CS106X
Spring 99-00

Handout #4
3/29/00

Assignment #1: The Magic Quilt Screensaver

Due: Fri, April 7 in class

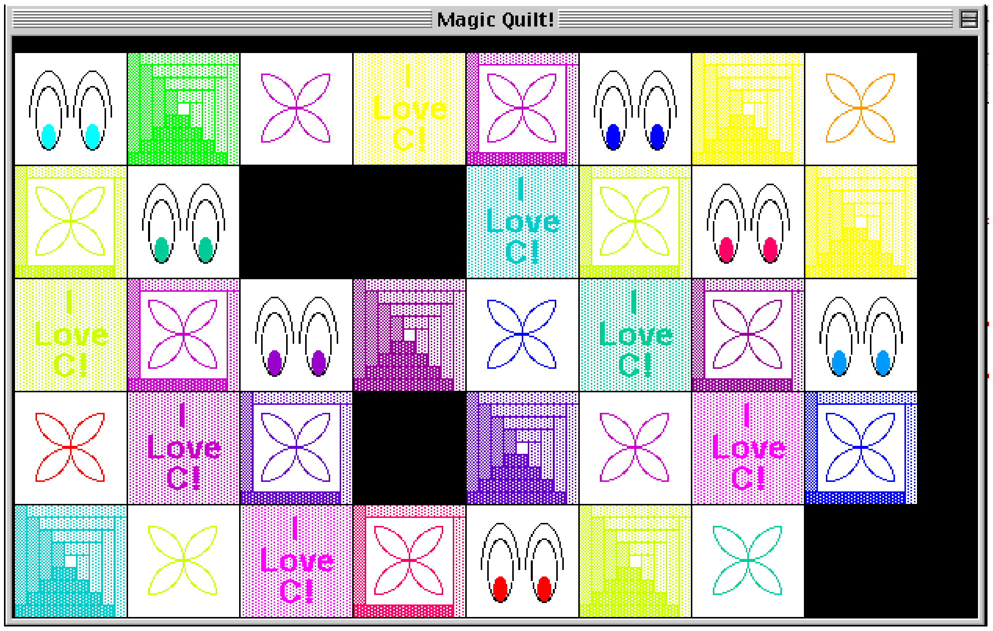
[In] making a quilt, you have to choose your combination carefully. The right choices will enhance your quilt. The wrong choices will dull the colors, hide their original beauty. There are no rules you can follow. You have to go by instinct and you have to be brave.

—from the film adaptation of Whitney Otto's *How to Make an American Quilt*, 1995

You may not realize that in addition to shaping you into an expert coder, CS106X is also going to develop your artistic and creative side with a special bonus of improved sewing skills. To start, you'll put on your programmer-artist hat and compose an electronic quilt screensaver as your first foray into C programming.

The purpose of this assignment is to give you practice in C syntax, using C control structures, designing and implementing functions that take parameters, and calling functions provided to you in a library, specifically the CS106 graphics and random libraries. Decomposition is an important part to solving this problem well. Rather than streaming your code in long linear routines, your approach should break the problem into smaller functions that in turn call on smaller building blocks you've identified as being common to many of the routines. Try to construct useful and versatile procedures, ones that could be used in a variety of situations.

We will not be able to cover all the details of C in class in time for you to do this assignment. This means that you will have to teach yourself any C fundamentals you aren't yet familiar with, and then learn about the fine points in lecture. If you're still wavering about CS106X versus A, this task will help you sort things out. This assignment is comparable to that given as the third or fourth in CS106A which gives a sense of the relative pace for the two courses. If your background is sufficient and you're engaged enough to put in the extra effort to make these kinds of quantum leaps through the material, then you have what it takes to succeed in CS106X!



The sampler quilt in action, randomly adding and re-drawing blocks·

Thanks to Marissa Mayer for the idea of using a screensaver. What a neat way to make the quilt come alive!

The goal

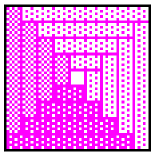
Your task is to explore the features exported from the interfaces `"graphics.h"` and `"extgraph.h"` in designing functions that draw a set of quilt blocks. Many quilts are made of up one or two blocks that are repeated and pieced together to form a unified effect, but a "sampler quilt" is made up of an assortment of different blocks, where each block demonstrates a particular quilting technique. Our sampler quilt will be made from five blocks, each of which will give you practice with different features of the graphics library.

