C Programming:
Graphics Applications

An introduction to graphics
programming concepts in C

Jack Straub, Instructor
Version 1.0.1 DRAFT

Table of Contents

Table of Figures

Course Overview

C Programming: Graphics Applications is a ten week course, consisting of two hours per week lecture plus assigned reading, quizzes and homework projects. This is primarily a class in the C programming language, and introduces the student to basic graphics concepts such as pixel representation of figures, color models and windows.

Instructor

Jack Straub
9:00 a.m. - 5:00 p.m. Monday through Friday
206 - 562 - 4574
jstraub@aa.net

Objectives

Upon completion of this course, the student will have successfully demonstrated the following skills:

- The ability to write C programs containing simple graphical elements
- The ability to properly create C program modules from project specifications
- An understanding of advanced C programming concepts

Nonobjectives

It is not the intention of this class to impart to the student the following:

- Comprehensive graphics programming skills
- Windows programming

Credit Requirements

In order to receive credit for this course, the student must successfully meet the following criteria:

- 80% attendance
- Satisfactory completion of 3 quizzes (out of 5)
- Satisfactory completion of the class project

Text Books

Required

Computer Graphics, Second Edition
by Donald Hearn and M. Pauline Baker

Recommended

C A Reference Manual Fourth Edition, by S. Harbison and G. Steele

Programming in ANSI C, by Stephen Kochan

Computer Graphics Principles and Practice, by Foley, vanDam, Feiner and Hughs

Programming Windows 95, by Charles Petzold

Programming Windows 3.1, by Charles Petzold

X Window System : The Complete Reference to Xlib, . . ., by Robert Scheifler and James Gettys

X Window System Toolkit, by Paul Asente and Ralph Swick

Motif Programming Manual Volumes 6A and 6B, by Dan Heller and Paula Ferguson

How To Write Macintosh Software, by S. Knaster

Inside Macintosh: Macintosh Toolbox Essentials, by Apple Computer, Inc.

Course Outline

	Topics	Assigned Reading	Work Due
Week 1	*C Skills Summary*	Chapters 1 and 2	
Week 2	*Getting Started*	Chapter 3	Quizz 1
Week 3	*Basics*	Chapter 4	
Week 4	*Output Primitives*	Chapter 5	Quiz 2
Week 5	*Output Primitives, cont'd*	Chapter 11 (optional)	Quiz 3
Week 6	*Primitives and Attributes, Fonts*		Project, Part 1
Week 7	*Primitives in Applications*	Chapter 12 (optional)	Quiz 4
Week 8	*Introduction to Color*	Chapter 15	
Week 9	*Color Models*	Chapter 15	Quiz 5, Final Project
Week 10	*Color Topics*		

1. C Skills Summary

In this section we will review C skills that are particularly important to graphics and other large applications. We will concentrate on the following topics:

- Typedef
- Pointers
- Bit operators
- Dynamic Memory Allocation

1.1 Typedef

In most C projects, the *typedef facility* is used to create equivalence names for other C types, particularly for structures and pointers, but potentially for any type. Using typedef equivalence names is a good way to hide implementation details. It also makes your code more readable, and improves the overall portability of your product.

1.1.1 Typedef and Portability

Typedef is frequently used to improve code portability. To cite a simple example, suppose you had a need to declare a data structure that was guaranteed to occupy the same amount of memory on every platform. If this structure had integer fields you might be tempted to declare them to be either *short* or *long*, believing that these types would always translate to two or four byte integers, respectively. Unfortunately ANSI C makes no such guarantee. Specifically, if you declared a field to be type *long*, it would break when you tried to port your code to an Alpha running OSF-1, where an *int* is four bytes, and a *long* is eight bytes. To avoid this problem you would use typedef to declare equivalence types; on most platforms the equivalence type for a four byte field would be *long*, but under Alpha/OSF-1 it would be *int*:

```
#if defined( ALPHA ) && defined ( OSF_1 )
    typedef int BYTE4
#else
    typedef long BYTE4
#endif
```

For the record, a variable or field of that needed to be exactly four bytes would then be declared like this:

```
BYTE4  field_name;
```

1.1.2 Typedef and Structures

To create a structure variable suitable for describing a point location in a window, and to pass the variable to a routine that could draw it, you might use code like that shown in **Figure 1-1**:

```
struct cgr_point_s
{
    short xco;
    short yco;
};

void CGR_draw_point(
    struct cgr_point_s *point_info;
)

static void draw_a_point( void )
{
    struct cgr_point_s point;

    point.xco = 42;
    point.yco = 99;
    CGR_draw_point( &point );
}
```

Figure 1-1 Primitive Structure Declarations

Normally, however, you would declare equivalence names for the point data structure using typedef, and the above example would be written as shown in **Figure 1-2**:

```
typedef struct cgr_point_s
{
    short xco;
    short yco;
} CGR_point_t, *CGR_point_p_t;

void CGR_draw_point(
    CGR_point_p_t point_info
);

static void draw_a_point( void )
{
    CGR_point_t point;

    point.xco = 42;
    point.yco = 99;
    CGR_draw_point( &point );
}
```

Figure 1-2 Structure Declarations Using Typedef

Note that, in the above example, one typedef statement was used to create two equivalence names: *CGR_point_t*, which is equivalent to *struct cgr_point_s*; and *CGR_point_p_t*, which is equivalent to *(struct cgr_point_s *)*.

1.1.3 Typedef and Functions

Typedef can be used to declare an equivalence name for a function type, and, subsequently, the equivalence name can be used to declare a prototype for a function of that type. One use of this is to reduce repetition when declaring many functions of the same type; in X, for example, we often find that we have to declare tens or even hundreds of *callback functions*. Each of these functions is type *function with three parameters of type Widget, XtPointer, XtPointer that returns void*. A prototype for such a function would traditionally be declared like this:

```
static void cancel_cb(
    Widget    widget,
    XtPointer client_parm,
    XtPointer callback_parm
);
```

Instead, however, we might consider declaring an equivalence type for a callback function, then declaring each callback function prototype using the equivalence type. We could do that this way:

```
typedef void XT_CBPROC_t( Widget, XtPointer, XtPointer );

static XT_CB_PROC_t cancel_cb;
```

This technique can also be used to drastically simplify compound declarations. For example, what if we had to declare a field in a structure to be a pointer to a callback function? This is a frequent occurrence, and traditionally would be done this way:

```
typedef struct button_desc_s
{
    char *name;
    char *label;
    void (*button_cb)( Widget, XtPointer, XtPointer );
} BUTTON_DESC_t, *BUTTON_DESC_p_t;
```

This structure declaration becomes much easier to write (and to read) if you use the callback function equivalence type from the previous example, and extend it to embrace type *pointer to callback function*, as shown in **Figure 1-3**:

```
typedef void XT_CBPROC_t( Widget, XtPointer, XtPointer );
typedef XT_CBPROC_t *XT_CBPROC_p_t;

typedef struct button_desc_s
{
    char          *name;
    char          *label;
    XT_CBPROC_p_t button_cb;
} BUTTON_DESC_t, *BUTTON_DESC_p_t;
```

Figure 1-3 Typedef and Functions

The ultimate example of using typedef to simplify function prototypes might be to rewrite the prototype for the ANSI function *signal*. Signal is a function with two parameters; the first is type *int*, and the second is *pointer to function with one parameter of type int that returns void*; the return value of signal is *pointer to function with one parameter of type int that returns void*. The prototype for signal is usually declared like this:

```
void (*signal(
      int sig,
      void (*func)(int)))(int);
```

This prototype would be much easier to decipher if, as demonstrated in **Figure 1-4**, we just declared and used an equivalence for type *pointer to function with one parameter of type int that returns void*:

```
typedef void SIG_PROC_t( int );
typedef SIG_PROC_t *SIG_PROC_p_t;

SIG_PROC_p_t signal(
    int          sig,
    SIG_PROC_p_t func
);
```

Figure 1-4 Typedef and Signal

1.2 Pointers

Pointers are so well embedded in C programming styles that sometimes it seems that C programs do little more than pointer manipulation. You cannot be a successful C programmer without becoming thoroughly comfortable with pointers.

1.2.1 Pointers and Arrays

In C, pointers and arrays are very closely related. With only one exception, the name of an array is equivalent to a pointer to the first element of the array. In **Figure 1-5**, the first parameter of the process_vertices function is declared to be *pointer to int*; the actual call to process_vertices passes the name of an *array of type int* as the corresponding argument. Since a C subroutine can't determine the length, or *cardinality*, of an array passed as an argument, the second argument to process_vertices specifies the length of the array.

```
static void process_vertices( int *vertices, int num_vertices );

    int vertices[16] = { 150, 200, 300, 125, 400,  90, 220, 500,
                          40, 220,  40, 230, 250, 150, 110, 125
                       };
    . . .
    process_vertices( vertices, 16 );
```

Figure 1-5 Passing an Array to a Subroutine

1.2.2 Pointers and Structures

In C, we frequently need to take the address of a structure. One of the most common reasons for doing so is to pass a structure to a subroutine *by reference*. Sometimes we do this in order to allow the subroutine to return data in the structure; another reason is to prevent the structure from being passed by value, which can be inefficient and might cause a stack underflow or other system error. To take the address of a function use the *address-of* operator, &. In **Figure 1-6**, the function CGR_draw_point requires an argument passed to it to be type *pointer to structure*; the actual call to CGR_draw_point satisfies this requirement by starting with a structure variable, and computing a pointer to it by using the address-of operator.

```
typedef struct cgr_point_s
{
    int xco;
    int yco;
} CGR_point_t, *CGR_point_p_t;
void CGR_draw_point( CGR_point_p_t point );
. . .
    CGR_point_t base_point;
    . . .
    base_point.xco = 99;
    base_point.yco = 71;
    CGR_draw_point( &base_point );
```

Figure 1-6 Using Pointers to Structures

1.2.3 Pointers and Functions

Any moderately sized C program is likely to use pointers to functions at least occasionally. As we shall later see, it is impossible to write a graphics program in either Microsoft Windows or the X Window System without using pointers to functions.

Much like arrays, the name of a function without the function operator is treated as a pointer to the function. In **Figure 1-7**, we see a sample of code from an X Window System program. This code creates a push button which, when selected, will cause the program to be terminated. In programs of this type, X knows when a push button has been selected, then calls a function, called a *callback function*, whose address has been supplied by the programmer via a call to *XtAddCallback*. In this specific example, the function supplied by the programmer is one that will terminate the program. (Note: the prototype for XtAddCallback has been provided for clarity only; normally the programmer obtains this prototype by including Intrinsic.h in the source code.)

```
typedef void XT_CBPROC_t( Widget, XtPointer, XtPointer );
typedef XT_CBPROC_p_t *XT_CB_PROC_t;

void XtAddCallback(
    Widget        widget,
    String        callback_name,
    XT_CBPROC_p_t callback_proc,
    XtPointer     client_data
);
static XT_CBPROC_t quit_cb;

. . .
    Widget quit = NULL;
    . . .
    quit = XmCreatePushButton( parent, “quit”, args, num_args );
    XtAddCallback( quit, XmNactivateCallback, quit_cb, NULL );
    . . .

static void quit_cb( Widget    widget,
                     XtPointer client_data,
                     XtPointer call_data
                   )
{
    exit( EXIT_SUCCESS );
}
```

Figure 1-7 Passing a Pointer to a Function

1.3 Bit Operators

Bit operators are used to change the values of individual bits within an integer location. The value of a bit can be either 1 or 0. If a bit has a value of 1 we also say that it is *on* or *set*; if it is 0, we say that it is *off* or *reset*.

Before we can talk about changing the value of bit, we need a way to name each bit in an integer. **Figure 1-8** illustrates the two most common way to name a bit: by its *sequence number*, where the least significant bit is sequence number 0; and by its *value*, that is, the value represented by the bit when it is set.

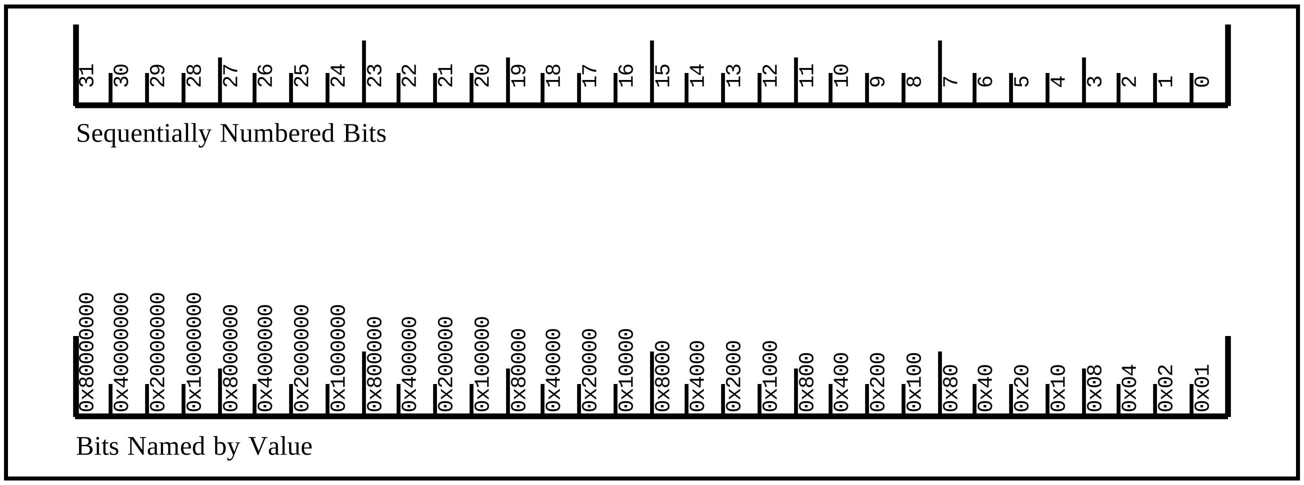


Figure 1-8 Bit Naming Conventions

C offers a variety of operators for testing and manipulating the value of a bit. This discussion is limited to the operators that allow you to perform the following functions:

- Setting a bit
- Resetting (clearing) a bit
- Toggling a bit
- Testing a bit

1.3.1 Setting a Bit

Bits are set using the *bitwise logical inclusive or* operator (|). As demonstrated in **Figure 1-9**, two integer values are or'd together by or'ing their corresponding bits in order to create a new integer value; if either or both bits in a pair are set, the corresponding bit is set in the result.

```
unsigned char val1   = 0x51,
              val2   = 0x21,
              result = 0;

result = val1 | val2;

    /* 01010001   val1, 0x51
       00100001   val2, 0x41
       --------
       01110001   result, 0x71
    */
```

Figure 1-9 The Bitwise Logical Inclusive Or Operator

In the example shown in **Figure 1-10**, the function CGR_create_window is used to create a window on a display. Like many other functions of this type in real graphical systems, the caller must configure a number of things about the window, such as its size and position, along with a great number of *Boolean* attributes, of which these represent just a few examples:

- Will the window be visible or invisible?
- Will the window be sensitive or insensitive?
- Will the window be used for input and output or output only?

