->r)) {
				match_total++;
				match_found = true;
			}
			pol_counter_2 = pol_counter_2->next_ccw;
		}
		pol_counter_1 = pol_counter_1->next_ccw;
	}
	result_2 = match_total/((float)s2->point_total);

	return result_1*result_2;
}

/*	Shape2D::sameShape(Shape2D *s1,Shape2D *s2,float r_error,float
theta_error)

	Compares two Shape2D objects to see if they have the same
	shape, regardless of orientaion.  Not used in final code.

	Parameters:

	  s1,s2 - the two shapes being compared
	  r_error - the maximum allowable error for the r value of a
	  point
	  theta_error - multiplier used in calculating maximum allowable
	  error for the theta value of a point

	Return Value:

	  A decimal value between 0 and 1, where 1 indicates
	  a perfect match and 0 indicates a complete mismatch.
*/

float Shape2D::sameShape(Shape2D *s1,Shape2D *s2,float r_error,float
theta_error)
{
	Polar2DList pol_counter_1, pol_counter_2, pol_counter_2_start,
best_pol_counter_2_start, pol_counter_2_mark;
	Shape2D *small_s, *large_s;
	float size_ratio;
	float result_1, result_2;
	float match_total, best_match_total;
	float theta_diff, theta_diff_2;
	bool match_found;
	int i, j, k;
	long order;
	float best_theta_diff;

	if (s1->avg_r<s2->avg_r) {
		small_s = s1;
		large_s = s2;
	}
	else {
		small_s = s2;
		large_s = s1;
	}

	if (large_s->avg_r==0.0f)
		return 1.0f;

	size_ratio = small_s->avg_r/large_s->avg_r;
	pol_counter_2_start = large_s->pol;
	best_match_total = 0;
	best_theta_diff = 0;
	best_pol_counter_2_start = pol_counter_2_start;
```

```
        order = 0;

        for (k=0;k<large_s->point_total;k++) {
                pol_counter_1 = small_s->pol;
                while ((fabs(pol_counter_2_start->r*size_ratio -
pol_counter_1->r) > r_error*size_ratio) && (k<large_s->point_total)) {
                        pol_counter_2_start = pol_counter_2_start->next_ccw;
                        k++;
                }
                pol_counter_2 = pol_counter_2_start;
                theta_diff = pol_counter_2->theta - pol_counter_1->theta;
                match_total = 0;
                for (i=0;i<small_s->point_total;i++) {
                        match_found = false;
                        pol_counter_2_mark = pol_counter_2;
                        theta_diff_2 = pol_counter_2->theta - pol_counter_1-
>theta - theta_diff;
                        if (theta_diff_2 < -PI)
                                theta_diff_2 += PI_TIMES_2;
                        if (theta_diff_2 <= theta_error * PI_OVER_2 /
pol_counter_1->r) {
                                j = 0;
                                while (!match_found && (j<large_s->point_total)
&& (theta_diff_2 <= theta_error*PI_OVER_2/pol_counter_1->r)) {
                                        order++;
                                        if ((pol_counter_1->r==0.0f) &&
(fabs(pol_counter_2->r*size_ratio - pol_counter_1->r) <=
r_error*size_ratio)) {
                                                match_total++;
                                                match_found = true;
                                        }
                                        else if ((fabs(pol_counter_2->r *
size_ratio - pol_counter_1->r) <= r_error*size_ratio) && (theta_diff_2
>= -theta_error*PI_OVER_2/pol_counter_1->r)) {
                                                match_total++;
                                                match_found = true;
                                        }
                                        pol_counter_2 = pol_counter_2->next_ccw;
                                        theta_diff_2 = pol_counter_2->theta -
pol_counter_1->theta - theta_diff;
                                        if (theta_diff_2 < -PI)
                                                theta_diff_2 += PI_TIMES_2;
                                        j++;
                                }
                        }
                        pol_counter_2 = pol_counter_2_mark;
                        theta_diff_2 = pol_counter_2->theta - pol_counter_1-
>theta - theta_diff;
                        if (theta_diff_2 < -PI)
                                theta_diff_2 += PI_TIMES_2;
                        if (theta_diff_2 >= -
theta_error*PI_OVER_2/pol_counter_1->r) {
                                j = 0;
                                while (!match_found && (j<large_s->point_total)
&& (theta_diff_2 >= -theta_error*PI_OVER_2/pol_counter_1->r)) {
                                        order++;
```