You will put those blocks to work in a collection of screensavers. Screensavers are animation sequences that run when your computer is idle. The point of a screensaver is to prevent the "burning" of an image onto your screen. Should you leave your computer running for a long period of time with your I-Hum paper displayed, the continual display of the same image can imprint the phosphors on the inner surface of the monitor. The result of this "burning" is that you will later see discolored areas where the text of your paper was displayed. This may not be a problem if your paper was a literary masterpiece and you are thrilled to be able to see it every time you look at your monitor from now on. However, this usually isn't the case. The underlying concept of a screensaver is that if your screen is displaying something that is constantly changing, no portion of the screen will be overused or damaged.

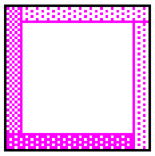
The quilt blocks

We have five block types in our sampler quilt. The contents of four blocks are given to you, the other block is your opportunity for creativity. The design for each block is controlled by one or more **`#define`**-d constants. The important thing to remember about defining constants is that the point is to make the behavior of the program easy to change. Each block will need to be able to be drawn in different colors and sizes for the various screensavers.

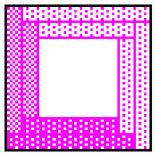
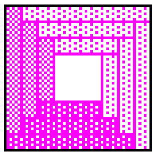
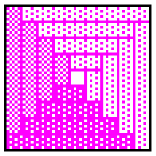
Block 0: Log cabin



The log cabin is an honest-to-goodness authentic pattern used in traditional quilting. It is composed of a series of frames, each of which is nested inside the next larger one. Each frame consists of four rectangular strips arranged to form a square. The outer frame, for example, looks like this:

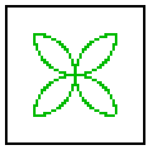


The strip on the bottom is filled with **`DenseFill`** (almost solid), the strip on the left with **`MediumFill`**, and the others with **`LightFill`** and **`VeryLightFill`**. Next you frame the same pattern of strips on the inset block and again, until you have drawn **`NFrames`** worth:

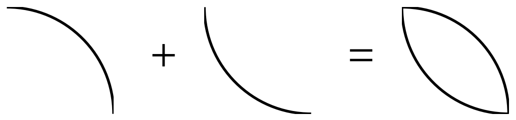
 (after 2 frames)  (after 3 frames)  (after 4, we're done!)

There is a little bit of math involved to figure what size and shape to make each of the rectangles. Given the **`NFrames`** constant and the size of the block being drawn, you can determine how wide each rectangular strip needs to be so that each of the frames and the interior square will all have the same width when completed. Study the finished pattern above and you can reason through what calculation is necessary.

Block 1: Flower petals

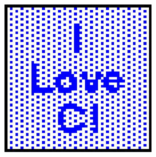


This block is a stylized representation of a four-petal flower and is included in the assignment mostly to give you practice drawing arcs. Each petal is composed of two quarter circles superimposed on top of each other as shown in the following enlarged picture of one petal:



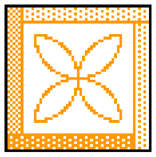
The only constant required to draw the flower petal is the ratio of the radius of the arc to the overall size of the block, which is defined as **`PetalRadiusRatio`**.

Block 2: I love C!



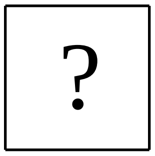
This block uses the text drawing functions of extgraph. The background of the block is filled with density **`LightFill`** and "I love C!" is drawn across three lines. Each line is horizontally centered and the three lines are vertically spaced. The centering and placement requires a little planning or use of **`TextStringWidth`**, **`GetFontHeight`** and **`GetFontAscent`**.

Block 3: Scrap combination



After making the Log Cabin and Flower blocks, there are just enough fabric scraps to create this hybrid block that combines the outer frame of the log cabin with the flower. The point of including this block is to emphasize the importance of software reuse. If you've designed your program well, you will have to write very little new code to create this design.

Block 4: User-defined



The last block is left as an opportunity for you to demonstrate your own creativity. You are encouraged to explore elliptical arcs, construct polygons, draw text, whatever suits your taste for the last block. Have fun, but don't spend too much time on this block until you get the rest of the program working and polished stylistically.

Combining blocks into screensavers

Now, let's put those pretty blocks together into something fun— animating sequences of flying, zooming, and twinkling quilt blocks! When the program runs, it allows the user to select one of three screensavers. You request the user's choice by displaying a menu like this:

```
Which screensaver would you like to see?
   Enter 1 for Sampler Quilt
   Enter 2 for Zooming Block
   Enter 3 for Bouncing Block
   Enter 0 to Quit
Your choice?
```

You should error-check the user's input to make sure it is a valid choice (0, 1, 2, or 3). If it is invalid, an appropriate error message should be printed and the prompt and menu should be repeated. If the user's choice is zero, your program should end. Otherwise, you run the chosen screensaver until the user clicks the mouse in the graphics window. At that time, your program should loop back to the menu shown above and let the user again pick a new screensaver to run. This cycle should continue until the user selects zero from the menu.