Rather than define a separate parameter for each Boolean attribute, CGR_create_window defines a 32 bit parameter in which each bit represents an attribute. In the example, bit 0x40 is the bit that determines whether the window will be sensitive or insensitive; this bit is explicitly set in the *boolean_attributes* argument, thereby causing the newly created window to be sensitive.

```
typedef unsigned long  UINT32;
typedef unsigned short CGR_dimension_t;
typedef short          CGR_position_t;
CGR_window_t CGR_create_window(
    CGR_position_t  x_position,
    CGR_position_t  y_position,
    CGR_dimension_t width,
    CGR_dimension_t height,
    UINT32          boolean_attributes
);

#define CGR_SENSITIVE (0x40)
. . .
    UINT32          window_attrs = 0;
    CGR_position_t  xco          = 0;
    CGR_position_t  yco          = 0;
    CGR_dimension_t width        = 100;
    CGR_dimension_t height       = 100;
    CGR_window_t    window       = CGR_NULL_WINDOW;
    . . .
    window_attr |= CGR_SENSITIVE;
    window = CGR_create_window( xco, yco, width, height, window_attrs );
```

Figure 1-10 Setting Bits in an Integer Location

1.3.2 Clearing a Bit

To clear a bit in an integer location we use the *complement* operator (~) in conjunction with the *bitwise logical and* operator (&).

As illustrated by **Figure 1-11**, the complement operator is a unary operator that changes the state of every bit in an integer.

```
unsigned char val    = 0x53;
unsigned char result = 0;

result = ~val;

    /* 01010111   val, 0x53
       10101000   result, 0xA8
    */
```

Figure 1-11 The Complement Operator

As shown in **Figure 1-12**, two integer values are and'd together by and'ing their corresponding bits in order to create a new integer value; if both bits in a pair are set, the corresponding bit is set in the result.

```
unsigned char val1   = 0x63;
unsigned char val2   = 0xB1;
unsigned char result = 0;

result = val1 & val2;

    /* 01100111   val1, 0x63
       10110001   val2, 0xB1
       --------
       00100001   result, 0x21

    */
```

Figure 1-12 The Bitwise Logical And Operator

To clear a bit in an integer location, we first name the bit by its value, then complement it and bitwise-and it into the target location; this ensures that every bit set in the original value will remain set except for the named bit. This is shown in **Figure 1-13**, in which the 0x40 bit is explicitly cleared in the integer *target*.

```
#define TEST (0x40)
unsigned char target = 0x65;

target &= ~TEST;

    /* 01100101  original target, 0x65
       10111111  TEST complemented, 0xBF
       ----------
       00100101  new target, 0x25
    */
```

Figure 1-13 Using the Complement and Bitwise Logical And Operators Together

Figure 1-14 shows an example of the need to clear bits. A window which is sensitive needs to be made insensitive. This is accomplished by calling function CGR_change_window_attributes.

This function, however, requires that you pass all of a windows Boolean attributes at once; so first we obtain the window’s current attributes by calling CGR_get_window attributes, then we change just the sensitivity bit and call CGR_change_window_attributes.

```
static void set_window_insensitive( CGR_window_t window )
{
    UNIT32 window_attrs = 0;

    window_attrs = CGR_get_window_attributes( window );
    window_attrs &= ~CGR_SENSITIVE;
    CGR_change_window_attributes( window_attrs );
}
```

Figure 1-14 Clearing Bits in an Integer Location

1.3.3 Toggling a Bit

To toggle a bit in an integer location we can use the *bitwise logical exclusive or* operator (^). As shown in **Figure 1-15**, two integer values are xor’d together by xor’ing their corresponding bits in order to create a new integer value; if exactly one bit in a pair is set, the corresponding bit is set in the result.

```
unsigned char val1   = 0x55;
unsigned char val2   = 0x63;
unsigned char result = 0;

result = val1 ^ val2;

    /*  01010101  val1, 0x55
        01100111  val2, 0x63
        --------
        00110010  result, 0x32
    */
```

Figure 1-15 The Bitwise Logical Exclusive Or Operator

Figure 1-16 shows how this operator can be used to toggle attributes. We have a window whose sensitivity is to be changed; if it is sensitive we want to make it insensitive, and if it is insensitive we want to make it sensitive. First we call CGR_get_window_attributes to get all of the window’s Boolean attributes; then we toggle just the sensitivity bit, and call CGR_change_window_attributes.

```
static void toggle_window_sensitivity( CGR_window_t window )
{
    UNIT32 window_attrs = 0;

    window_attrs = CGR_get_window_attributes( window );
    window_attrs ^= CGR_SENSITIVE;
    CGR_change_window_attributes( window_attrs );
}
```

Figure 1-16 Toggling Bits in an Integer Location

1.3.4 Testing Bits

Individual bits in an integer location can be tested by using the *bitwise logical and* operator (&), which was introduced above, in **Section 1.3.2, *Clearing a Bit***. An illustration of the need to test a bit is shown in **Figure 1-17**. The motivation is that we need to refresh the contents of a number of windows, which can be a resource intensive process. So before initiating a refresh, we first make sure that the window is visible. To do this we get the current attributes of the window by calling CGR_get_window_attributes, then we test the CGR_VISIBILITY bit; only if this bit is set do we execute the refresh operation.

```
#define CGR_VISIBILITY (0x80)
. . .
    UINT32 window_attrs = 0;
    . . .
    window_attrs = CGR_get_window_attributes( window );
    if ( window_attrs & CGR_VISIBILITY )
        refresh_window( window );
```

Figure 1-17 Testing Bits in an Integer Location

1.4 Dynamic Memory Allocation

Dynamic memory allocation in C is performed by using one of the standard library functions *malloc, calloc* or *realloc* (or a cover for one or more of these routines). Memory allocated by one of these routines must eventually be freed by calling the standard library function *free*. Dynamic memory allocation in a graphics application is no different than in any other application, except, perhaps, that programmers have to be more on guard for hidden memory allocations. It is a frequent occurrence for a graphics utility to dynamically allocate memory on behalf of a caller, then expect the caller to free the memory when it is no longer needed. The example in Error! Reference source not found shows a routine that gathers and returns data about vertices in a graphics object; the object may contain anywhere from a few to thousands of vertices. The utility dynamically allocates the memory needed to hold an array of vertices, fills in the array and returns a pointer to the array to the caller. Since the utility cannot know when this memory is no longer needed, the caller must remember to free the memory at the appropriate time.

```
CGR_point_p_t CGR_get_vertices( CGR_object_t object, int *num_vertices )
{
    CGR_point_p_t vertex_data = NULL;
    int           num_vx      = 0;
    int           inx         = 0;

    num_vx = CGR_get_num_vertices( object );
    if ( num_vx > 0 )
    {
        vertex_data = calloc( num_vx, sizeof(CGR_point_t) );
        for ( inx = 0 ; inx < num_vx ; ++inx )
            CGR_get_vertex( object, inx, &vertex_data[inx] );
    }
    *num_vertices = num_vx;
    return num_vertices;
}
```

Figure 1-18 Hidden Dynamic Memory Allocations

2. Getting Started

In this section we will examine a sample graphics program in detail. Three samples have been provided: an MS Windows program, an X program using the Motif widget set, and an X program using the Athena widget set. Each of these programs is complete, and can serve as the basis for developing a test program for your project. The Windows sample is most suited to MS Windows users. The Motif sample should be used by students using the University system and others who have access to Motif libraries. The Athena sample may be used by students who are using Linux, and who do not have the Motif libraries.

Whatever system you employ for completing your project, you should be able to find many similarities between each example.

2.1 Introduction

When we talk about a graphics application, we often mean an application that will be used to draw lines and curves, or display pictures. However, a graphics program may also be any program that has a *graphical user interface*, or *GUI*. All graphics programs tend to have several things in common:

1. They are *event driven*. This means that the flow of a program is determined by events such as a mouse button click or a key press on a keyboard.
2. They are usually *object oriented*. This is actually true of most well-written programs, but it is especially true of graphics programs where so many objects are easily identified. A few examples of such objects are windows, push buttons, lines and menus.
3. They typically depend on *client/server* implementations. In the context of a graphics program, a server may be a process that manages scarce resources such as the display or the keyboard. A client is any application that needs access to these resources; such clients must request use of a resource from the server, and the server may choose to deny the request, or grant the request in a modified form.
4. They are *not entirely under application control*. Most of the management of a graphics application is performed at the operating system or system library level. Actual application control (that is, control by the code that you write) is usually limited to initialization of the user interface, and response to events.

2.2 MS Windows Example

```
#include <stdio.h>
#include <string.h>
#include <windows.h>

 static void paint_hello_detail(
    HDC  hdc,
    HWND hwnd
);

 static long PASCAL WndProc(
    HWND hwnd,
    UINT msg,
    UINT wParam,
    LONG lParam
);
```

```
int PASCAL WinMain( HANDLE instance,
                    HANDLE prev_instance,
                    LPSTR  cmd_param,
                    int    cmd_show
                  )
{
    static char *app_name = "WinHello";

    HWND        hwnd;
    MSG         msg;
    WNDCLASS    wnd_class;

    if( !prev_instance )
    {
        wnd_class.style         = CS_HREDRAW | CS_VREDRAW;
        wnd_class.lpfnWndProc   = WndProc;
        wnd_class.cbClsExtra    = 0;
        wnd_class.cbWndExtra    = 0;
        wnd_class.hInstance     = instance;
        wnd_class.hIcon         = 0;
        wnd_class.hCursor       = LoadCursor( 0, IDC_ARROW );
        wnd_class.hbrBackground = GetStockObject( WHITE_BRUSH );
        wnd_class.lpszMenuName  = 0;
        wnd_class.lpszClassName = app_name;
        RegisterClass( &wnd_class );
    }
    hwnd = CreateWindow(
        app_name,                       // window class name
        "Uh, Hello? World?",            // window caption
        WS_OVERLAPPEDWINDOW,            // window style
        CW_USEDEFAULT,                  // initial X position
        CW_USEDEFAULT,                  // initial Y position
        CW_USEDEFAULT,                  // initial X size
        CW_USEDEFAULT,                  // initial Y size
        0,                              // parent window handle
        0,                              // window menu handle
        instance,                       // program inst. handle
        NULL                            // creation parameters
                       );

   ShowWindow( hwnd, cmd_show );
   UpdateWindow( hwnd );

   while( GetMessage( &msg, 0, 0, 0 ) )
   {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
   }

   return msg.wParam;
}
```

```
static long PASCAL WndProc( HWND hwnd,
                            UINT msg,
                            UINT wParam,
                            LONG lParam
                          )
{
    long        rcode = 0;
    HDC         hdc;
    PAINTSTRUCT paint;

    switch( msg )
    {
        case WM_PAINT:
            hdc = BeginPaint( hwnd, &paint );
            paint_hello_detail( hdc, hwnd );
            EndPaint( hwnd, &paint );
            break;

        case WM_RBUTTONUP:
            PostQuitMessage( 0 );
            break;

        case WM_DESTROY:
            PostQuitMessage( 0 );
            break;

        default:
            rcode = DefWindowProc( hwnd, msg, wParam, lParam );
   }

   return( rcode );
}

static void paint_hello_detail( HDC hdc, HWND hwnd )
{
    RECT        rect;
    HPEN        old_pen,
                new_pen;
    int         start_x     = 0,
                start_y     = 0;
    COLORREF    color       = RGB( 0, 0, 0 );

    GetClientRect( hwnd, &rect );
    start_x = (rect.right - SIDE) / 2;
    start_y = (rect.bottom - SIDE) / 2;
    new_pen = CreatePen( PS_SOLID, 4, color );
    old_pen = SelectObject( hdc, new_pen );
    Rectangle( hdc, start_x,
                    start_y,
                    start_x + SIDE,
                    start_y + SIDE
             );
    SelectObject( hdc, old_pen );
    DeleteObject( new_pen );
}
```

2.3 X/Motif Sample

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>

#define SIDE (50)

typedef void XT_CBPROC_t( Widget, XtPointer, XtPointer );

static XT_CBPROC_t expose_cb;
static XT_CBPROC_t input_cb;

static void draw_square(
    Widget widget
);

void make_test(
    Widget top
);
```

```
int main( argc, char **argv )
{
    XtAppContext context;
    Widget       top      = NULL;

    top = XtAppInitialize( &context,
                           "Test",
                           NULL,
                           0,
                           &argc,
                           argv,
                           NULL,
                           NULL,
                           0
                         );

    make_test( top );
    XtRealizeWidget( top );
    XtAppMainLoop( context );
}

static void make_test( Widget top )
{
    Widget draw    = NULL;
    Screen *screen = XtScreen( top );
    Pixel  black   = BlackPixelOfScreen( screen );

    draw = XmCreateDrawingArea( top, “drawWin”, NULL, 0 );
    XtVaSetValues( draw, XmNbackground, black, NULL );
    XtAddCallback( draw, XmNexposeCallback, expose_cb, NULL );
    XtAddCallback( draw, XmNinputCallback, input_cb, NULL );

    XtManageChild( draw );
}

static void expose_cb( Widget    widget,
                       XtPointer client_data,
                       XtPointer call
                     )
{
    draw_square( widget );
}

static void input_cb( Widget    widget,
                      XtPointer client_data,
                      XtPointer call_data
                    )
{
    XmDrawingAreaCallbackStruct *draw_data = call_data;

    if ( draw_data->event->type == ButtonPress )
        if ( draw_data->event->xbutton.button == 3 )
            exit( EXIT_SUCCESS );
}
```

```
static void draw_square( Widget widget )
{
    Dimension width    = 0,
              height   = 0;
    GC        draw_gc  = None;
    Window    window   = XtWindow( widget );
    Screen    *screen  = XtScreen( widget );
    XtGCMask  gc_mask  = GCForeground | GCLineWidth;
    Display   *display = XtDisplay( widget );
    XGCValues gc_values;

    gc_values.foreground = WhitePixelOfScreen( screen );
    gc_values.line_width = 4;
    draw_gc = XtGetGC( widget, gc_mask, &gc_values );

    XtVaGetValues( widget, XtNwidth,  &width,
                           XtNheight, &height,
                           NULL
                 );

    XClearWindow( display, window );

    XDrawRectangle( display,
                    window,
                    draw_gc,
                    (width - SIDE) / 2,
                    (height - SIDE) / 2,
                    SIDE,
                    SIDE
                  );
}
```

2.4 X/Athena Example

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Simple.h>

#define SIDE (50)

typedef void XT_EVENT_HANDLER_t( Widget     widget,
                                 XtPointer  client_data,
                                 XEvent     *event,
                                 Boolean    *continue_to_dispatch
                               );

static void draw_square(
    Widget widget
);

static void make_test(
    Widget top
);

static XT_EVENT_HANDLER_t canvas_event_handler;
```

```
int main( int argc, char **argv )
{
    XtAppContext context;
    Widget       top     = NULL;

    top = XtAppInitialize( &context,
                           "Test",
                           NULL,
                           0,
                           &argc,
                           argv,
                           NULL,
                           NULL,
                           0
                         );

    make_test( top );
    XtRealizeWidget( top );
    XtAppMainLoop( context );
}

static void make_test( Widget top )
{
    EventMask events = ExposureMask        |
                       ButtonReleaseMask   |
                       StructureNotifyMask;
    Widget simple = NULL;

    simple =
        XtCreateWidget( "Canvas", simpleWidgetClass, top, NULL, 0 );
    XtVaSetValues( simple, XtNx,      200,
                           XtNy,      200,
                           XtNwidth,  300,
                           XtNheight, 300,
                           NULL
                 );
    XtAddEventHandler( simple,
                       events,
                       False,
                       canvas_event_handler,
                       NULL
                     );
    XtManageChild( simple );
}
```

```
static void canvas_event_handler( Widget    canvas,
                                  XtPointer client,
                                  XEvent    *event,
                                  Boolean   *dispatch
                                )
{
    switch ( event->xany.type )
    {
        case Expose:
        case ConfigureNotify:
            draw_square( canvas );
            break;

        case ButtonRelease:
            if ( event->xbutton.button == 3 )
                exit( EXIT_SUCCESS );
            break;

       default:
           break;
    }
}

static void draw_square( Widget widget )
{
    Dimension width    = 0,
              height   = 0;
    GC        draw_gc  = None;
    Window    window   = XtWindow( widget );
    Screen    *screen  = XtScreen( widget );
    XtGCMask  gc_mask  = GCForeground | GCLineWidth;
    Display   *display = XtDisplay( widget );
    XGCValues gc_values;

    gc_values.foreground = BlackPixelOfScreen( screen );
    gc_values.line_width = 4;
    draw_gc = XtGetGC( widget, gc_mask, &gc_values );

    XtVaGetValues( widget, XtNwidth,  &width,
                           XtNheight, &height,
                           NULL
                 );

    XClearWindow( display, window );

    XDrawRectangle( display,
                    window,
                    draw_gc,
                    (width - SIDE) / 2,
                    (height - SIDE) / 2,
                    SIDE,
                    SIDE
                  );
}
```

3. Basics

In this section, we will look at some basic graphics concepts. We'll begin by examining some of the properties of input and output devices, then take a closer look at video devices and how they are used to display images.