```
                              if ((pol_counter_1->r==0.0f) &&
(fabs(pol_counter_2->r*size_ratio - pol_counter_1->r) <= r_error *
size_ratio)) {
                                    match_total++;
                                    match_found = true;
                              }
                              else if ((fabs(pol_counter_2->r *
size_ratio - pol_counter_1->r) <= r_error*size_ratio) && (theta_diff_2
<= theta_error*PI_OVER_2/pol_counter_1->r)) {
                                    match_total++;
                                    match_found = true;
                              }
                              pol_counter_2 = pol_counter_2->next_cw;
                              theta_diff_2 = pol_counter_2->theta -
pol_counter_1->theta - theta_diff;
                              if (theta_diff_2 < -PI)
                                    theta_diff_2 += PI_TIMES_2;
                              j++;
                        }
                  }
                  pol_counter_1 = pol_counter_1->next_ccw;

            }
            if (match_total>best_match_total) {
                  best_match_total = match_total;
                  best_theta_diff = theta_diff;
                  best_pol_counter_2_start = pol_counter_2_start;
            }
            pol_counter_2_start = pol_counter_2_start->next_ccw;
      }

      result_1 = best_match_total/((float)small_s->point_total);

      pol_counter_1 = best_pol_counter_2_start;
      pol_counter_2 = small_s->pol;
      theta_diff = -best_theta_diff;
      match_total = 0;

      for (i=0;i<large_s->point_total;i++) {
            match_found = false;
            pol_counter_2_mark = pol_counter_2;
            theta_diff_2 = pol_counter_2->theta - pol_counter_1->theta
- theta_diff;
            if (theta_diff_2 < -PI)
                  theta_diff_2 += PI_TIMES_2;
            else if (theta_diff_2 > PI)
                  theta_diff_2 -= PI_TIMES_2;
            if (theta_diff_2 <= theta_error*PI_OVER_2/pol_counter_2->r)
{
                  j = 0;
                  while (!match_found && (j<small_s->point_total) &&
(theta_diff_2 <= theta_error*PI_OVER_2/pol_counter_2->r)) {
                        order++;
                        if ((pol_counter_1->r==0.0f) &&
(fabs(pol_counter_2->r/size_ratio - pol_counter_1->r) <=
r_error*size_ratio)) {
                              match_total++;
                              match_found = true;
                        }
                        else if ((fabs(pol_counter_2->r/size_ratio -
pol_counter_1->r) <= r_error/size_ratio) && (theta_diff_2 >= -
theta_error*PI_OVER_2/pol_counter_2->r)) {
                              match_total++;
                              match_found = true;
                        }
                        pol_counter_2 = pol_counter_2->next_ccw;
                        theta_diff_2 = pol_counter_2->theta -
pol_counter_1->theta - theta_diff;
                        if (theta_diff_2 < -PI)
                              theta_diff_2 += PI_TIMES_2;
                        else if (theta_diff_2 > PI)
                              theta_diff_2 -= PI_TIMES_2;
                        j++;
                  }
            }
            pol_counter_2 = pol_counter_2_mark;
            theta_diff_2 = pol_counter_2->theta - pol_counter_1->theta
- theta_diff;
            if (theta_diff_2 < -PI)
                  theta_diff_2 += PI_TIMES_2;
            else if (theta_diff_2 > PI)
                  theta_diff_2 -= PI_TIMES_2;
            if (theta_diff_2 >= -theta_error*PI_OVER_2/pol_counter_2-
>r) {
                  j = 0;
                  while (!match_found && (j<large_s->point_total) &&
(theta_diff_2 >= -theta_error*PI_OVER_2/pol_counter_2->r)) {
                        order++;
                        if ((pol_counter_1->r==0.0f) &&
(fabs(pol_counter_2->r/size_ratio - pol_counter_1->r) <= r_error /
size_ratio)) {
                              match_total++;
                              match_found = true;
                        }
                        else if ((fabs(pol_counter_2->r/size_ratio -
pol_counter_1->r) <= r_error/size_ratio) && (theta_diff_2 <=
theta_error*PI_OVER_2/pol_counter_2->r)) {
                              match_total++;
                              match_found = true;
                        }
                        pol_counter_2 = pol_counter_2->next_cw;
                        theta_diff_2 = pol_counter_2->theta -
pol_counter_1->theta - theta_diff;
                        if (theta_diff_2 < -PI)
                              theta_diff_2 += PI_TIMES_2;
                        else if (theta_diff_2 > PI)
                              theta_diff_2 -= PI_TIMES_2;
                        j++;
                  }
            }
            pol_counter_1 = pol_counter_1->next_ccw;
      }