The sampler quilt screensaver

The sampler quilt screensaver (shown on the first page) animates the sampler quilt of blocks laid out in a regular pattern. It randomly chooses a location to add a new block and a random color for that block. It continues adding and replacing blocks in the quilt until the user clicks the mouse.

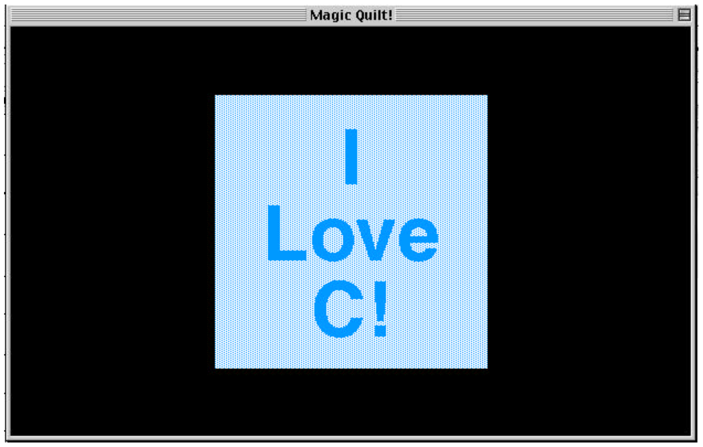
The window starts with a black background. Each quilt block is drawn in a 1-inch square and the blocks are laid out in a grid. For example, if the window is 7.5 inches wide, you can draw 7 blocks across (ignore the bit left over), one at position 0.0, the next at 1.0, and so on. On each iteration, choose a random grid location and draw the correct block there. You don't have to make provisions to find an empty location instead of drawing over a previously placed block, just choose randomly You should choose a random color to draw each block in, and thus the quilt will appear to "twinkle" because of the changing colors as blocks are re-drawn. Wait a bit (using **`Pause()`** from extgraph) after each block so the user can admire the new addition before you go on.

The different block types are arranged in the sampler quilt in a specific pattern. Each successive row in the quilt has the same blocks in the same cyclic order. The only difference is that the blocks in each row are shifted one position to the left relative to the blocks in the row underneath them. If you number the blocks 0, 1, 2, 3, and 4, and had space for a sampler quilt of 5 rows by 7 columns, the blocks would be arranged in the following configuration (read from the bottom row up to get the right effect). **Hint**: Think about how you could use the mod operator (%) to generate this pattern!

4	0	1	2	3	4	0
3	4	0	1	2	3	4
2	3	4	0	1	2	3
1	2	3	4	0	1	2
0	1	2	3	4	0	1

The zooming block screensaver

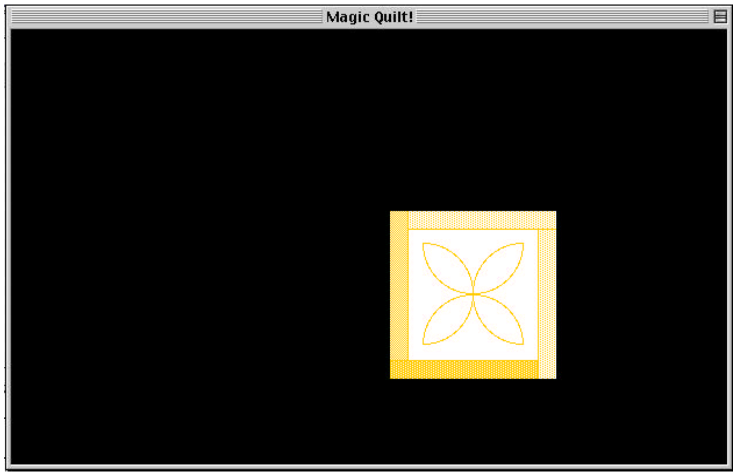
The zooming block draws a single block growing and shrinking in the center of the window.



The window starts with a black background. It randomly chooses one of the block types and a random color for that block. It first draws a small version of the block in the center of the window, pauses for a bit, then draws a slightly larger version, pauses, and so on until the block is as big as can be given the constraints of the window size. At that point, it starts shrinking, drawing the block in a smaller size each time, pausing in between, until it is back to the small starting size. A new random block type and random color are chosen and the process repeats until the user clicks the mouse to stop the madness. Note that you must "erase" the previous block by filling the rectangle it occupied with the background color to avoid leaving a trail behind as the block changes size.