3.1 Input and Output Devices

Input devices are employed by the user to enter data into an application. Output devices are used by an application to translate internal data into a form that humans can assimilate. Applications have used input and output devices since computers were invented, but graphics applications use such devices far more robustly than traditional applications, and, as a result, handle input and output far differently.

3.1.1 Traditional Forms of Input and Output

One of the most common output devices is the *display*, also called the *monitor*. As seen in **Figure 3-1**, traditional applications relied primarily on text output, and text display was principally a function of hardware processing. To display a character, the application transmitted the character's ASCII code to a *device output register* located in the display hardware. Logic within the hardware itself then determined the shape of the character and displayed it.

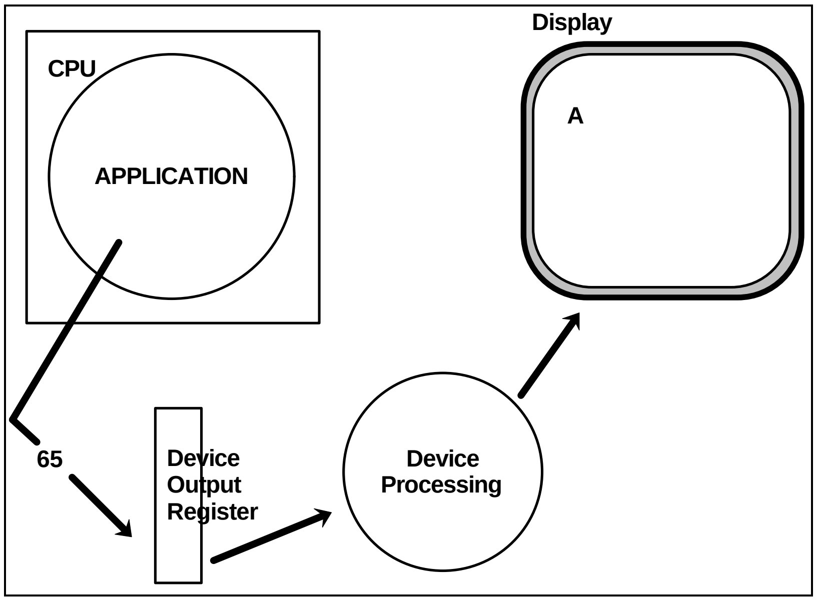


Figure 3-1 Traditional Output Device Processing

Likewise applications traditionally relied on text input from a *keyboard* device. Like traditional output devices, most keyboard processing was also done at the hardware level. As depicted in **Figure 3-2**, when the operator struck a key the hardware would translate the keystroke into the ASCII value of the associated character; this value would then be deposited in a *device input register* and then imported directly into the application.

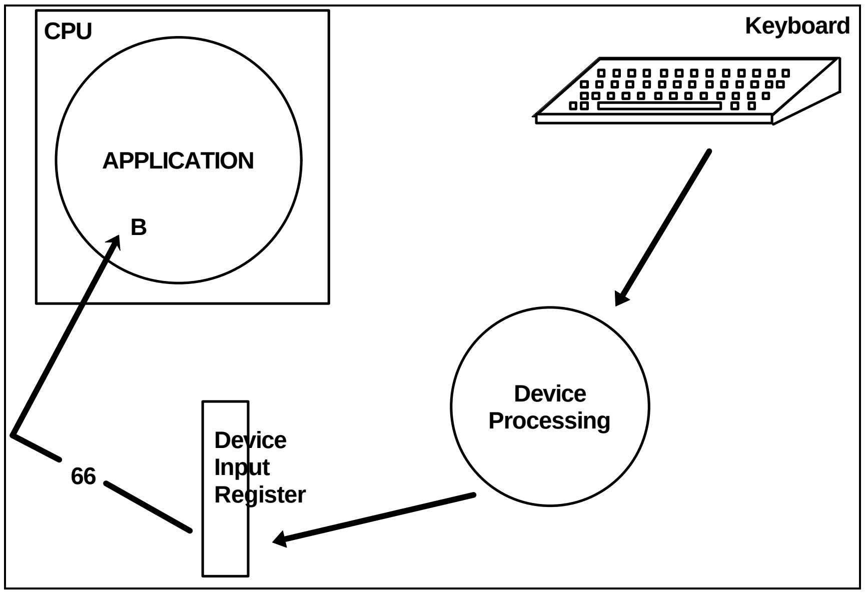


Figure 3-2 Traditional Input Device Processing

3.1.2 Graphical Input and Output

In a graphical system, most input and output is treated as graphics, even text. This provides applications with powerful flexibility, but at a cost in processing power. **Figure 3-3** shows how a graphics application is responsible for completely determining the shape of the characters (and other pictures) that it wishes to display.

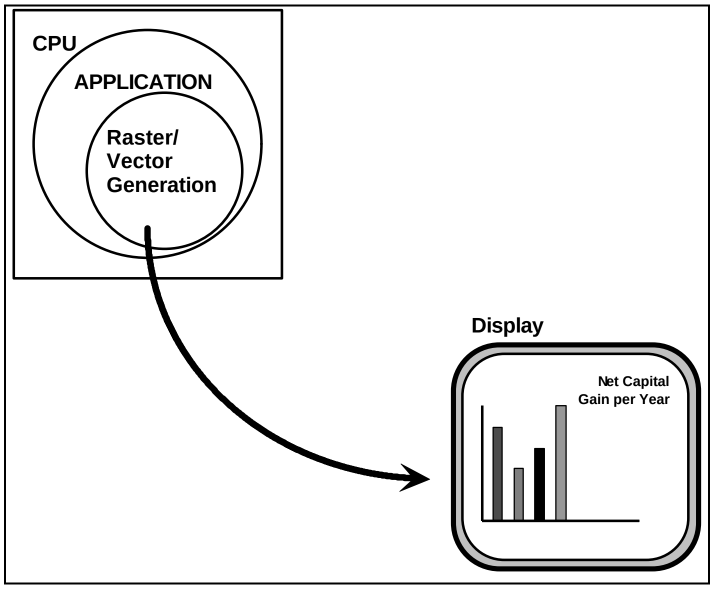


Figure 3-3 Graphical Output Device Processing

Similarly, as shown in **Figure 3-4,** keyboard input is no longer automatically converted to ASCII on the application's behalf; instead, the application is merely informed of the fact of a key press, the identifying number of the pressed key, and state information, such as whether the shift and/or control keys were depressed when the key was selected. It then becomes the application's responsibility to map the key identifier and state information to a particular character (if any).

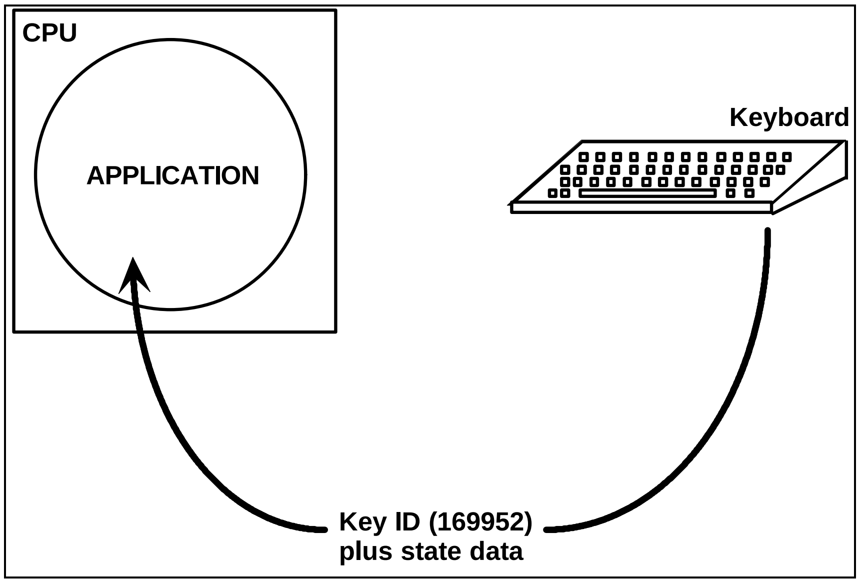
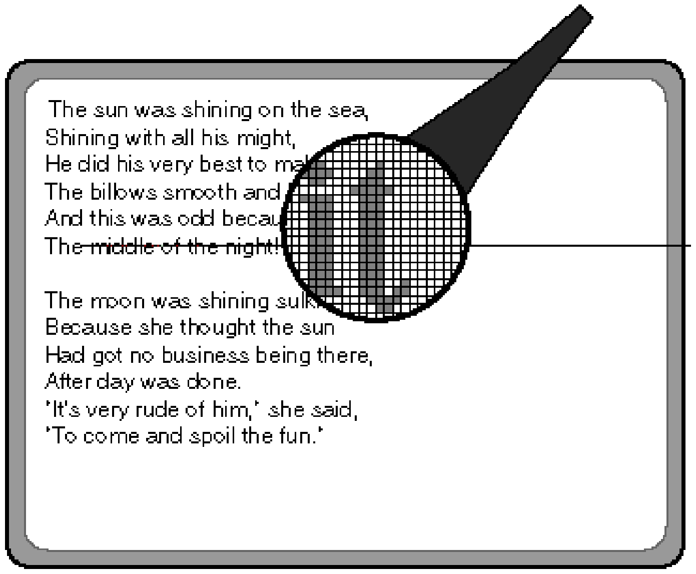


Figure 3-4 Graphical Input Device Processing

3.2 Picture Elements and Rasterization

Rasterization may be defined as the process of creating an image out of closely spaced dots, much like the formation of a mosaic. Most often this is the technique that a graphics application will use to display or print a picture; and since, as discussed in **Section 3.1, *Input and Output Devices***, graphics applications treat characters as pictures, it is also the technique used to display



The sun was shining on the sea,
Shining with all his might,
He did his very best to ma
The billows smooth and
And this was odd becau
The middle of the night!

The moon was shining sulk
Because she thought the sun
Had got no business being there,
After day was done.
'It's very rude of him,' she said,
'To come and spoil the fun.'

Figure 3-5 Rasters

text. As shown in **Figure 3-5**, close examination of the text on your monitor will reveal characters pieced together out of tiny rectangular units, called *picture elements*, or *pixels*.

The resolution of your monitor is determined by the number of pixels it contains. If you have a 1024 x 860 monitor, then you have 1024 horizontal *scan lines* each of which is divided into 860 pixels, for a total of 880,640 pixels. On a monochrome monitor, each of these pixels contains a

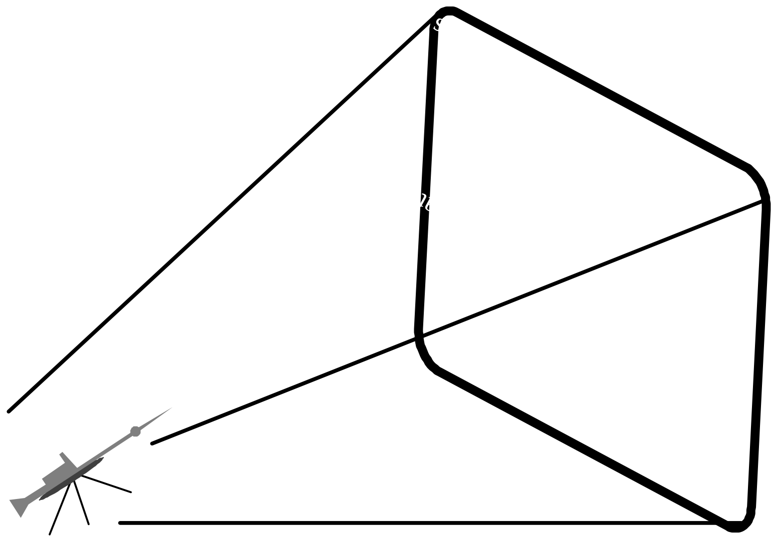


Figure 3-6 Pixels

photo-emitting dot, and rasters are created by "lighting up" some pixels, and leaving others dark. As depicted in **Figure 3-6**, an electron gun at the back of the monitor constantly scans the screen, firing electrons at those pixels which should be emitting light. The frequency with which the electron gun scans the screen is the monitor's *refresh rate*, typically 60 to 80 times per second.

As demonstrated by **Figure 3-7**, to produce color rasters, a color monitor has three electron guns, and each pixel on the screen of has three photo-emitting dots: one red, one green and one blue. Each electron gun is dedicated to one color, and on every scan each dot in each pixel is excited to some intensity. The combined intensities of the red, green and blue dots then combine to produce one of approximately 16 million different colors.

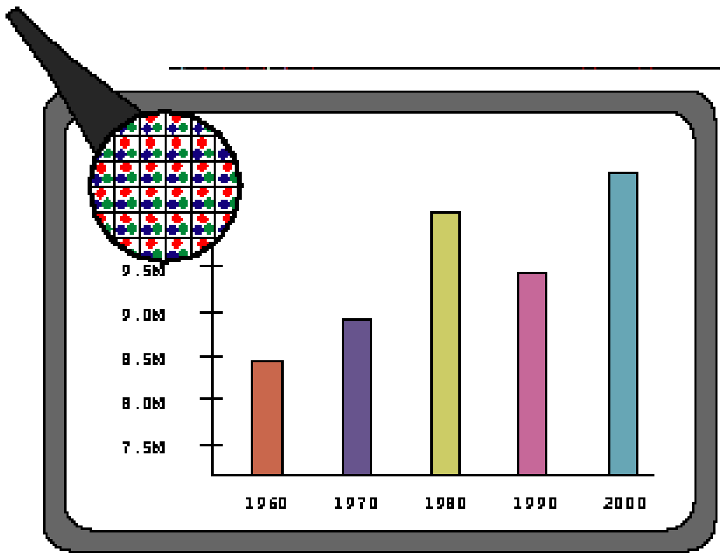


Figure 3-7 Color Monitors

The color of each pixel is controlled by data stored in a system's *frame buffer*. The frame buffer is a device which consists of logic circuits and memory. Like conventional memory, the memory

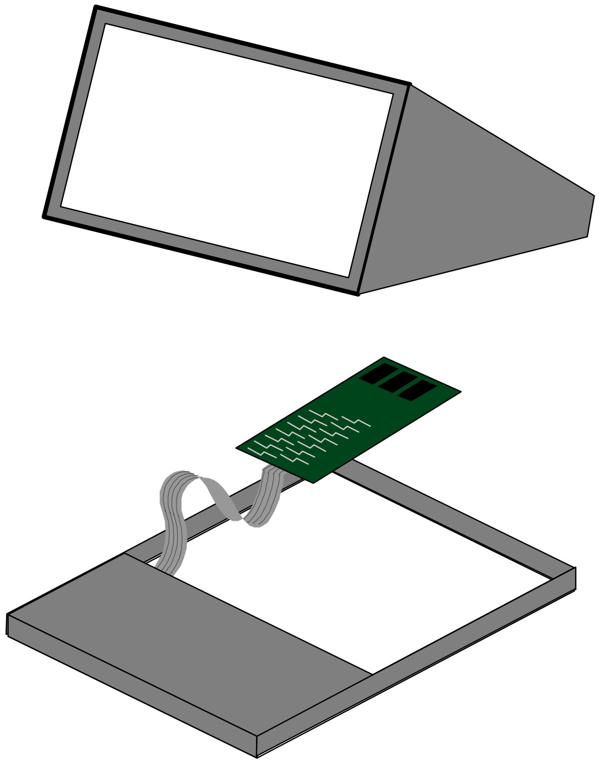


Figure 3-8 Frame Buffers

in the frame buffer is divided into units of sequential bits, one unit for each pixel on the monitor. The number stored in a memory location then determines the color of the associated pixel. Unlike conventional memory, a unit of memory in a frame buffer is not necessarily eight bits; for example, a unit of memory in a CGA device is four bits, and a unit of memory in an EGA device is six bits. The number of bits constituting a unit of memory is referred to as the frame buffer's *depth*. Since the number of values that can be stored in a memory location consisting of *n* bits is 2^n, the number of colors that can be displayed on a monitor is equal to 2^{depth}.