      result_2 = match_total/((float)large_s->point_total);
      return result_1*result_2;
}

/* StdShape2D Shape2D::standardize(int point_limit)

      Converts the current Shape2D object into a StdShape2D structure.

      Parameters:

        point_limit - maximum number of points allowed in StdShape2D
        representation being created

      Return Value:

        The StdShape2D structure created from the current Shape2D
        object
*/

StdShape2D Shape2D::standardize(int point_limit)
{
      StdShape2D new_ss = new StdShape2DData;
      Ring r_counter;
      RingList rl_counter;
      Polar2DList pol_counter;
      float curr_theta, theta_inc;
      float r_total = 0.0f, stddev_total = 0.0f;
      bool match_found;
      int i, j;

      new_ss->cl = new CenterListData;
      new_ss->cl->x = center_x;
      new_ss->cl->y = center_y;
      new_ss->cl->next = NULL;

      new_ss->color = new unsigned char[3];
      new_ss->color[R] = color[R];
      new_ss->color[G] = color[G];
      new_ss->color[B] = color[B];

      if (point_total<=point_limit) {
            new_ss->ring_total = (int)max_r;
      }
      else {
            new_ss->ring_total = (int)((sqrt(4*point_limit/PI+1)-1)/2);
      }

      new_ss->r_error = 1.0f/(float)new_ss->ring_total;
      new_ss->theta_error = PI_TIMES_2/(float)((int)(PI_TIMES_2 *
new_ss->ring_total));
      new_ss->point_total = 0;
      new_ss->rl = new RingListData;
      rl_counter = new_ss->rl;

      for (i=new_ss->ring_total;i>0;i--) {      //StdShape2D creation
loop
            rl_counter->ring_size = (int)(i*PI_TIMES_2);
```

```
			rl_counter->ring = new RingData;
			rl_counter->r = (float)i/(float)new_ss->ring_total;
			r_counter = rl_counter->ring;
			r_counter->theta = -PI_TIMES_2;
			pol_counter = pol;
			curr_theta = -PI;
			theta_inc = PI_TIMES_2/(float)((int)(i*PI_TIMES_2));
			for (j=0;j<(int)(i*PI_TIMES_2);j++) {//Ring creation loop
				match_found = false;
				while ((fabs(pol_counter->theta-curr_theta) <=
theta_inc) && !match_found) {
					if (fabs(pol_counter->r-rl_counter->r * max_r)
<= 1.0f) {
						if (r_counter->theta>-PI_TIMES_2) {
							r_counter->next = new RingData;
							r_counter = r_counter->next;
						}
						r_total += rl_counter->r;
						new_ss->point_total++;
						r_counter->theta = curr_theta;
						match_found = true;
					}
					pol_counter = pol_counter->next_ccw;
				}
				while (fabs(pol_counter->theta-curr_theta) <=
theta_inc) {
					pol_counter = pol_counter->next_ccw;
				}
				curr_theta += theta_inc;
			}
			if (r_counter->theta==-PI_TIMES_2) {
				delete r_counter;
				rl_counter->ring = NULL;
			}
			else {
				r_counter->next = rl_counter->ring;
			}
			if (i>1) {
				rl_counter->next = new RingListData;
				rl_counter = rl_counter->next;
			}
		}