The bouncing block screensaver

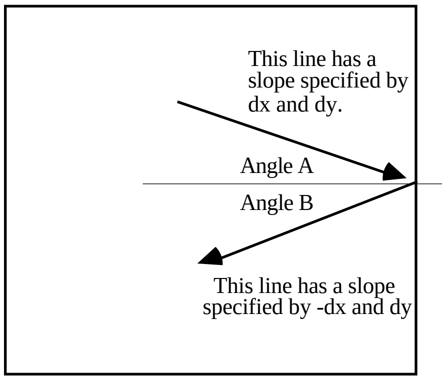
The bouncing block screensaver shows a single block moving across a black background and whenever it hits an edge of the graphics window, it changes block type and color and bounces off.



The window starts with a black background. It randomly chooses one of the block types and a random color for that block. It also choose a random velocity (expressed as dx and dy) for the block to move along between the **`Min`** and **`MaxVelocity`** constant constraints. It first draws the block in the center of the window, pauses a bit, then moves the block by dx in the x direction, dy in the y direction. It pauses a bit, then moves again, and so on. Like the zooming screensaver, you must "erase" the background before drawing the block in its new position to avoid leaving a trail behind.

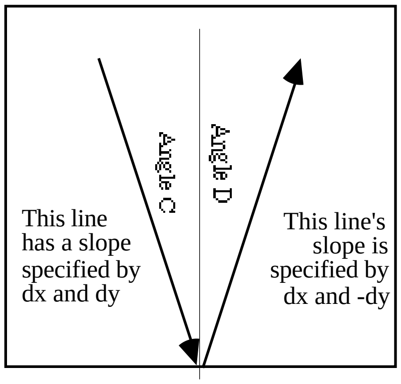
The bouncing effect should follow the law of physics. That is, the block should bounce off the wall at the same angle at which it hits. As a physicist would put it, "the angle of incidence equals the angle of reflection." Fortunately, this is easy to achieve. If the blocks hit a wall in the x-direction (i.e. a side wall), simply invert the sign of the value of dx (if it is positive it becomes negative and vice versa). If the block hits the top or bottom wall, invert the sign of dy. The following figures show how and why this works:

Horizontal bounces work like this:


This line has a slope specified by dx and dy.
Angle A
Angle B
This line has a slope specified by -dx and dy

Note: Angle A = Angle B

Vertical bounces work like this:


Angle C
Angle D
This line has a slope specified by dx and dy
This line's slope is specified by dx and -dy

Note: Angle C = Angle D

Each the block hits the wall, choose a new random block type and random color to bounce.

The Graphics and Extended Graphics libraries

The first few pages of section 5.3 in the text briefly introduce the minimal graphics library of lines and arcs. We will use these simple routines as well as the extended graphics functions which add the ability to draw elliptical arcs, draw text, set font size and style, create filled regions, draw in color, do erasing and animation, obtain mouse input, and more. We will not have much, if any, time to discuss these libraries in lecture, so you will need to read the interface files (both attached at the end of the handout) and experiment with some test code to learn what functionality is available and how to use it.

Strategy and Tactics

- The key to this assignment is working on one piece at a time, in isolation. There are way too many details to try to write it all the code at once. Start by focusing on one block at a time, and breaking it into manageable pieces. Test each helper and single block functions by calling them from the main function with different sizes, colors, and positions. Once you have all blocks working in isolation, string them together to do one screensaver, and grow that to the full collection of screensavers.

- Concentrate carefully on how you break the program down into pieces, and what parameters you need to pass to each function. Each of your functions will likely benefit from a comment which specifies what the routine does, what the parameters mean, and what the pre- and post-conditions of the function are. In particular, you should document things such as whether a function expects a certain pen position or color setting or if this function will change the pen or font in some way.

- Be sure to make use of the **`#define`**-d constants we provide as well as defining your own where needed. You should strive to avoid "magic numbers"— i.e. unnamed number constants floating around in the code. By naming a constant, you make it more clear what the purpose of the value is and make it easier to modify those values in the future. In fact, you should test your program with different values of these constants to make sure that everything continues to work.

- Unify similar passages where you can. For example, consider tasks such as filling a rectangle, clearing the window, choosing a random color, drawing then erasing a block, or drawing a centered string. You need these operations in several places, so be sure to unify the common requirements into one function that that can be re-used, rather than copy and pasting code. Try to be general in designing your functions—i.e. a box-drawing function that takes parameters for width and height is infinitely preferable that one that can only draw a box of exactly one size.

- Getting the program to work is only half the battle here. **We're looking for a well-crafted program with good decomposition, appropriate identifier names, and intelligent comments.** Your program should be easy to read and stylistically elegant. You should take care to avoid any unnecessary code repetition—if you need something more than once, write one function and call it from both places. We will frown heavily on long functions that attempt to do everything at once. Decomposing tasks into smaller subtasks is an essential step toward managing more complex programs, this is a skill you should strive to master right away. If in doubt about issues of engineering style, please don’t hesitate to seek advice!