4. Output Primitives

In a graphics system, simple geometric objects are created using *output primitives*. Most systems limit their implementation of primitives to a very few objects, then the programmer or operator use the available primitives to construct more complex objects. Listed below are some common primitives implemented by various graphics systems; not all of them are implemented by all systems:

- Points
- Lines
- Rectangles
- Polygons
- Circular Arcs
- Elliptical Arcs
- Splines

In this section we will focus on primitive routines for drawing *points, lines, circular arcs, rectangles* and *polygons*. We will begin with a brief discussion on interrogating the physical capabilities of an output device.

4.1 Output Device Capabilities

It is often helpful, sometimes essential, to be able to determine the physical properties of an output device, such as the resolution of a monitor in pixels, or the width and height of a monitor in centimeters. Most graphics systems provide you with a way to obtain these properties.

In MS Windows, you first create an *information context (IC)* for a device. This is an opaque data structure that is created on your behalf, and owned by Windows; Windows gives you the id of the data structure, and you use the id to make calls to the operating system to obtain values from the IC. The code fragment in **Figure 4-1** shows how you might obtain the resolution, in pixels, of your monitor:

```
int x_pixels = 0,
    y_pixels = 0;

hdc_info = CreateIC( “DISPLAY”, NULL, NULL, NULL );
x_pixels = GetDeviceCaps( hdc_info, HORZRES );
y_pixels = GetDeviceCaps( hdc_info, VERTRES );
```

Figure 4-1 Interrogating Device Capabilities in MS Windows

X Windows allows you to obtain the same sort of data, but using a different technique. Instead of a single repository of data from which specific information may be extracted, X provides a different macro or function call for each desired value. The following code demonstrates the X Windows mechanism for obtaining the resolution of your monitor:

```
int x_pixels = 0,
    y_pixels = 0,
    screen   = 0;

screen = XDefaultScreenOfDisplay( display );
x_pixels = XDisplayWidth( display, screen );
y_pixels = XDisplayHeight( display, screen );
```

Figure 4-2 Interrogating Device Capabilities in X

4.2 Point Drawing Primitives

Every output primitive requires a starting point; many also specify an end point, and/or intermediate points. When the output device is a raster device, such as most monitors and laser printers, the output primitive will create a raster corresponding to the target object as a series of points. At the primitive level, a point in a raster is usually identified using a two-dimensional Cartesian coordinate system.

Most common monitors operate in raster mode, and most (but not all) use a Cartesian coordinate system in which the origin is in the upper left-hand corner of the monitor. As seen in **Figure 4-3**, the X axis of the coordinate system spans the width of the display, and X values increase from left to right. The Y axis spans the height of the display, and Y values increase from top to bottom.

At the physical level, the lengths of the X and Y axes are determined by the resolution of the display; that is, the width and height of the display, in pixels. However some graphics systems allow you the flexibility to impose a logical coordinate system on top of the physical. As seen in **Figure 4-4**, this might allow you the convenience of a logical system in which the X and Y coordinates were measured in different units, and in which the origin was somewhere other than

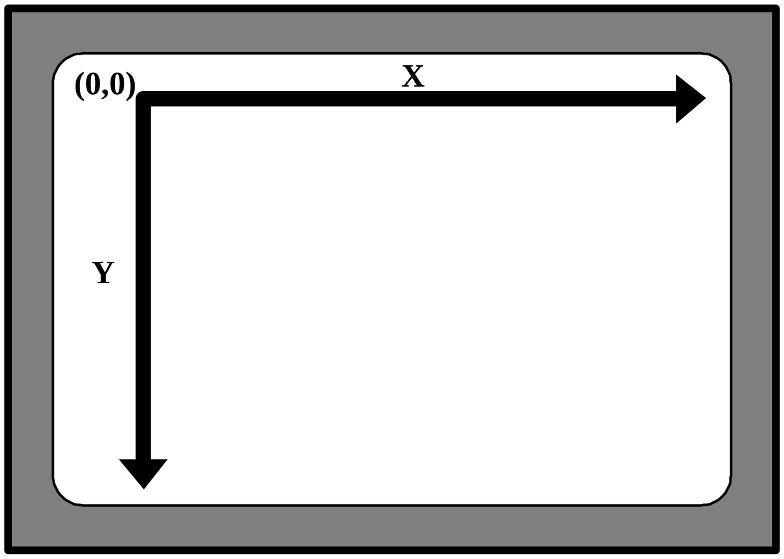


Figure 4-3 Cartesian Coordinates for Monitors

the upper left-hand corner of the display.

After setting up logical coordinates for a monitor (or a window on a monitor), the programmer passes logical point coordinates to the point drawing primitive, and the primitive converts each

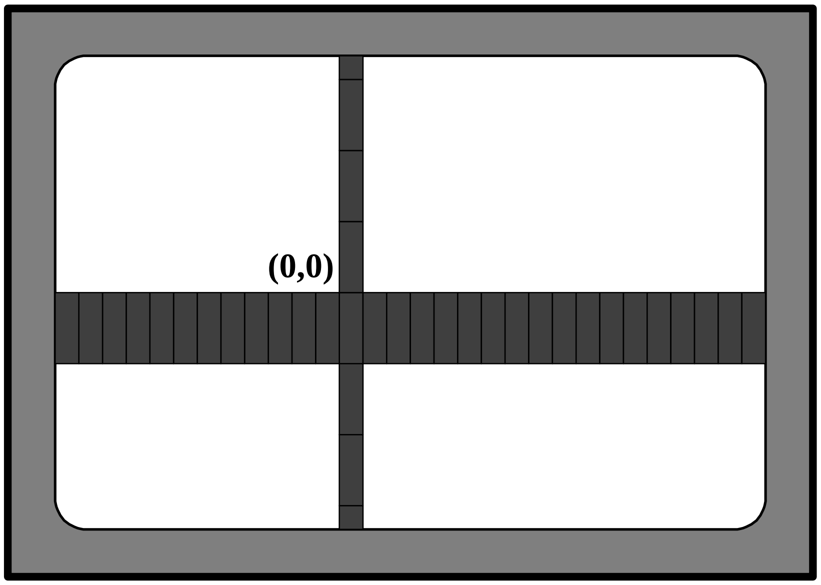


Figure 4-4 Logical Coordinates

logical specification to one or more pixels.

MS Windows provides the programmer with a facility for establishing a logical coordinate system; X does not. The MS Windows code fragment **Figure 4-5** creates a logical coordinate system for a window with a logical width and height of 1000 and 5000, respectively, and sets the origin to the center of the window.

```
SetMapMode( hdc, MM_ANISOTROPIC );
SetWindowExt( hdc, 1000, 5000 );
SetViewportExt( hdc, 1000, 5000 );
SetWindowOrg( hdc, 500, 2500 );
```

Figure 4-5 Establishing Logical Coordinates in MS Windows

To draw a point in a window in MS Windows use SetPixel(); to draw a point in X Windows, use XDrawPoint(), as shown in **Figure 4-6**:

```
SetPixel( hdc, 25, 100, 0L );

XDrawPoint( display, window, gc, 25, 100 );
```

Figure 4-6 Point Drawing Primitives

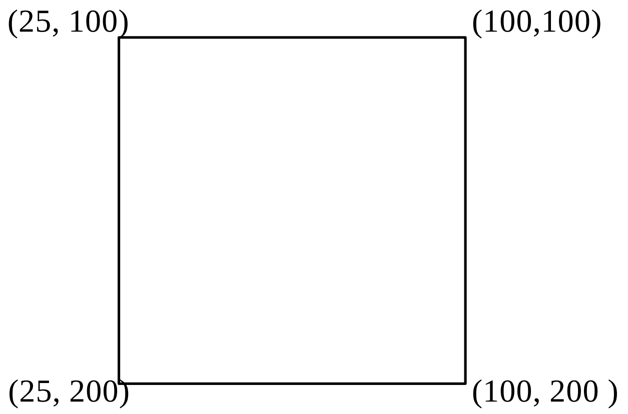
4.3 Line Drawing Primitives

Within an application there are many ways to represent a line:

- The end points of the line in *world coordinates*
- One end point, plus *slope* and *length*.
- Relative to another object.

Output primitives usually require the you specify the end points of a line in logical or physical pixel coordinates; if, within your application, you have specified a line using some more convenient representation, you will have to convert the representation before calling a primitive to draw the line.

The MS Windows line drawing primitive makes use of a concept called *the current cursor location*. From your application you set the current location to one end point of the target line by calling MoveTo. Next you call LineTo, passing the coordinates of the other end point of the line; Windows draws a line from the current cursor location to the coordinates passed to LineTo, then changes the current cursor location to those coordinates. **Figure 4-7** shows an example of drawing a rectangle using MoveTo and LineTo:



```
MoveTo( hdc,  25, 100 );
LineTo( hdc, 100, 100 );
LineTo( hdc, 100, 200 );
LineTo( hdc,  25, 200 );
LineTo( hdc,  25, 100 );
```

Figure 4-7 Drawing Lines in MS Windows

X Windows provides you with a number of ways to draw lines; **Figure 4-8** shows three different X primitives for drawing the above rectangle[1].

```
XDrawLine( display, window, gc,  25, 100, 100, 100 );
XDrawLine( display, window, gc, 100, 100, 100, 200 );
XDrawLine( display, window, gc, 100, 200,  25, 200 );
XDrawLine( display, window, gc,  25, 200,  25, 100 );
```
Method 1: XDrawLine

```
typedef struct
{
    short x;
    short y;
} XPoint;

XPoint points[5] =
    { { 25, 100 }, { 100, 100 }, { 100, 200 },
      { 25, 200 }, {  25, 100 }
    };
XDrawLines( display, window, gc, points, 5, CoordModeOrigin );
```
Method 2: XDrawLines

```
typedef struct
{
    short x1;
    short x2;
    short y1;
    short y2;
} XSegment;

XSegment lines[4] =
    { {  25, 100, 100, 100 }, {100, 100, 100, 200 },
      { 100, 200,  25, 200 }, { 25, 200,  25, 100 }
    };
XDrawSegments( display, window, gc, lines, 4 );
```
Method 3: XDrawSegments

Figure 4-8 X Windows Line Drawing Primitives

[1] The typedefs are included in the examples for clarity only. You would not normally declare these yourself; they are declared for you in Xlib.h.

4.4 Primitives for Drawing Circular Arcs

A *circular arc* is a segment of the edge of a circle. As depicted in **Figure 4-9**, to define an arc you must specify the following information:

- The circle along whose edge the arc resides
- The end points of the arc
- The direction of draw.

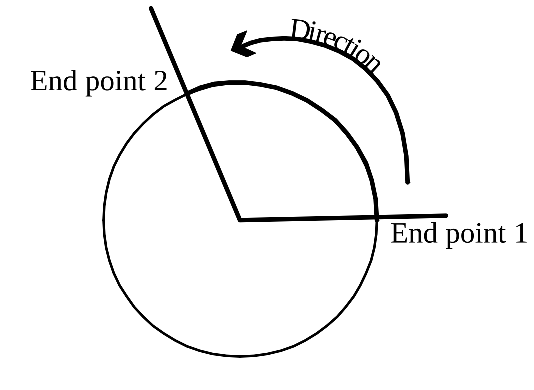


Figure 4-9 Defining an Arc

As seen in **Figure 4-10**, the circle that defines the arc is, itself, usually described by a *bounding box*, that is, a rectangle within which the target circle is inscribed; if the bounding box is not square, the arc may be said to be *elliptical*, though this is not usually true in a rigid, geometric sense. An end point of the arc is defined by a *radial line*,, that is, a line drawn from the center of the defining circle at an angle; the end point is placed where the line intersects the circle. There are two common ways to define a radial line:

1. By specifying the literal angle of the line.
2. By specifying a *radial point*. The radial line will be determined by drawing a line from the center of the circle through the radial point.

To draw an arc in MS Windows, use the Arc primitive. This function requires you to define the arc using a bounding box and two radial points; the bounding box is itself defined by specifying the box's upper-left and lower-right hand corners,. The direction of draw is determined by the

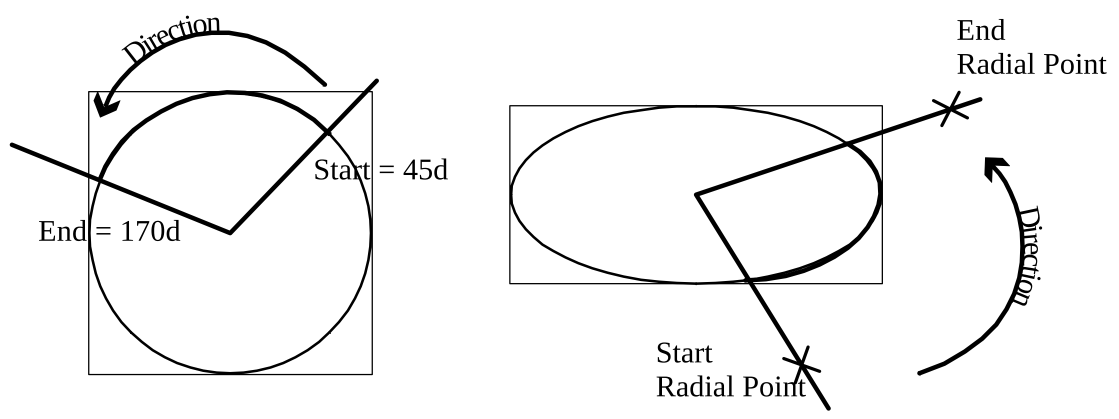


Figure 4-10 Defining an Arc (continued)

device context; the default is counter clockwise. Windows also provides the Ellipse primitive for drawing complete circles, which requires only a bounding box. **Figure 4-11** provides examples of the use of these functions.

```
Arc( hdc,
     25,     /* upper left x coordinate of bounding box  */
     100,    /* upper left y coordinate of bounding box  */
     125,    /* lower right x coordinate o f bounding box */
     200,    /* lower right y coordinate of bounding box */
     90,    /* x coordinate of start radial point       */
     50,    /* y coordinate of start radial point       */
     30,    /* x coordinate of end radial point         */
     70     /* y coordinate of end radial point         */
    );
Ellipse( hdc,
         25,       /* upper left x coordinate of bounding box  */
         100,      /* upper left y coordinate of bounding box  */
         125,      /* lower right x coordinate of bounding box *  /
         200       /* lower right y coordinate of bounding box */
       );
```

Figure 4-11 MS Windows Arc Drawing Primitives

To draw an arc in X, use the XDrawArc primitive. This routine requires a bounding box specified by providing the box's upper left-hand corner, plus its width and height. Start and end points are determined by angles specified in 64ths of a degree, using the *three o'clock rule*; that is if you think of the defining circles as a clock face, 0 degrees points to 3. Direction of draw is determined by the sign of the start angle; a positive sign draws counter clockwise, and a negative sign draws clockwise. To draw complete circles, X provides the XDrawCircle function, which requires only a bounding box for the circle.

```
XDrawArc( display, window, gc,
          25,       /* upper left x coordinate of bounding box  */
          100,      /* upper left y coordinate of bounding box  */
          125,      /* lower right x coordinate of bounding box */
          200       /* lower right y coordinate of bounding box */
          10 * 64, /* start angle = 10 degrees                */
          170 * 64 /* end angle = 170 degrees                 */
        );
XDrawCircle( display, window, gc,
             25,    /* upper left x coordinate of bounding box  */
             100,   /* upper left y coordinate of bounding box  */
             125,   /* lower right x coordinate of bounding box */
             200    /* lower right y coordinate of bounding box */
           );
```

Figure 4-12 X Arc Drawing Primitives

4.5 Primitives for Drawing Polygons

Polygons are closed shapes consisting of three or more connected lines, or edges. A polygon in which no edges intersect is called a *simple polygon*; a polygon in which one or more edges intersect is called a *self-intersecting polygon* (see **Figure 4-13**.)