		if (new_ss->point_total==0) {
			return NULL;
		}

		rl_counter->next = NULL;
		new_ss->avg_r = r_total/(float)new_ss->point_total;
		rl_counter = new_ss->rl;

		do {	//stadard deviation calculation loop
			if (rl_counter->ring!=NULL) {
				r_counter = rl_counter->ring;
				do {
					stddev_total += (float)fabs(rl_counter->r-
new_ss->avg_r);
```

```
                                r_counter = r_counter->next;
                        } while (r_counter!=rl_counter->ring);
                }
                rl_counter = rl_counter->next;
        } while (rl_counter!=NULL);

        new_ss->stddev_r = stddev_total/new_ss->point_total;

        return new_ss;
}
```

```
/*	Concept.h

	Contains information for Concept class, used for
	storing previously identified shapes.  Also contains
	ConceptList structure, which is used to form a linked
	list of concepts.
*/

#ifndef _CONCEPTH_
#define _CONCEPTH_

#include "Shape2D.h"
#include "misc.h"

class Concept {
	protected:
		StdShape2D s;		//Representative shape instance for
					//concept
		int instances;		//Number of found instances of the
					//concept
	public:
		Concept(StdShape2D new_s) {
			s = new_s;
			instances = 1;
		}
		float sameShape(StdShape2D s1,StdShape2D s2);
		float sameColor(StdShape2D s1,StdShape2D s2) {
			return ((float)(765 - (abs(s1->color[R]-s2->color[R])
+ abs(s1->color[G]-s2->color[G]) + abs(s1->color[B]-s2->color[B])))) /
765;
		}
		bool isInstance(StdShape2D nominee);
		StdShape2D getS(void) {
			return s;
		}
		int getInstances(void) {
			return instances;
		}
		void printStdShape(void);
};

struct ConceptListData {
	Concept *c;
	ConceptListData *next;
};
typedef ConceptListData *ConceptList;

#endif
```

```
/*    Concept.cpp

      Function definitions for Concept class.
*/

#include <iostream.h>
#include "Concept.h"
#include "Shape2D.h"

/*    float Concept::sameShape(StdShape2D s1,StdShape2D s2)

      Compares two StdShape2D structures (see Shape2D.h).

      Parameters:

        s1, s2 - The two shapes being compared.

      Return Value:

        A decimal value between 0 and 1, where 1 indicates
        a perfect match and 0 indicates a complete mismatch.
*/

float Concept::sameShape(StdShape2D s1,StdShape2D s2)
{
      StdShape2D s_temp;
      RingList rl_counter_1, rl_counter_2;
      Ring r_counter_1, r_counter_2, r_counter_1_start;
      int match_total_1, match_total_2, best_match_total_1,
best_match_total_2;
      float result, best_result = 0.0f;
      float theta_diff = 0.0f, theta_inc, theta_1_adjusted,
last_theta_1_adjusted;
      bool match_found, ring_1_start, ring_2_start, new_ring_1;

      if (s1->rl->ring_size>s2->rl->ring_size) {
            s_temp = s1;
            s1 = s2;
            s2 = s_temp;
      }

      theta_inc = PI_TIMES_2/(float)s2->rl->ring_size;