- A little Macintosh detail: if the Quilt Demo or your program only shows up in black and white on a color monitor, this is probably because the monitor has been set to "thousands of colors" which requires more memory than we have allocated for your project. Set the monitor to 256 colors using the "Monitors" control panel and that should fix it.

Getting Started

The class web site **`http://www.stanford.edu/class/cs106x/`** is the place to find starter files for the assignments. For Assignment #1, there is only one source file `quilt.c`, a starting main file with a handy list of constants and two functions to set up some colors for you. We've also provided a compiled solution called "Quilt Demo" for you to check out.

Mac users will find a link to a self-extracting binhex archive. Your web browser may automatically unpack the folder when you download it—if not, you will need to do this manually. PC users will find a link to a .zip file. Please contact your local system administrator, RCC, computer consultant, etc. for help if needed.

The compilers to be supported this quarter are:

Mac:	Metrowerks CodeWarrior Pro Release 5
PC:	Microsoft Visual C++ Version 6

You will need to download and install some libraries before you start; see the Metrowerks and Visual C++ handouts for details.

Deliverables

Assignments are due at the beginning of lecture. Friday in class, you should turn a manila envelope containing a printout of your code (quilt.c) and a floppy disk containing your entire project. <u>Everything should be clearly marked with your name, CS106X and your section leader's name!</u>