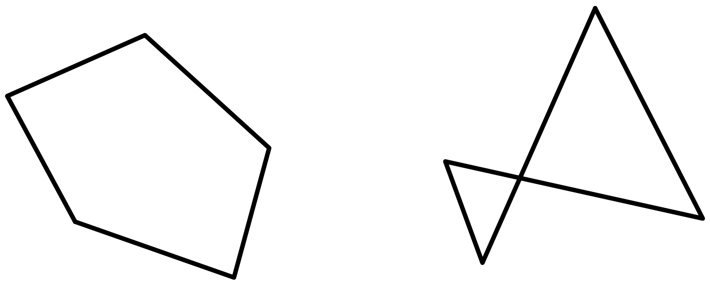


Figure 4-13 Polygons

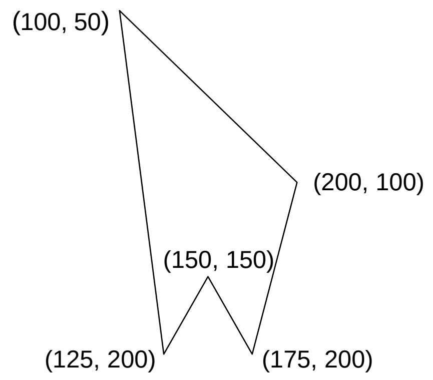
A *rectangle* is a kind of polygon that is used so frequently in graphics applications that most graphics packages devote a primitive to it. MS Windows provides the primitive *Rectangle*, which characterizes a rectangle using the rectangle's upper-left and lower-right hand corner coordinates. X provides the primitive *XDrawRectangle*, which characterizes a rectangle using the rectangle's upper-left hand corner coordinates, plus its width and height. The prototypes for these two functions are shown in **Figure 4-14**:

```
Rectangle( HDC hdc,
           int nLeftRect,   /* x coordinate, upper left corner  */
           int nTopRect,    /* y coordinate, upper left corner  */
           int nRightRect,  /* x coordinate, lower right corner */
           int nBottomRect  /* y coordinate, lower right corner */
         );

XDrawRectangle( Display *display, Drawable drawable, GC gc,
                int          x,      /* upper left x coordinate */
                short        y,      /* upper left y coordinate */
                unsigned int width,  /* width                   */
                unsigned int height  /* height                  */
              );
```

Figure 4-14 Rectangle Drawing Primitives

To draw a generalized polygon in MS Windows use *Polygon*. This primitive takes an array of vertices (x, y coordinates of points in a raster) and draws edges connecting them, closing the polygon if necessary, as seen in **Figure 4-15**:



```
typedef struct
{
    long x,
    long y,
} POINT;

POINT vertices[] =
{ { 100,  50 },
  { 200, 100 },
  { 175, 200 },
  { 150, 150 },
  { 125, 200 }
};

Polygon( hdc, vertices, 5
```

Figure 4-15 Polygon Drawing Primitive

MS Windows also provides a primitive for drawing multiple polygons. *PolyPolygon* takes two arrays: one array of vertices, and a second array of integers; each integer in the second array tells PolyPolygon how many sequential vertices in the first array should be used to draw each polygon. This is demonstrated in **Figure 4-16**:

```
POINT vertices[] =
{ { 200,  25 }, { 300, 125 }, { 225, 150 },
  {  75,  50 }, { 125, 100 }, {  75,  50 }, {  25, 100 },
  { 175, 175 }, { 225, 250 }, { 125, 250 }
};
int num_vert_vertices[] = { 3, 4, 3 };
PolyPolygon( hds, vertices, num_vertices );
```

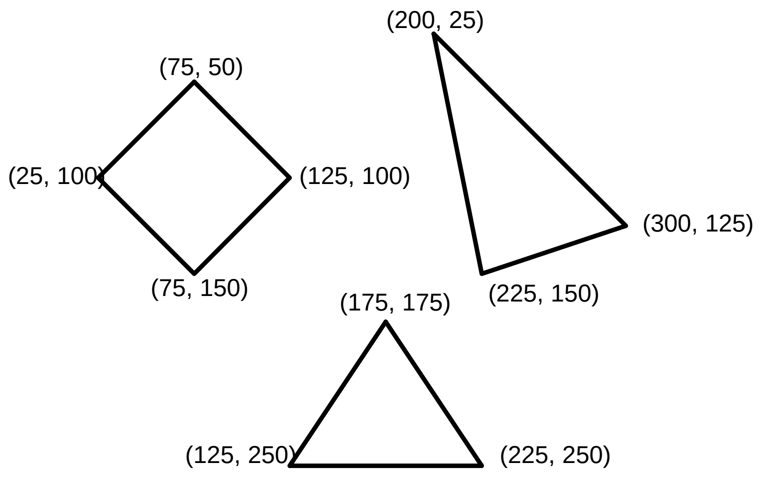


Figure 4-16 The PolyPolygon Primitive

4.6 Splines

Spline is a general term applied to a smooth curve, such as the one shown in **Figure 4-17**. Splines are usually constrained by endpoints, plus additional data that define the shape of the curve, such as:

- Tangent lines
- Polynomials
- Nonrational equations

MS Windows supplies two functions for drawing a type of spline called a *Bezier Curve*, based on cubic polynomials; they are *PolyBezier* and *PolyBezierTo*. X provides none.

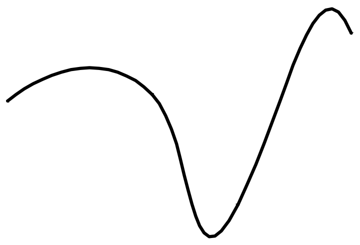


Figure 4-17 Splines

5. Primitives and Attributes

When creating a primitive figure you typically need to specify the properties of the figure, such as the color of a circle. These properties are called *attributes*, and the most common such attributes extend to characterizing the area and edges of the figure. In this section we will take a quick look at the common attributes of *areas* and *lines*.

5.1 Area Attributes

Graphics packages usually allow the area enclosed by a figure to be filled with a solid color or a pattern. In order to do so, the graphics package must be able to distinguish between the *interior* and *exterior* of the figure; when the figure is a simple, closed object, the distinction is unambiguous, as shown in **Figure 5-1**:

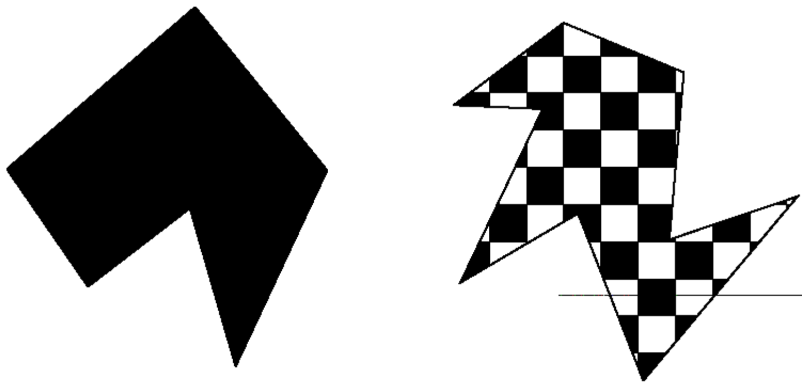


Figure 5-1 Filling Simple, Closed Figures

When a figure is not closed or not simple, determining the interior of the figure may be ambiguous, and you have to provide the drawing primitive with additional information that will allow the ambiguity to be resolved. For example, when filling an arc, you must specify whether the arc will be filled using the *arc-pie* or *arc-chord* fill rule. **Figure 5-2** demonstrates the difference between the two rules.

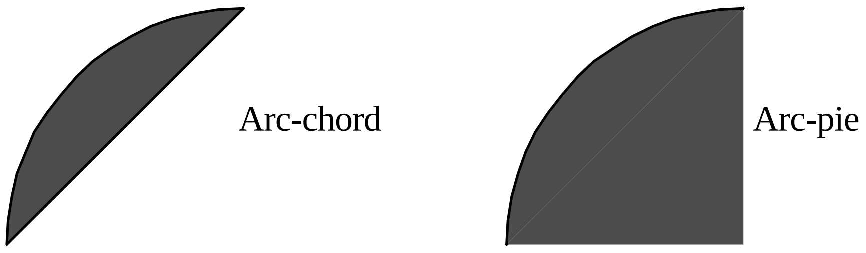


Figure 5-2 Filling an Arc

When a figure contains self-intersecting edges, you will have to specify the rule by which an interior point is distinguished from an exterior point. The two most common rules are:

- odd-even rule
- non-zero winding number rule

Figure 5-3 shows how the same figure filled using these two different rules can yield different results.

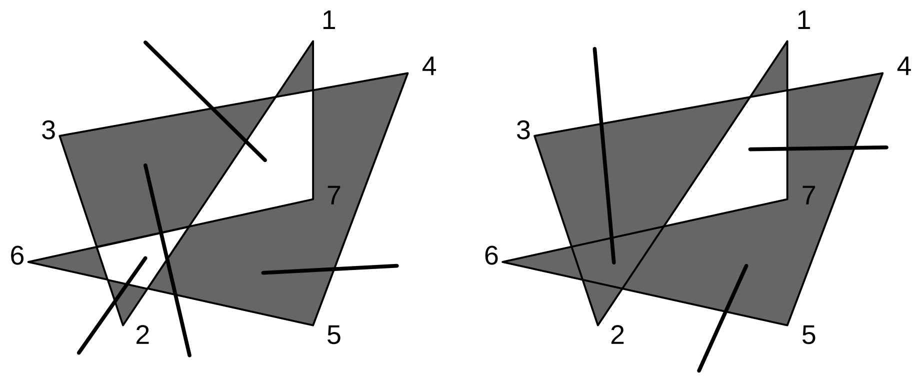


Figure 5-3 Polygon Fill Rules

In MS Windows, the closed figures drawn by Ellipse, Rectangle, Polygon and PolyPolygon are automatically filled; the fill color and pattern, and polygon fill rule (odd-even vs. non-zero winding number) are specified in the device context (HDC). To fill an arc in MS Windows, use *Chord* and *Pie* to fill the arc using the arc-chord and arc-pie rules, respectively; these two functions have the same prototype as Arc.

To fill figures in X, use functions such as *XFillArc, XFillRectangle* and *XFillPolygon*. The fill color and pattern, polygon fill rule and arc fill rule are specified in the graphics context (GC).

5.2 Line Attributes

Line attributes may apply to lines and edges of other primitives, such as arcs and polygons. The most common attributes characterized by graphics packages are:

- Line width
- Line style
- Line caps
- Join style

Line width determines the width of a line or edge; it is usually specified in pixels, but some drawing packages allow you the option of specifying other units, such as inches, centimeters or points. **Figure 5-4** shows lines of several different widths.

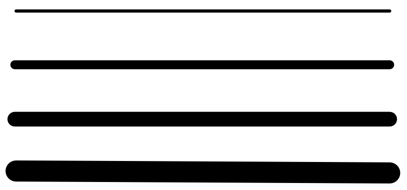


Figure 5-4 Line Width

Line style determines the pattern in which a line is drawn, such as *solid, dash, dash-dot* and *user defined*. **Figure 5-5** demonstrates a few examples.

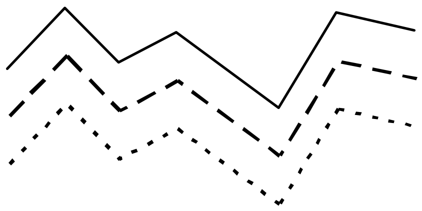


Figure 5-5 Line Styles

Line caps determine the size and shape of the end of a line; they are only meaningful for lines of greater than one pixel in width. As shown in **Figure 5-6**, typical caps include:

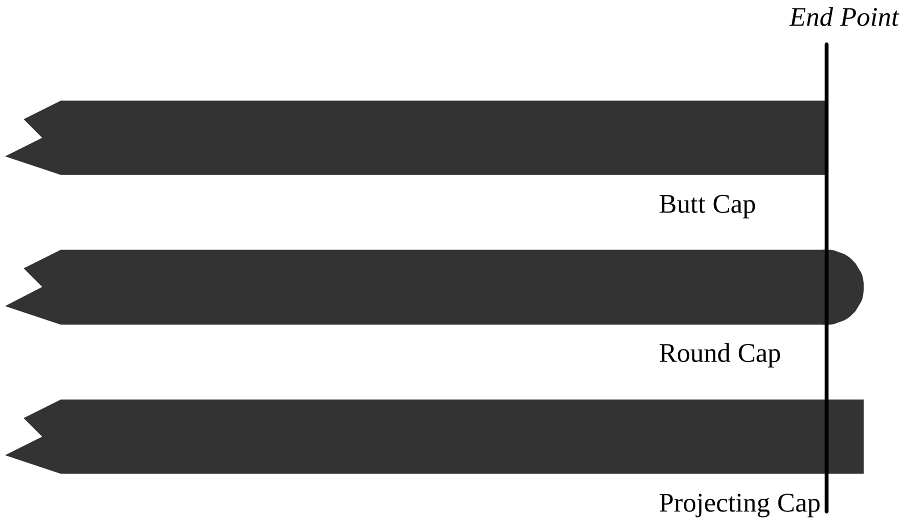


Figure 5-6 Line Caps

- *Butt cap*: the end of the line terminates abruptly at the indicated end point.
- *Round cap*: the end of the line terminates in a circular arc that extends beyond the indicated end point for a distance of one-half the width of the line.
- *Projecting cap*: the end of the line terminates in a square end that extends beyond the indicated end point for distance of one-half the width of the line.