      do {  //    Main comparison loop
            rl_counter_1 = s1->rl;
            rl_counter_2 = s2->rl;
            match_total_1 = 0;
            match_total_2 = 0;
            new_ring_1 = true;
            do {  //    Orientation comparison loop
                  r_counter_1 = rl_counter_1->ring;
                  r_counter_2 = rl_counter_2->ring;
                  match_found = false;
                  theta_1_adjusted = -PI_OVER_2;
                  do {
                        last_theta_1_adjusted = theta_1_adjusted;
            theta_1_adjusted = r_counter_1->theta +
theta_diff;
            if (theta_1_adjusted>PI) {
               theta_1_adjusted -= PI_TIMES_2;
            }
            if (theta_1_adjusted>last_theta_1_adjusted) {
               r_counter_1 = r_counter_1->next;
            }
         } while (theta_1_adjusted>last_theta_1_adjusted);
         r_counter_1_start = r_counter_1;
         ring_1_start = true;
         ring_2_start = true;
         do {  //    Ring comparison loop
            theta_1_adjusted = r_counter_1->theta +
theta_diff;
            if (theta_1_adjusted>PI) {
               theta_1_adjusted -= PI_TIMES_2;
            }
            if (fabs(theta_1_adjusted-r_counter_2->theta) <
s1->theta_error/rl_counter_1->r-0.0001f) {
               if (ring_2_start) {
                  match_total_2++;
                  match_found = true;
                  r_counter_2 = r_counter_2->next;
                  ring_2_start = false;
               }
               else if (r_counter_2!=rl_counter_2->ring)
               {
                  match_total_2++;
                  match_found = true;
                  r_counter_2 = r_counter_2->next;
               }
               else {
                  r_counter_1 = r_counter_1_start;
               }
            }
            else if (theta_1_adjusted>r_counter_2->theta) {
               if (ring_2_start) {
                  r_counter_2 = r_counter_2->next;
                  ring_2_start = false;
               }
               else if (r_counter_2!=rl_counter_2->ring)
               {
                  r_counter_2 = r_counter_2->next;
               }
               else {
                  r_counter_1 = r_counter_1_start;
               }
            }
            else {
               if (match_found && new_ring_1) {
                  match_total_1++;
               }
               match_found = false;
               if (ring_1_start) {
                  r_counter_1 = r_counter_1->next;
                  ring_1_start = false;
```

```
                              }
                              else if (r_counter_1!=r_counter_1_start)
                              {
                                    r_counter_1 = r_counter_1->next;
                              }
                              else {
                                    r_counter_2 = rl_counter_2->ring;
                              }
                        }
                  } while ((r_counter_1!=r_counter_1_start) ||
(r_counter_2!=rl_counter_2->ring));
                  rl_counter_2 = rl_counter_2->next;
                  while (rl_counter_2!=NULL) {
                        if (rl_counter_2->ring!=NULL)
                              break;
                        else
                              rl_counter_2 = rl_counter_2->next;
                  }
                  new_ring_1 = false;
                  while ((rl_counter_1!=NULL) && (rl_counter_2!=NULL))
                  {
                        if ((fabs(rl_counter_1->r-rl_counter_2->r) <
s1->r_error-0.0001) && (rl_counter_1->ring!=NULL) && (rl_counter_2-
>ring!=NULL))
                              break;
                        if (rl_counter_1->r>rl_counter_2->r) {
                              while (rl_counter_1!=NULL) {
                                    if (((rl_counter_1->r <
rl_counter_2->r) || (fabs(rl_counter_1->r-rl_counter_2->r)<s1->r_error-
0.0001)) && (rl_counter_1->ring!=NULL))
                                          break;
                                    rl_counter_1 = rl_counter_1->next;
                                    new_ring_1 = true;
                              }
                        }
                        else {
                              while (rl_counter_2!=NULL) {
                                    if (((rl_counter_2->r<rl_counter_1-
>r) || (fabs(rl_counter_1->r-rl_counter_2->r)<s1->r_error-0.0001)) &&
(rl_counter_1->ring!=NULL))
                                          break;
                                    rl_counter_2 = rl_counter_2->next;
                              }
                        }
                  }
            } while((rl_counter_1!=NULL) && (rl_counter_2!=NULL));
            result = ((float)match_total_1/(float)s1->point_total) *
((float)match_total_2/(float)s2->point_total);
            if (result>best_result) {
                  best_match_total_1 = match_total_1;
                  best_match_total_2 = match_total_2;
                  best_result = result;
            }
            theta_diff += theta_inc;
      } while (theta_diff<PI_TIMES_2);

      return best_result;
```

```
}

/*	bool Concept::isInstance(StdShape2D nominee)

	Determines whether or not a shape is a member of this concept.