The average person's lifetime includes six years eating, four years cleaning, two years trying to return telephone calls to people who never seem to be in, six months at stoplights, one year searching for misplaced objects, and eight months opening junk mail. So spending a night working on this assignment doesn't seem like so much, now does it?

```
/*
 * File: graphics.h
 * Version: 1.0
 * Last modified on Mon Jun  6 11:03:27 1994 by eroberts
 * -----------------------------------------------------
 * This interface provides access to a simple library of
 * functions that make it possible to draw lines and arcs
 * on the screen.  This interface presents a portable
 * abstraction that can be used with a variety of window
 * systems implemented on different hardware platforms.
 */

#ifndef _graphics_h
#define _graphics_h

/*
 * Overview
 * --------
 * This library provides several functions for drawing lines
 * and circular arcs in a region of the screen that is
 * defined as the "graphics window."  Once drawn, these
 * lines and arcs stay in their position, which means that
 * the package can only be used for static pictures and not
 * for animation.
 *
 * Individual points within the window are specified by
 * giving their x and y coordinates.  These coordinates are
 * real numbers measured in inches, with the origin in the
 * lower left corner, as it is in traditional mathematics.
 *
 * The calls available in the package are listed below.  More
 * complete descriptions are included with each function
 * description.
 *
 *   InitGraphics();
 *   MovePen(x, y);
 *   DrawLine(dx, dy);
 *   DrawArc(r, start, sweep);
 *   width = GetWindowWidth();
 *   height = GetWindowHeight();
 *   x = GetCurrentX();
 *   y = GetCurrentY();
 */

/*
 * Function: InitGraphics
 * Usage: InitGraphics();
 * ----------------------
 * This procedure creates the graphics window on the screen.
 * The call to InitGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output.  In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics(void);
```

```
/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * ---------------------
 * This procedure moves the current point to the position
 * (x, y), without drawing a line.  The model is that of
 * the pen being lifted off the graphics window surface and
 * then moved to its new position.
 */

void MovePen(double x, double y);

/*
 * Function: DrawLine
 * Usage: DrawLine(dx, dy);
 * ------------------------
 * This procedure draws a line extending from the current
 * point by moving the pen dx inches in the x direction
 * and dy inches in the y direction.  The final position
 * becomes the new current point.
 */

void DrawLine(double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point.  The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle.  This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on.  The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees.  If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle.  If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point.  If sweep is negative, the arc is drawn
 * clockwise from the current point.  The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
 *
 * Examples:
 *   DrawArc(r, 0, 360)    Draws a circle to the left of the
 *                         current point.
 *   DrawArc(r, 90, 180)   Draws the left half of a semicircle
 *                         starting from the 12 o'clock position.
 *   DrawArc(r, 0, 90)     Draws a quarter circle from the 3
 *                         o'clock to the 12 o'clock position.
 *   DrawArc(r, 0, -90)    Draws a quarter circle from the 3
 *                         o'clock to the 6 o'clock position.
 *   DrawArc(r, -90, -90)  Draws a quarter circle from the 6
 *                         o'clock to the 9 o'clock position.
 */
void DrawArc(double r, double start, double sweep);
```

```
/*
 * Functions: GetWindowWidth, GetWindowHeight
 * Usage: width = GetWindowWidth();
 *        height = GetWindowHeight();
 * ------------------------------------------
 * These functions return the width and height of the graphics
 * window, in inches.
 */

double GetWindowWidth(void);
double GetWindowHeight(void);

/*
 * Functions: GetCurrentX, GetCurrentY
 * Usage: x = GetCurrentX();
 *        y = GetCurrentY();
 * -----------------------------------
 * These functions return the current x and y positions.
 */

double GetCurrentX(void);
double GetCurrentY(void);

#endif
```

```
/*
 * File: extgraph.h
 * Version: 3.0
 * Last modified on Tue Oct  4 11:24:41 1994 by eroberts
 * -----------------------------------------------------
 * This interface is the extended graphics interface.
 * It includes all of the facilities in graphics.h, plus
 * several additional functions that are designed to
 * support more sophisticated, interactive graphics.
 */
#ifndef _extgraph_h
#define _extgraph_h
#include "genlib.h"

/* Section 1 -- Basic functions from graphics.h */
#include "graphics.h"

/* Section 2 -- Elliptical arcs */

/*
 * Function: DrawEllipticalArc
 * Usage: DrawEllipticalArc(rx, ry, start, sweep);
 * -----------------------------------------------
 * This procedure draws an elliptical arc.  It is exactly
 * the same in its operation as DrawArc in the graphics.h
 * interface, except that the radius is different along the
 * two axes.
 */

void DrawEllipticalArc(double rx, double ry,
                       double start, double sweep);

/* Section 3 -- Graphical regions*/

/*
 * Functions: StartFilledRegion, EndFilledRegion
 * Usage: StartFilledRegion(density);
 *        . . . other calls . . .
 *        EndFilledRegion();
 * ------------------------------
 * These calls make it possible to draw filled shapes on the
 * display.  After calling StartFilledRegion, any calls to
 * DrawLine and DrawArc are used to create a shape definition
 * and do not appear on the screen until EndFilledRegion is
 * called.  The lines and arcs must be consecutive, in the
 * sense that each new element must start where the last
 * one ended.  MovePen calls may occur at the beginning
 * or the end of the region, but not in the interior. When
 * EndFilledRegion is called, the entire region appears on the
 * screen, with its interior filled in.  The density parameter
 * is a number between 0 and 1 and indicates how the dot density
 * to be used for the fill pattern.  If density is 1, the shape
 * will be filled in a solid color; if it is 0, the fill will be