Join style determines how two lines are joined at their end points. **Figure 5-7** shows three common join styles:

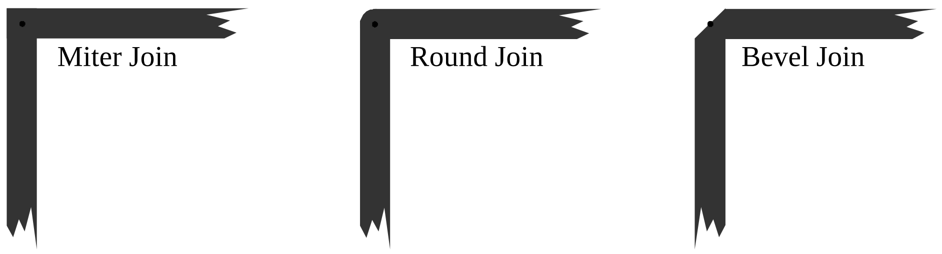


Figure 5-7 Line Join Styles

Not all graphics packages allow you to set all the attributes described above, nor is the list of attributes so far discussed exhaustive. MS Windows 3.1 allows only the *width* and *style* to be set; Win32 (Windows 95 and Windows NT) also allow *line caps* and *join style* to be set. In MS Windows, line attributes are determined by creating an object called a *pen* with the desired features. After creation, the pen is selected into a device context, then any edge drawn using that device context will have the attributes of the pen. **Figure 5-8** shows an MS Windows code fragment that creates a pen to draw black edges, 4 pixels in width, in a dash-dot pattern. To investigate the additional features in Win32, see the reference page for *ExtCreatePen* in your Win32 documentation.

```
COLORREF color   = RGB( 0, 0, 0 );
HPEN     new_pen,
         old_pen;
new_pen = CreatePen( PS_DASHDOT, 4, color );
old_pen = SelectObject( hdc, new_pen );
```

Figure 5-8 Selecting Line Attributes in MS Windows

X allows all the line attributes discussed above to be configured by adjusting values in the graphics context. **Figure 5-9** shows an X code fragment that configures a graphics context to draw dashed edges, four pixels in width, with round caps and a miter join style:

```
XSetLineAttributes( display,
                    gc,
                    4,
                    LineOnOffDash,
                    CapRound,
                    JoinMiter
                  );
```

Figure 5-9 Selecting Line Attributes in X

6. Fonts

A *font* is a graphical representation of a *character set*. In this section we will examine the two major classes of font representation, *bitmapped* and *outline*, and the following *font attributes*:

- Serif vs. Sans Serif
- Fixed vs. Proportional
- Size
- Weight
- Class
- Metrics

6.1 Font Representation

There are two common ways to define fonts: *bitmapped* fonts define a character in a font as a raster; *outline* fonts define a character in a font as a set of vectors. **Figure 6-1** shows an example of each.

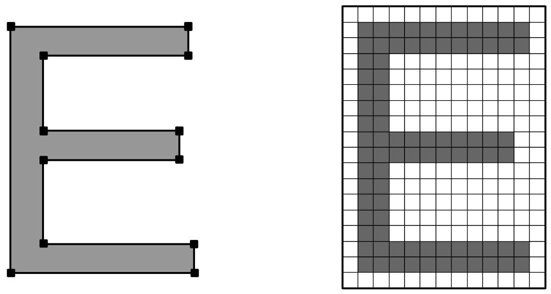


Figure 6-1 Font Representation: Outline and Bitmapped

Outline fonts provide character profiles which can easily be adapted to a variety of sizes, weights and slants. They are relatively new, and fairly efficient. Before a character in an outline font can be displayed, its outline has to be converted to a raster.

Each character in a bitmapped font has a hard-coded raster. Bitmapped fonts require a lot of storage space and are not as flexible as outline fonts, but many font designers prefer them because they provide finer artistic control over each character.

6.2 Serif vs. Sans Serif

Serif refers to the "decorations" (ascending and descending lines) that grace a character in a font. A font without such decorations is called *sans serif*. **Figure 6-2** provides a sample of each.

B B

Figure 6-2 Serif vs. Sans Serif Fonts

6.3 Fixed vs. Proportional

The relative width of a character in a font is determined by whether the font is *fixed* or *proportional*. Every character in a fixed font (also called a *nonproportional* or *monospaced* font) is the same width; a character in a proportional font, on the other hand, is only as wide as it needs to be. **Figure 6-3** shows an example of each.

Call me Ishmael
`Call me Ishmael`

Figure 6-3 Fixed vs. Proportional Fonts

Proportional fonts are more aesthetic than fixed fonts, but fixed fonts are more convenient when text needs to be aligned in columns, as in writing source code for a program.

6.4 Size, Weight and Slant

Size, usually measured in *points* (units equal to approximately 1/72 of an inch), gives you an idea of how large a font is. *Weight* is a relative measure that tells you how dark or *bold* a font is. *Slant* tells you how the font "bends in the wind"; there are five common slants:

- *Roman* is upright (no slant)
- *Italic* is slanted clockwise
- *Oblique* is slanted clockwise, usually less so than italic
- *Reverse Italic* is slanted counter clockwise
- *Reverse Oblique* is slanted counter clockwise, usually less so than reverse italic

Figure 6-4 shows several examples.

Eight point font
Sixteen point font, bold
Twenty four point font, italic

Figure 6-4 Font Size, Weight and Slant

6.5 Metrics

Font *metrics* describe the *geometry* of the characters in a font. As seen in Figure 6-5, each character in a font can be described by the following measurements:

- *Descent*, the height of the character below the *baseline*
- *Ascent*, the height of the character above the baseline
- *Left bearing*, the width of the character to the left of the *origin*
- *Right bearing*, the width of the character to the right of the origin
- *Width*, the distance from the origin of the character to the origin of the next character.

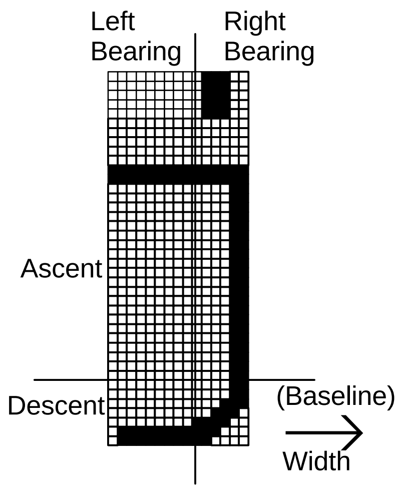


Figure 6-5 Font Metrics

In addition to the above measurements, for a given font you can usually find out the minimum, maximum and average width of a character, and the maximum ascent and descent of all the characters in the font.

6.6 Font Drawing Primitives

Graphics packages usually provide a wide variety of utilities for interrogating the attributes of a font, selecting a font to draw with, and drawing a character string. In X, a font is selected in the graphics context, and drawn using routines like *XDrawText* and *XDrawString*. In MS Windows, a font is selected in the device context, and drawn using routines such as *TextOut* and *DrawText*.

When working with font primitives, there are a few things important things to remember about fonts and character sets:

- A character set isn't restricted to 256 characters; two or more bytes may be needed to specify the ID of a character.
- "Font" and "character set" are not synonymous with "ASCII"; you can't assume that the ID of a character is the same as its a ASCII value, or even that it has an ASCII value.
- Since ASCII values don't apply to fonts and character sets, a particular character set may not have a NUL character, and strings are not guaranteed to be NUL-terminated.

7. Primitives in Applications

The last few section have introduced the concept of primitives for drawing text and simple geometric figures. In this section we will look at the application of primitives. First we'll see simple MS Windows and X programs that incorporate drawing primitives. Next will be an examination of the algorithm for calculating radial points for drawing arcs. The section will conclude with a subroutine for drawing lines that can be used to test the line drawing function of your project.

7.1 "Hello Again," MS Windows Example

As demonstrated by **Figure 7-1**, the first part of this example will draw chords in opposing corners of a window. The chords will lie on the edge of a circle of diameter 100, will have end points at 45^o and 225^o, and will be open to the center of the window.

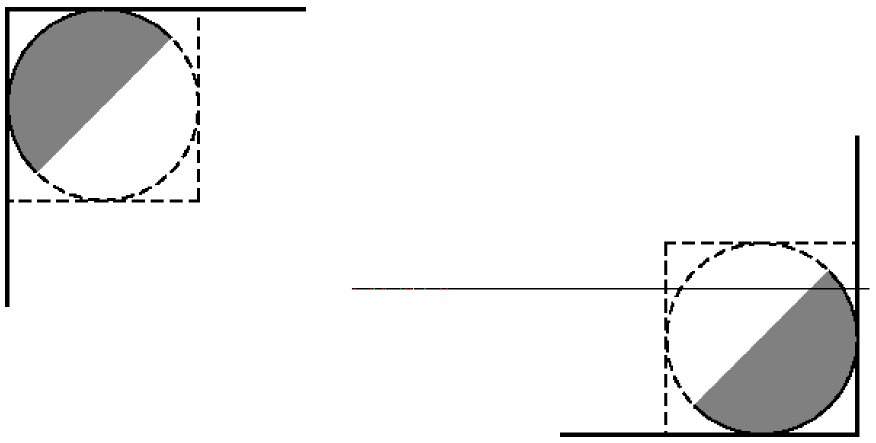


Figure 7-1 Chords in the MS Windows Example

The second part, as shown in **Figure 7-2**, will draw text vertically and horizontally centered in the window, and enclose the text in a rectangle with a thick edge. As we will see later, centering the

Hello, World!

Figure 7-2 Drawing Text in the MS Windows Example

text will be easy. Then in order to draw the box around the text, we will have to calculate the width and height of the text; we'll also want to leave a bit of space between the edge of the box and the text. If *space-x* and *space-y* represent the amount of space we want to leave horizontally and vertically, respectively, then the coordinates for the box can be calculated like this:

```
upper-left-x  = (window-width - text-width) / 2 - space-x
upper-left-y  = (window-height - text-height) / 2 = space-y
lower-right-x = upper-left-x + text-width + 2 * space-x
lower-right-y = upper-left=y + text-height + 2 * space-y
```

```
#include <stdio.h>
#include <string.h>
#include <windows.h>

static void paint_hello_chords( HDC hdc, HWND hwnd );
static void paint_hello_detail( HDC hdc, HWND hwnd );
static void paint_hello_text( HDC hdc, HWND hwnd );
static long PASCAL WndProc(HWND hwnd, UINT msg, UINT wParm, LONG lParm);

int PASCAL WinMain( HANDLE instance,
                    HANDLE prev_instance,
                    LPSTR  cmd_param,
                    int    cmd_show
                  )
{
    static char *app_name = "WinHello";
    HWND        hwnd;
    MSG         msg;
    WNDCLASS    wnd_class;

    if( !prev_instance )
    {
        wnd_class.style         = CS_HREDRAW | CS_VREDRAW;
        wnd_class.lpfnWndProc   = WndProc;
        wnd_class.cbClsExtra    = 0;
        wnd_class.cbWndExtra    = 0;
        wnd_class.hInstance     = instance;
        wnd_class.hIcon         = 0;
        wnd_class.hCursor       = LoadCursor( 0, IDC_ARROW );
        wnd_class.hbrBackground = GetStockObject( WHITE_BRUSH );
        wnd_class.lpszMenuName  = 0;
        wnd_class.lpszClassName = app_name;
        RegisterClass( &wnd_class );
    }
    hwnd = CreateWindow(
        app_name,                     /* window class name    */
        "Uh, Hello? World?",          /* window caption       */
        WS_OVERLAPPEDWINDOW,          /* window style         */
        CW_USEDEFAULT,                /* initial X position   */
        CW_USEDEFAULT,                /* initial Y position   */
        CW_USEDEFAULT,                /* initial X size       */
        CW_USEDEFAULT,                /* initial Y size       */
        0,                            /* parent window handle */
        0,                            /* window menu handle   */
        instance,                     /* program inst. handle */
        NULL                          /* creation parameters  */
                       );

   ShowWindow( hwnd, cmd_show );
   UpdateWindow( hwnd );

   while( GetMessage( &msg, 0, 0, 0 ) )
   {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
   }
   return msg.wParam;
}
```

```
static long PASCAL WndProc( HWND hwnd,
                            UINT msg,
                            UINT wParam,
                            LONG lParam
                          )
{
    long        rcode = 0;
    HDC         hdc;
    PAINTSTRUCT paint;

    switch( msg )
    {
        case WM_PAINT:
            hdc = BeginPaint( hwnd, &paint );
            paint_hello_detail( hdc, hwnd );
            EndPaint( hwnd, &paint );
            break;

        case WM_RBUTTONUP:
            PostQuitMessage( 0 );
            break;

        case WM_DESTROY:
            PostQuitMessage( 0 );
            break;

        default:
            rcode = DefWindowProc( hwnd, msg, wParam, lParam );
   }

   return( rcode );
}

static void paint_hello_detail( HDC hdc, HWND hwnd )
{
    paint_hello_text( hdc, hwnd );
    paint_hello_chords( hdc, hwnd );
}
```

```
static void paint_hello_chords(  HDC hdc, HWND hwnd )
{
    RECT        rect;
    HBRUSH      old_brush,
                new_brush;
    int         diameter = 100;
    COLORREF    color    = RGB( 127, 127, 127 );

    GetClientRect( hwnd, &rect );
    new_brush = CreateSolidBrush( color );
    old_brush = SelectObject( hdc, new_brush );
    Chord( hdc,
           0,         0,
           diameter,  diameter,
           diameter,  0,
           0,         diameter
         );
    Chord( hdc,
           rect.right - diameter, rect.bottom - diameter,
           rect.right,            rect.bottom,
           rect.right - diameter, rect.bottom,
           rect.right,            rect.bottom - diameter
         );
    SelectObject( hdc, old_brush );
    DeleteObject( new_brush );
}

static void paint_hello_text( HDC hdc, HWND hwnd )
{
    RECT        rect;
    SIZE        text_rect;
    HPEN        old_pen,
                new_pen;
    int         start_x     = 0,
                start_y     = 0;
    COLORREF    color       = RGB( 0, 0, 0 );
    char        *hello      = "Hello World!";

    GetClientRect( hwnd, &rect );
    GetTextExtentPoint( hdc,
                        hello,
                        strlen( hello ),
                        &text_rect
                      );
    start_x = (rect.right - text_rect.cx) / 2 - 8;
    start_y = (rect.bottom - text_rect.cy) / 2 - 5;
    new_pen = CreatePen( PS_SOLID, 4, color );
    old_pen = SelectObject( hdc, new_pen );
    Rectangle( hdc, start_x,
                    start_y,
                    start_x + text_rect.cx + 16,
                    start_y + text_rect.cy + 10
             );
    DrawText( hdc, hello, -1, &rect,
              DT_SINGLELINE | DT_CENTER | DT_VCENTER );
    SelectObject( hdc, old_pen );
    DeleteObject( new_pen );
}
```

7.2 "Circles," X/Athena Example

This example uses X and the Athena Widget Set to draw five filled circles along the diagonal of a window; the bounding box for each circle is shown as a dotted line. For aesthetic reasons, there are 25 pixels of space between the edges of the window, and the first and last circles; after subtracting the empty space, each circle is one-fifth the geometry of the window. If the width and/or height of the window is not a perfect multiple of five, the circles will be slightly imperfect. **Figure 7-3** provides a sample of the program output.