	Parameters:

	  nominee - Shape being compared to current concept.

	Return Value:

	  Boolean value answering the question, "Is the new shape
	  an instance of the current concept?"
*/

bool Concept::isInstance(StdShape2D nominee)
{
	CenterList cl_counter;

	if ((sameShape(nominee,s)>=MIN_SAMENESS) &&
(sameColor(nominee,s)>=MIN_SAMENESS)) {
		instances++;
		cl_counter = s->cl;
		while (cl_counter->next!=NULL) {
			cl_counter = cl_counter->next;
		}
		cl_counter->next = nominee->cl;
		if (nominee->point_total>s->point_total) {
			nominee->cl = s->cl;
			s = nominee;
		}
		return true;
	}
	else {
		return false;
	}
}

/*	void Concept::printStdShape(void)

	Displays information about the current concept's color
	and intstances.
*/

void Concept::printStdShape(void)
{
	CenterList cl_counter;

	cout << "Color: Red=" << (int)s->color[R] << ", Green=" <<
(int)s->color[G] << ", Blue=" << (int)s->color[B] << "\n";
	cl_counter = s->cl;

	while (cl_counter!=NULL) {
		cout << "Center at (" << cl_counter->x << "," <<
cl_counter->y << ")\n";
		cl_counter = cl_counter->next;
```

```
                }
        }
```

```
/*    Tommy.h

      The Tommy class, responsible for using the other classes to
      extract shapes from image files and for categorizing the shapes
      into concepts.
*/

#ifndef _TOMMYH_
#define _TOMMYH_

#include "Concept.h"
#include "misc.h"

class Tommy {
      protected:
            ConceptList cl, last;   //the list of known concepts, and
                                    //the last concept in that list
      public:
            Tommy(int a);
            void processFile(char *bitmap);
            void addConcept(Concept *new_c);
            void printConceptList(void);
};

#endif
```

```
/*    Tommy.cpp

      Function definitions for the Tommy class.
*/

#include <stdio.h>
#include <iostream.h>
#include "Tommy.h"
#include "Scene2D.h"
#include "View2D.h"
#include "Shape2D.h"
#include "Concept.h"

/*    Tommy::Tommy(int a)

      The constructor for the Tommy class; creates an empty concept
      list.

      Parameters:

        a - a dummy variable created to fix a glitch in the compiler
        that caused the default constructor to be called instead of
        this one
*/

Tommy::Tommy(int a)
{
      cl = NULL;
      last = NULL;
}

/*    void Tommy::processFile(char *bitmap)

      Reads in a BMP file and adds its shapes to the current concept
      list.

      Parameters:

        bitmap - the file name of the BMP file
*/

void Tommy::processFile(char *bitmap)
{
      Scene2D *sc;
      View2D *v;
      Shape2DList sl, sl_counter;
      ConceptList cl_counter;
      StdShape2D std;

      sc = new Scene2D(bitmap);
      v = sc->makeView2DPtr();
      sl = v->makeShape2DList();
      sl_counter = sl;

      while (sl_counter!=NULL) {    //loop comparing shapes to known
                                    //concepts
            std = sl_counter->s->standardize(POINT_LIMIT);
            if (std!=NULL) {
                  cl_counter = cl;
                  while (cl_counter!=NULL) {
                        if (cl_counter->c->isInstance(std))
                              break;
                        cl_counter = cl_counter->next;
                  }
                  if (cl_counter==NULL) {
                        addConcept(new Concept(std));
                  }
                  sl_counter = sl_counter->next;
            }
      }
}

/*    void Tommy::addConcept(Concept *new_c)

      Adds a new concept to the concept list, which is sorted in
      order of decreasing numbers of instances.