 * invisible.  In between, the implementation will use a dot
 * pattern that colors some of the screen dots but not others.
 */

void StartFilledRegion(double density);
void EndFilledRegion(void);
```

```
/* Section 4 -- String functions */

/*
 * Function: DrawTextString
 * Usage: DrawTextString(text);
 * ----------------------------
 * This function displays the string text at the current point
 * in the current font and size.  The current point is updated
 * so that the next DrawTextString command would continue from
 * the next character position.  The string may not include the
 * newline character.
 */

void DrawTextString(string text);

/*
 * Function: TextStringWidth
 * Usage: w = TextStringWidth(text);
 * ---------------------------------
 * This function returns the width of the text string if displayed
 * at the current font and size.
 */

double TextStringWidth(string text);

/*
 * Function: SetFont
 * Usage: SetFont(font);
 * ---------------------
 * This function sets a new font according to the font string,
 * which is case-independent.  Different systems support different
 * fonts, although common ones like "Times" and "Courier" are often
 * supported.  Initially, the font is set to "Default" which is
 * always supported, although the underlying font is system
 * dependent.  If the font name is unrecognized, no error is
 * generated, and the font remains unchanged.  If you need to
 * detect this condition, you can call GetFont to see if the
 * change took effect.  By not generating an error in this case,
 * programs become more portable.
 */

void SetFont(string font);

/*
 * Function: GetFont
 * Usage: font = GetFont();
 * ------------------------
 * This function returns the current font name as a string.
 */

string GetFont(void);
```

```
/*
 * Function: SetPointSize
 * Usage: SetPointSize(size);
 * -------------------------
 * This function sets a new point size.  If the point size is
 * not supported for a particular font, the closest existing
 * size is selected.
 */

void SetPointSize(int size);

/*
 * Function: GetPointSize
 * Usage: size = GetPointSize();
 * ----------------------------
 * This function returns the current point size.
 */

int GetPointSize(void);

/*
 * Text style constants
 * --------------------
 * The constants Bold and Italic are used in the SetStyle
 * command to specify the desired text style.  They may also
 * be used in combination by adding these constants together,
 * as in Bold + Italic.  The constant Normal indicates the
 * default style.
 */

#define Normal  0
#define Bold    1
#define Italic  2

/*
 * Function: SetStyle
 * Usage: SetStyle(style);
 * -----------------------
 * This function establishes the current style properties
 * for text based on the parameter style, which is an integer
 * representing the sum of any of the text style constants.
 */

void SetStyle(int style);

/*
 * Function: GetStyle
 * Usage: style = GetStyle();
 * -------------------------
 * This function returns the current style.
 */

int GetStyle(void);
```

```
/*
 * Functions: GetFontAscent, GetFontDescent, GetFontHeight
 * Usage: ascent = GetFontAscent();
 *        descent = GetFontDescent();
 *        height = GetFontHeight();
 * -------------------------------------------------------
 * These functions return properties of the current font that are
 * used to calculate how to position text vertically on the page.
 * The ascent of a font is the distance from the baseline to the
 * top of the largest character; the descent is the maximum
 * distance any character extends below the baseline.  The height
 * is the total distance between two lines of text, including the
 * interline space (which is called leading).
 *
 * Examples:
 *   To change the value of y so that it indicates the next text
 *   line, you need to execute
 *
 *        y -= GetFontHeight();
 *
 *   To center text vertically around the coordinate y, you need
 *   to start the pen at
 *
 *       y - GetFontAscent() / 2
 */

double GetFontAscent(void);
double GetFontDescent(void);
double GetFontHeight(void);

/* Section 5 -- Mouse support */

/*
 * Functions: GetMouseX, GetMouseY
 * Usage: x = GetMouseX();
 *        y = GetMouseY();
 * -------------------------------
 * These functions return the x and y coordinates of the mouse,
 * respectively.  The coordinate values are real numbers measured
 * in inches from the origin and therefore match the drawing
 * coordinates.
 */

double GetMouseX(void);
double GetMouseY(void);

/*
 * Functions: MouseButtonIsDown
 * Usage: if (MouseButtonIsDown()) . . .
 * -------------------------------------
 * This function returns TRUE if the mouse button is currently
 * down.  For maximum compatibility among implementations, the
 * mouse is assumed to have one button.  If the mouse has more
 * than one button, this function returns TRUE if any button
 * is down.
 */

bool MouseButtonIsDown(void);
```

```
/*
 * Functions: WaitForMouseDown, WaitForMouseUp
 * Usage: WaitForMouseDown();
 *        WaitForMouseUp();
 * -------------------------------------------
 * The WaitForMouseDown function waits until the mouse button
 * is pressed and then returns.  WaitForMouseUp waits for the
 * button to be released.
 */

void WaitForMouseDown(void);
void WaitForMouseUp(void);

/* Section 6 -- Color support */

/*
 * Function: HasColor
 * Usage: if (HasColor()) . . .
 * ----------------------------
 * This function returns TRUE if the graphics window can display a
 * color image.  Note that this condition is stronger than simply
 * checking whether a color display is available.  Because color
 * windows require more memory than black and white ones, this
 * function will return FALSE with a color screen if there is
 * not enough memory to store a colored image.  On the Macintosh,
 * for example, it is usually necessary to increase the partition
 * size to at least 1MB before color windows can be created.
 */

bool HasColor(void);

/*
 * Function: SetPenColor
 * Usage: SetPenColor(color);
 * --------------------------
 * This function sets the color of the pen used for any drawing,
 * including lines, text, and filled regions.  The color is a
 * string, which will ordinarily be one of the following
 * predefined color names:
 *    Black, Dark Gray, Gray, Light Gray, White,
 *    Red, Yellow, Green, Cyan, Blue, Magenta
 *
 * The first line corresponds to standard gray scales and the