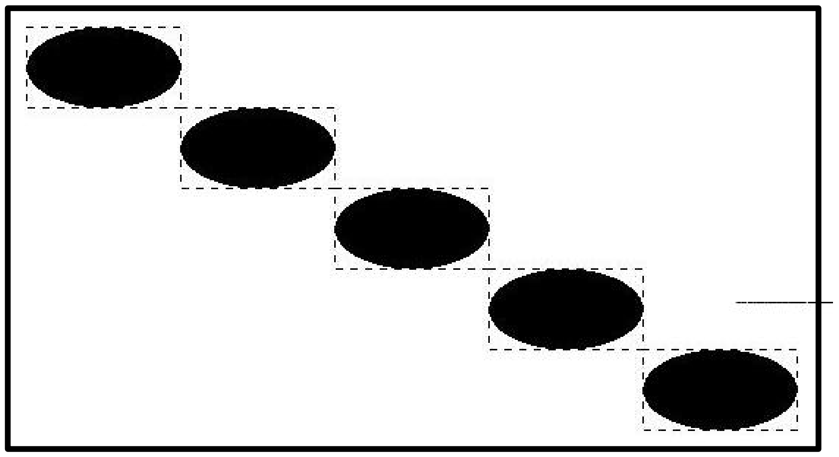


Figure 7-3 Drawing Circles in the X Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Simple.h>

typedef void XT_EVENT_HANDLER_t(Widget, XtPointer, XEvent *, Boolean *);

static void draw_circles(
    Widget widget
);

static void make_test(
    Widget top
);

static XT_EVENT_HANDLER_t event_handler;

main( int argc, char **argv )
{
    XtAppContext context;
    Widget       top      = NULL;

    top = XtAppInitialize( &context,
                           "Test",
                           NULL,
                           0,
                           &argc,
                           argv,
                           NULL,
                           NULL,
                           0
                         );

    make_test( top );
    XtRealizeWidget( top );
    XtAppMainLoop( context );
}

static void make_test( Widget top )
{
    EventMask events = ExposureMask | ButtonReleaseMask |
                       StructureNotifyMask;
    Widget    simple = NULL;

    simple = XtCreateWidget("Canvas", simpleWidgetClass, top, NULL, 0 );
    XtVaSetValues( simple, XtNx,      200,
                           XtNy,      200,
                           XtNwidth,  575,
                           XtNheight, 325,
                           NULL
                 );
    XtAddEventHandler( simple, events, False, event_handler, NULL );
    XtManageChild( simple );
}
```

```
static void draw_circles( Widget widget )
{
    Dimension width    = 0,
              height   = 0;
    XtGCMask  gc_mask  = GCForeground | GCBackground |
                         GCDashList   | GCLineStyle;
    GC        draw_gc  = None;
    Window    window   = XtWindow( widget );
    Screen    *screen  = XtScreen( widget );
    Display   *display = XtDisplay( widget );
    XGCValues gc_values;

    int       inx      = 0;
    int       xlen     = 0;
    int       ylen     = 0;
    int       fudge    = 25;

    gc_values.foreground = BlackPixelOfScreen( screen );
    gc_values.background = WhitePixelOfScreen( screen );
    gc_values.dashes = 5;
    gc_values.line_style = LineOnOffDash;
    draw_gc = XtGetGC( widget, gc_mask, &gc_values );

    XtVaGetValues( widget, XtNwidth,  &width,
                           XtNheight, &height,
                           NULL
                 );
    xlen = (width - 2 * fudge) / 5;
    ylen = (height - 2 * fudge) / 5;

    XClearWindow( display, window );
    for ( inx = 0 ; inx < 5 ; ++inx )
    {
        XFillArc( display,
                  window,
                  draw_gc,
                  inx * xlen + fudge,
                  inx * ylen + fudge,
                  xlen,
                  ylen,
                  0 * 64,
                  360 * 64
                );
        XDrawRectangle( display,
                        window,
                        draw_gc,
                        inx * xlen + fudge,
                        inx * ylen + fudge,
                        xlen,
                        ylen
                      );
    }

    XtReleaseGC( widget, draw_gc );
}
```

```
static void event_handler( Widget    canvas,
                           XtPointer client,
                           XEvent    *event,
                           Boolean   *dispatch
                         )
{
    switch ( event->xany.type )
    {
        case Expose:
        case ConfigureNotify:
            draw_circles( canvas );
            break;

        case ButtonRelease:
            if ( event->xbutton.button == 3 )
                exit( EXIT_SUCCESS );
            break;

       default:
           break;
    }
}
```

7.3 "Circles," X/Motif Example

This example is functionally the same as the example described in **Section 7.2, *"Circles," X/Athena Example***, but it uses the Motif widget set in place of Athena. The only difference is that, instead of a black foreground on a white window background, it will have a white foreground on a black background.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>

typedef void XT_CBPROC_t( Widget, XtPointer, Xtpointer );

static void draw_circles(
    Widget widget
);

static void make_test(
    Widget top
);

static XT_CBPROC_t draw_expose_cb;
static XT_CBPROC_t draw_input_cb;

main( int argc, char **argv )
{
    XtAppContext context;
    Widget       top      = NULL;

    top = XtAppInitialize( &context,
                           "Test",
                           NULL,
                           0,
                           &argc,
                           argv,
                           NULL,
                           NULL,
                           0
                         );

    make_test( top );
    XtRealizeWidget( top );
    XtAppMainLoop( context );
}

static void make_test( Widget top )
{
    Widget  draw    = NULL;
    Screen  *screen = XtScreen( top );
    Pixel   black   = BlackPixelOfScreen( screen );

    draw = XmCreateDrawingArea( top, “canvas”, NULL, 0 );
    XtVaSetValues( draw, XmNbackground, black
                         XmNwidth,      575,
                         XmNheight,     325,
                         NULL
                 );
    XtAddCallback( draw, XmNexposeCallback, expose_cb, NULL );
    XtAddCallback( draw, XmNinputCallback, input_cb, NULL );
    XtManageChild( draw );
}
```

```
static void draw_circles( Widget widget )
{
    Dimension width    = 0,
              height   = 0;
    XtGCMask  gc_mask  = GCForeground | GCBackground |
                         GCDashList   | GCLineStyle;
    GC        draw_gc  = None;
    Window    window   = XtWindow( widget );
    Screen    *screen  = XtScreen( widget );
    Display   *display = XtDisplay( widget );
    XGCValues gc_values;

    int       inx      = 0;
    int       xlen     = 0;
    int       ylen     = 0;
    int       fudge    = 25;

    gc_values.foreground = WhitePixelOfScreen( screen );
    gc_values.background = BlackPixelOfScreen( screen );
    gc_values.dashes = 5;
    gc_values.line_style = LineOnOffDash;
    draw_gc = XtGetGC( widget, gc_mask, &gc_values );

    XtVaGetValues( widget, XtNwidth,  &width,
                           XtNheight, &height,
                           NULL
                 );
    xlen = (width - 2 * fudge) / 5;
    ylen = (height - 2 * fudge) / 5;

    XClearWindow( display, window );
    for ( inx = 0 ; inx < 5 ; ++inx )
    {
        XFillArc( display,
                  window,
                  draw_gc,
                  inx * xlen + fudge,
                  inx * ylen + fudge,
                  xlen,
                  ylen,
                  0 * 64,
                  360 * 64
                );
        XDrawRectangle( display,
                        window,
                        draw_gc,
                        inx * xlen + fudge,
                        inx * ylen + fudge,
                        xlen,
                        ylen
                      );
    }

    XtReleaseGC( widget, draw_gc );
}
```

```
static void expose_cb( Widget widget, XtPointer client, XtPointer call )
{
    draw_circles( widget );
}

static void input_cb( Widget widget, XtPointer client, XtPointer call )
{
    XmDrawAreaCallbackStruct *cb_data = call;
    XEvent                   *event   = call->event;
    Dimension                width    = 0,
                             height   = 0;
    Position                 xco      = 0,
                             yco      = 0;

    if ( event->xany.type == ButtonRelease )
        if ( event->xbutton.button == 3 )
        {
            XtVaGetValues( widget, XmNwidth,  &width,
                                   XmNheight, &height,
                                   NULL
                         );
            xco = event->xbutton.x;
            yco = event->xbutton.y;
            if ( xco >= 0     &&
                 yco >= 0     &&
                 xco < width  &&
                 yco < height
               )
                exit( EXIT_SUCCESS );
        }
}
```

7.4 Calculating Radial Points

As we have seen, an end point of a circular arc is defined by an angle with an origin at the center of the circle along whose edge the arc lies. Some arc drawing primitives, notably the primitive provide by MS Windows, require that this angle, in turn, be defined by a *radial point*. A line is drawn from the origin of the angle to the radial point and extended to the horizon; the point at which the line intersects the edge of the circle becomes the end point of the arc. So far all our examples of arcs have used special cases of angles for which radial points are easy to define; now we will address the general case of calculating the coordinates of radial points.

Consider an angle, α, between 0^o and 90^o that defines the end point of an arc lying on a circle with center (C_x, C_y). For the x coordinate of a radial point, any value at some distance, Δx, to the right of the center of the circle will do, so choose one. As shown in **Figure 7-4**, we can construct the right triangle with the following vertices:

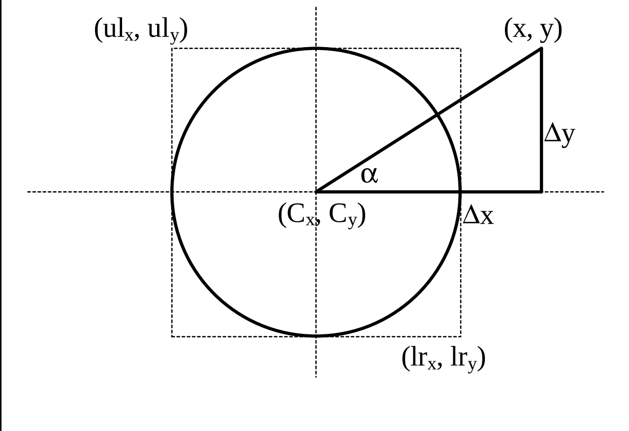


Figure 7-4 Calculating Radial Points

$$(C_x, C_y), (C_x + \Delta x, C_y), (x, y)$$

where (x, y) is a radial point that defines the angle α. The length of the vertical line, Δy, that connects ($C_x + \Delta x$, C_y) to (x, y), and subsequently the value of the y coordinate of the radial point, can be calculated using the following:

$$\Delta y = \Delta x * \tan(\alpha)$$
$$y = C_y - \Delta y$$

(Note: if $\alpha = 90^o$, use $\Delta x = 0$ and $\Delta y = 10$.)

If the circle is inscribed in a bounding box with upper-right and lower-left hand coordinates, (ul_x, ul_y), (lr_x, lr_y), respectively, the center of the circle is given by:

$$C_x - (lr_x - ul_x) / 2$$
$$C_y - (lr_y - ul_y) / 2$$

and the radial point can be calculated using the following:

$$x = (lr_x - ul_x) / 2 + \Delta x$$
$$y = (lr_y - ul_y) / 2 - \Delta x * \tan(\alpha)$$

Exercise: As noted above, this algorithm works for angles between 0^o and 90^o. Adapt the algorithm to work for all angles; keep in mind that the tangent of an angle in the second or fourth quadrant is negative.

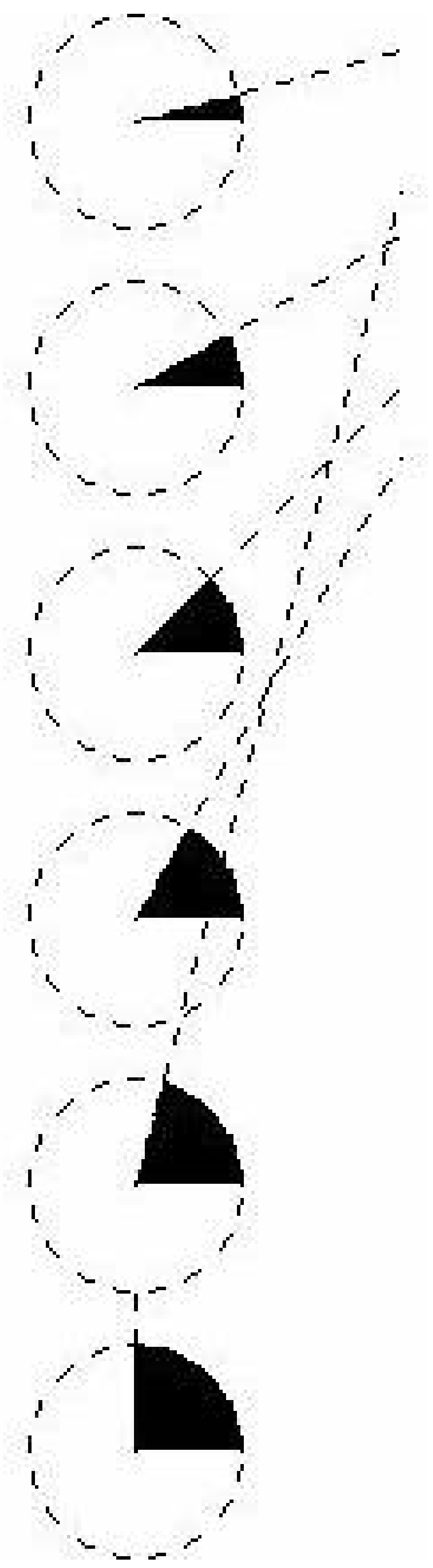


Figure 7-5 Calculating Radial Points, X Example

The following X Window System subroutine demonstrates the above algorithm by drawing 6 filled arcs at 15° increments. For each arc, it shows the circle along whose edge the arc is drawn, then calculates a radial point for one end point of the arc and draws a line from the center of the circle to the radial point.

The output of the subroutine is shown in Error! Reference source not found.. This demonstrates that the line extending from the origin of the angle to the radial point does, indeed, intersect the edge of the defining circle at the end point of the arc.