      Parameters:

        new_c - the new concept
*/

void Tommy::addConcept(Concept *new_c)
{
      ConceptList cl_counter, new_cl;

      new_cl = new ConceptListData;
      new_cl->c = new_c;
      new_cl->next = NULL;

      if (cl==NULL) {
            cl = new_cl;
      }
      else {
            cl_counter = cl;
            while (cl_counter->next!=NULL) {
                  if (new_c->getInstances()>=cl_counter->c-
>getInstances()) {
                        new_cl->next = cl_counter->next;
                        cl_counter->next = new_cl;
                        break;
                  }
                  cl_counter = cl_counter->next;
            }
            if (cl_counter->next==NULL) {
                  cl_counter->next = new_cl;
            }
      }
}

/*    void Tommy::printConceptList(void)

      Displays information on all of the concepts in the concept list.
*/
```

```
void Tommy::printConceptList(void)
{
      ConceptList cl_counter = cl;

      while (cl_counter!=NULL) {
            cl_counter->c->printStdShape();
            cout << "Instances: " << cl_counter->c->getInstances() <<
"\n";
            cout << "\n";
            cl_counter = cl_counter->next;
      }
}
```

```
/*    misc.h

      Miscellaneous constants, structures and
      function headers that didn't fit under any
      one class.
*/

#ifndef _MISCH_
#define _MISCH_

#define PI 3.14159265f  //Value of PI
#define PI_OVER_2 1.57079633f //Value of PI/2
#define PI_TIMES_2 6.28318531f      //Value of PI*2

#define BMP_WIDTH 256   //width, in pixels, of BMP file
#define BMP_HEIGHT 256  //height, in pixels, of BMP file

#define MAX_DIST 1000000.0f

#define R 0       //red value in a color array
#define G 1       //green value in a color array
#define B 2       //blue value in a color array

#define MIN_SAMENESS 0.9f     //threshold for determining whether two
                              //colors or shapes are the same

//"type" values for Tommy constructor; not used in current program
#define LOAD_TOMMY 0;
#define NEW_TOMMY 1;

#define POINT_LIMIT 5000      //maximum number of points in allowed in
                              //a standardized shape

/*    Point2DList

      A linked list of rectangular integer coordinates
*/

struct Point2DListData
{
      int x, y;
      unsigned char *color;   //color at coordinate (x.y)
      Point2DListData *next;
};
typedef Point2DListData *Point2DList;

bool isBlack (unsigned char *c);
bool isSameColor(unsigned char *c1,unsigned char *c2);

#endif
```

```
#include "misc.h"

/*    isBlack(unsigned char *c)

      Determines whether a given color value is black

      Parameters:

        c - A color value, stored as a 3-character array

      Return Value:

        "Is the specified color black?"
*/

bool isBlack(unsigned char *c)
{
      return (c[R]==0) && (c[G]==0) && (c[B]==0);
}

/*    isSameColor(unsigned char *c1,unsigned char *c2)

      Determines whether two color values are exactly the same

      Parameters:

        c1, c2 - the two colors being compared.

      Return Value:

        Are the two specified colors exactly the same?
*/

bool isSameColor(unsigned char *c1,unsigned char *c2)
{
      return (c1[R]==c2[R]) && (c1[G]==c2[G]) && (c1[B]==c2[B]);
}
```

```cpp
/*	testmain.cpp

	Location of the main function, which uses an instance of the
	Tommy class to convert a BMP file into a list of Concept objects.
*/

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "Bitmap.h"
#include "Scene2D.h"
#include "View2D.h"
#include "Shape2D.h"
#include "Concept.h"
#include "Tommy.h"
#include "misc.h"

void main(void)
{
	char bmp[40];
	Tommy *t;
	char end;
	int x = (int)22.9;

	cout << "Enter name of the image to examine: ";
	cin >> bmp;
	t = new Tommy(0);
	t->processFile(bmp);
	cout << "The image contained the following distinct concepts:\n";
	t->printConceptList();
	cin >> end;
}
```