 * second to the primary and secondary colors of light.  The
 * built-in set is limited to these colors because they are
 * likely to be the same on all hardware devices.  For finer
 * color control, you can use the DefineColor function to
 * create new color names as well.
 */

void SetPenColor(string color);

/*
 * Function: GetPenColor
 * Usage: color = GetPenColor();
 * -----------------------------
 * This function returns the current pen color as a string.
 */
string GetPenColor(void);
```

```
/*
 * Function: DefineColor
 * Usage: DefineColor(name, red, green, blue);
 * -------------------------------------------
 * This function allows the client to define a new color name
 * by supplying intensity levels for the colors red, green,
 * and blue, which are the primary colors of light.  The
 * color values are provided as real numbers between 0 and 1,
 * indicating the intensity of that color.  For example,
 * the predefined color Magenta has full intensity red and
 * blue but no green and is therefore defined as:
 *
 *      DefineColor("Magenta", 1, 0, 1);
 *
 * DefineColor allows you to create intermediate colors on
 * many displays, although the results vary significantly
 * depending on the hardware.  For example, the following
 * usually gives a reasonable approximation of brown:
 *
 *      DefineColor("Brown", .35, .20, .05);
 */

void DefineColor(string name,
                 double red, double green, double blue);

/* Section 7 -- Miscellaneous functions */

/*
 * Function: SetEraseMode
 * Usage: SetEraseMode(TRUE);
 *        SetEraseMode(FALSE);
 * --------------------------
 * The SetEraseMode function sets the value of the internal
 * erasing flag.  Setting this flag is similar to setting the
 * color to "White" in its effect but does not affect the
 * current color setting.  When erase mode is set to FALSE,
 * normal drawing is restored, using the current color.
 */

void SetEraseMode(bool mode);

/*
 * Function: GetEraseMode
 * Usage: mode = GetEraseMode();
 * ----------------------------
 * This function returns the current state of the erase mode flag.
 */

bool GetEraseMode(void);
```

```
/*
 * Function: SetWindowTitle
 * Usage: SetWindowTitle(title);
 * -----------------------------
 * This function sets the title of the graphics window, if such
 * an operation is possible on the display.  If it is not possible
 * for a particular implementation, the call is simply ignored.
 * This function may be called prior to the InitGraphics call to
 * set the initial name of the window.
 */

void SetWindowTitle(string title);

/*
 * Function: GetWindowTitle
 * Usage: title = GetWindowTitle();
 * --------------------------------
 * This function returns the title of the graphics window.  If the
 * implementation does not support titles, this call returns the
 * empty string.
 */

string GetWindowTitle(void);

/*
 * Function: UpdateDisplay
 * Usage: UpdateDisplay();
 * -----------------------
 * This function initiates an immediate update of the graphics
 * window and is necessary for animation.  Ordinarily, the
 * graphics window is updated only when the program waits for
 * user input.
 */

void UpdateDisplay(void);

/*
 * Function: Pause
 * Usage: Pause(seconds);
 * ----------------------
 * The Pause function updates the graphics window and then
 * pauses for the indicated number of seconds.  This function
 * is useful for animation where the motion would otherwise
 * be too fast.
 */

void Pause(double seconds);

/*
 * Function: ExitGraphics
 * Usage: ExitGraphics();
 * ----------------------
 * The ExitGraphics function closes the graphics window and
 * exits from the application without waiting for any additional
 * user interaction.
 */

void ExitGraphics(void);
```

```
/*
 * Functions: SaveGraphicsState, RestoreGraphicsState
 * Usage: SaveGraphicsState();
 *        . . . graphical operations . . .
 *        RestoreGraphicsState();
 * ---------------------------------------------------
 * The SaveGraphicsState function saves the current graphics
 * state (the current pen position, the font, the point size,
 * and the erase mode flag) internally, so that they can be
 * restored by the next RestoreGraphicsState call.  These two
 * functions must be used in pairs but may be nested to any depth.
 */

void SaveGraphicsState(void);
void RestoreGraphicsState(void);

/*
 * Functions: GetFullScreenWidth, GetFullScreenHeight
 * Usage: width = GetFullScreenWidth();
 *        height = GetFullScreenHeight();
 * --------------------------------------
 * These functions return the height and width of the entire
 * display screen, not the graphics window.  Their only
 * significant use is for applications that need to adjust
 * the size of the graphics window based on available screen
 * space.  These functions may be called before InitGraphics
 * has been called.
 */

double GetFullScreenWidth(void);
double GetFullScreenHeight(void);

/*
 * Functions: SetWindowSize
 * Usage: SetWindowSize(width, height);
 * ------------------------------------
 * This function sets the window size to the indicated dimensions,
 * if possible.  This function should be called before the graphics
 * window is created by InitGraphics.  Attempts to change the size
 * of an existing window are ignored by most implementations.  This
 * function should be used sparingly because it reduces the
 * portability of applications, particularly if the client
 * requests more space than is available on the screen.
 */

void SetWindowSize(double width, double height);
```

```
/*
 * Functions: GetXResolution, GetYResolution
 * Usage: xres = GetXResolution();
 *        yres = GetYResolution();
 * -----------------------------------------
 * These functions return the number of pixels per inch along
 * each of the coordinate directions and are useful for applications
 * in which it is important for short distances to be represented
 * uniformly in terms of dot spacing.  Even though the x and y
 * resolutions are the same for most displays, clients should
 * not rely on this property.
 *
 * Note: Lines in the graphics library are one pixel unit wide and
 * have a length that is always one pixel longer than you might
 * expect.  For example, the function call
 *
 *     DrawLine(2 / GetXResolution(), 0);
 *
 * draws a line from the current point to the point two pixels
 * further right, which results in a line of three pixels.
 */

double GetXResolution(void);
double GetYResolution(void);

#endif
```