Note that, since the X primitive that draws filled arcs requires that the bounding box for the arc be defined using the upper-left corner plus width and height method, a slight modification is required to the algorithm as stated above, which assumed that the bounding box would be defined using upper-left and lower-right hand corners.

```
#ifndef PI
#define PI           (3.14159)
#endif
#define RADIANS( d ) ((d) * PI / 180)

static void draw_demo( Widget widget )
{
    XtGCMask  gc_mask  = GCForeground | GCBackground |
                         GCDashList   | GCLineStyle;
    GC        draw_gc  = None;
    Window    window   = XtWindow( widget );
    Screen    *screen  = XtScreen( widget );
    Display   *display = XtDisplay( widget );
    XGCValues gc_values;

    int       inx      = 0;
    int       xlen     = 60;
    int       ylen     = 60;
    int       fudge    = 15;
    int       angle    = 0;
    int       delta_x  = xlen + 15;
    int       delta_y  = 0;
    int       center_x = 0;
    int       center_y = 0;
    int       line_x   = 0;
    int       line_y   = 0;

    gc_values.foreground = BlackPixelOfScreen( screen );
    gc_values.background = WhitePixelOfScreen( screen );
    gc_values.dashes = 5;
    gc_values.line_style = LineOnOffDash;
    draw_gc = XtGetGC( widget, gc_mask, &gc_values );

    XClearWindow( display, window );
    for ( inx = 0 ; inx < 6 ; ++inx )
    {
        angle = (inx + 1) * 15;
        XFillArc( display,
                  window,
                  draw_gc,
                  fudge,
                  inx * (ylen + fudge),
                  xlen,
                  ylen,
                  0 * 64,
                  angle * 64
                );
        XDrawArc( display,
                  window,
                  draw_gc,
                  fudge,
                  inx * (ylen + fudge),
                  xlen,
                  ylen,
                  0 * 64,
                  360 * 64
                );

        center_x = fudge + xlen / 2;
        center_y = inx * (ylen + fudge) + ylen / 2;
        if ( angle == 90 )
        {
            line_x = center_x;
            line_y = center_y - fudge;
        }
        else
        {
            line_x = center_x + delta_x;
            delta_y = delta_x * tan( RADIANS( angle ) ) + .5;
            line_y = center_y - delta_y;
        }

        XDrawLine( display,
                   window,
                   draw_gc,
                   center_x,
                   center_y,
                   line_x,
                   line_y
                 );
    }

    XtReleaseGC( widget, draw_gc );
}
```

7.5 Line Drawing Test Driver

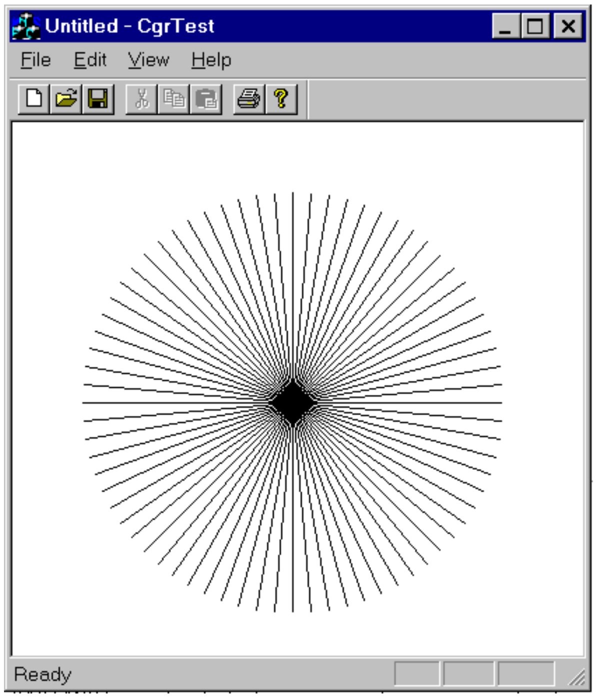


Figure 7-6 Line Test Output

Another trigonometric trick enables us to calculate a radial point that falls exactly on the edge of a circle. This suggests a test driver for the line drawing algorithm that you are implementing as part of your project. If you have coded your algorithm correctly, the subroutine below will draw the starburst pattern shown in**Figure 7-6**.

```
#include <cgr.h>
#define PI           (3.14159)
#define RADIANS( d ) ((d) * PI / 180)

static int      misc_incr      = 5;
static int      misc_scale     = 150;
static int      misc_start_x   = 200;
static int      misc_start_y   = 200;

static void drawStarburst()
{
    CGR_line_t line;
    long       angle   = 0;
    double     end_x   = 0;
    double     end_y   = 0;

    line.end1.xco = misc_start_x;
    line.end1.yco = misc_start_y;
    for ( angle = 0 ; angle < 360 ; angle += misc_incr )
    {
        end_x = cos(RADIANS( angle ) ) * misc_scale + misc_start_x + .5;
        end_y = sin(RADIANS( angle ) ) * misc_scale + misc_start_y + .5;
        line.end2.xco = (int)end_x;
        line.end2.yco = (int)end_y;
        CGR_draw_line( &line );
    }
}
```

8. Introduction to Color

Color is a difficult but important topic. In this lecture we will discuss some of the basic properties of color, the mechanism used by display devices to model color, and some of the approaches used by operating systems to specify color.

8.1 Properties of Color

Color is light, and the properties of what we percieve to be color are directly related to the properties of light: *wavelength, purity* and *amplitude*. In terms of color, these characteristics of light describe *hue, saturation* and *brightness*, respectively. As seen in **Figure 8-1**, brightening a

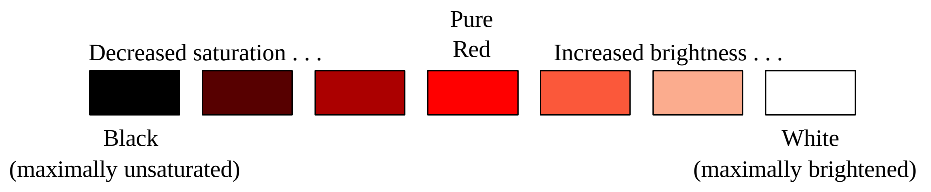

Decreased saturation . . .
Pure Red
Increased brightness . . .
Black (maximally unsaturated)
White (maximally brightened)

Figure 8-1 Hue, Saturation and Brightness

hue lightens the color; a hue at maximum brightness is white. Reducing the saturation of a hue, or *unsaturating* the hue, darkens the color; a maximally unsaturated hue is black.

8.2 Primary Colors

A set of *primary colors* consists of two or more colors that are mixed together to obtain other colors. The range of colors that can be produced by mixing colors in the set is known as the *gamut* for the set. Having many colors in a set of primaries broadens the set's gamut, but no finite set of primaries can produce all possible colors.

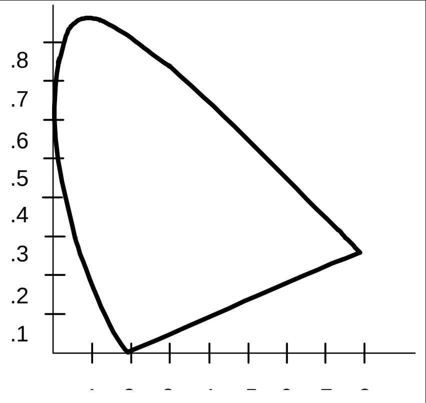


Figure 8-2 The CIE Chromaticity Diagram

In 1931, the International Commission on Illumination developed an idealized model that plots all visible colors in three dimensions. By normalizing the model so that all colors with the same brightness map to the same pont, the two-dimensional graph shown in **Figure 8-2**[2], known as the *CIE Chromaticity Diagram*, is obtained. To discover the subset of colors representing the gamut for some set of primaries, locate each primary color within the diagram, then join them to enclose the largest possible area; the area then represents the primaries' gamut.

[2] Please note that this figure was sketched by hand, and is likely inaccurate. For an accurate representation of the figure, please consult your text book.

8.3 The RGB Color Model

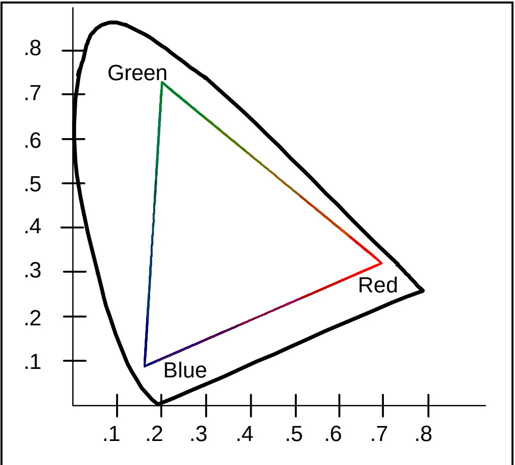


Figure 8-3 The RGB Gamut

Most luminescent graphics devices (for example, your monitor) display colors by dynamically mixing them from a set of primaries; the primaries of choice are most often *red*, *green* and *blue*. The colors that can be formed from this set of primaries, along with the method for mixing colors, is known as the *RGB Color Model*. The gamut for the model, as plotted within the CIE Chromaticity diagram, is shown in **Figure 8-3**.

The RGB Color Model is *additive*, that is, red green and blue frequencies are added together to produce additional colors; red plus blue produces magenta, red plus green produces yellow, and so forth. The full RGB gamut is usually visualized as a cube plotted in three-dimensional cartesian coordinates, with red on the X axis, green on the Y axis and blue on the Z axis. As seen in **Figure 8-4**, the corners of the cube are occupied by red, green and blue; the principle additive colors, cyan, magenta and yellow; plus white, obtained by adding all three colors together, and black, the absence of any of the colors.

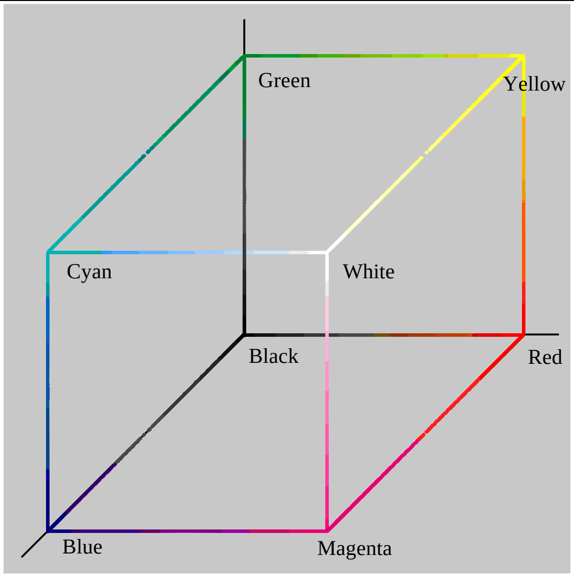


Figure 8-4 The RGB Color Cube

An RGB color can now be specified as a cartesian coordinate triple representing a point inside the color cube. If we choose the length of a side of the cube to be 1, red would be (1, 0, 0), cyan would be (0, 1, 1), and dark magenta would be (.5, 0, .5). Gray values occupy the line, known as the *gray scale*, connecting the black and white corners of the cube, so that red, green and blue values are all equal; 60% gray (Gray-60) would then be (.4, .4, .4), and 35% gray would be (.65, .65, .65).

The RGB model in which red, green and blue intensities are determined by a floating point value between 0 and 1 is known as the *idealized* model. However, since computer operations are usually more efficient when conducted in integers, there are two other common models; in the *scaled 255* model, a side of the color cube is 255, and RGB intensities are encoded as integer values between 0 and 255; in the *scaled 65535* model, a side of the cube is 65535, and intensities are encoded as integer values between 0 and 65535. A few examples of RGB values in all three models are shown in **Figure 8-5**.

		Idealized	Scaled 255	Scaled 65535
⬜	White	(1, 1, 1)	(255, 255, 255)	(65535, 65535, 65535)
⬛	Black	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
🟩	Green	(0, 1, 0)	(0, 255, 0)	(0, 65535, 0)
🟨	Yellow	(1, 1, 0)	(255, 255, 0)	(65535, 65535, 0)
🟪	Magenta	(1, 0, 1)	(255, 0, 255)	(65535, 0, 65535)
▨	Gray-50	(.50, .50, .50)	(127, 127, 127)	(32767, 32767, 32767)
⬛	Gray-75	(.25, .25, .25)	(63, 63, 63)	(16383, 16383, 16383)
🟧	DarkOrange	(.97, .50, .09)	(248, 128, 23)	(63568, 32767, 5898)

Figure 8-5 RGB Values

8.4 Color Displays

Frame Buffers

Color displays store images in a *frame buffer.* The frame buffer is a block of memory associated with your graphics hardware, usually resident on your video card.

255	16	123	255	49
12	96	255	127	33
14	97	129	231	1
66	241	201	111	135
14	56	56	201	135

Figure 8-6 Frame Buffers

As shown in **Figure 8-6**, a frame buffer can be visualized as a two dimensional array of integers. Each integer corresponds to a pixel on the associated display, and the value of the integer encodes an RGB value that determines the color of the pixel. The range of colors that can be displayed by a pixel is determined by the range of values that can be stored in a memory location in the frame buffer; this in turn is determined by the *depth* of the frame buffer; that is, the number of bits assigned to each memory location. Monochrome displays have a depth of one, restricting them to displaying images in black or white. Color displays have a minimum depth of 3, allowing them to display up to 8 colors; *true color* displays have a minimum depth of 24, and can display up to 16 million different colors.

How an integer value in a frame buffer maps to a color is dependent on the specific type of graphics hardware. Consider a theoretical color display, with a depth of three. Each bit in an integer will control one of the three electron guns in the monitor; when a bit is turned on, the associated gun fires at full intensity, and when the bit is off, the gun doesn't fire at all. If *red* is 4, *green* is 2 and *blue* is 1, then 5 corresponds to magenta. The original CGA (for *color graphics adaptor*) devices have a depth of four; bits 4/2/1 correspond to red/green/blue, respectively, and bit 8 represents a *multiplier* or *intensity* bit. When a red, green or blue bit is on, and the intensity bit is off, the corresponding electron gun fires at half intensity; when the intensity bit is on, the gun associated with any other bit that is also on fires at full intensity. In a CGA device, a value of 3 would be dark cyan, and 12 (8 plus 4) would be light red. EGA/VGA devices have a depth of 6, and assign two bits to each of the electron guns, allowing red, green and blue component to have a value of *off* (0), *light intensity* (1), *medium intensity* (2) or *high intensity* (3). **Figure 8-7** demonstrates the range of physical RGB values for CGA devices, and our theoretical depth 3 device.

	R	G	B	Color
⬛	0	0	0	Black
🟥	1	0	0	Red
🟩	0	1	0	Green
🟦	0	0	1	Blue
🟪	1	0	1	Magenta
🟨	1	1	0	Yellow
🟦	0	1	1	Cyan
⬜	1	1	1	White

Depth 3 Device

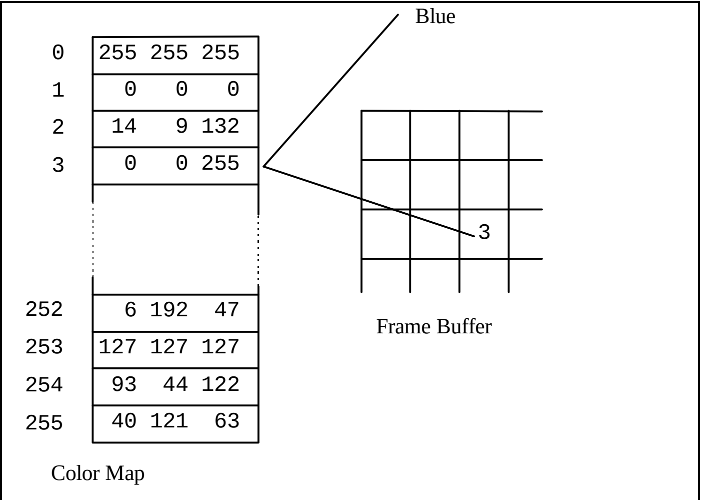
	i	R	G	B	Color
⬛	0	0	0	0	Black
🟥	0	1	0	0	Dark Red
🟩	0	0	1	0	Dark Green
🟦	0	0	0	1	Dark Blue
🟪	0	1	0	1	Dark Magenta
🟫	0	1	1	0	Brown
🟦	0	0	1	1	Dark Cyan
⬜	0	1	1	1	Light Gray
⬛	1	0	0	0	Dark Gray
🟥	1	1	0	0	Light Red
🟩	1	0	1	0	Light Green
🟦	1	0	0	1	Light Blue
🟪	1	1	0	1	Light Magenta
🟨	1	1	1	0	Yellow
🟦	1	0	1	1	Light Cyan
⬜	1	1	1	1	White

CGA (Depth 4) Device

Figure 8-7 Devices and RGB Values

Color Maps

The successor to VGA, Super VGA or SVGA, extended the depth of the frame buffer to eight, allowing up to 256 different colors to be assembled simultaneously. It also added an element of flexibility to color specification by taking advantage of a *color map* (also called a *color lookup table* or *palette*).



0	255	255	255
1	0	0	0
2	14	9	132
3	0	0	255
252	6	192	47
253	127	127	127
254	93	44	122
255	40	121	63

Figure 8-8 Color Maps

As seen in **Figure 8-8**, a color map is an array of RGB triples maintained by an operating system in “normal” memory (i.e. not on the video card). For a frame buffer of depth 8 the array can contain up to 256 colors, but may be shorter depending on operating system constraints; for frame buffers of greater depth the array can be longer. The RGB values contained in the array are usually stored as scaled 255 or scaled 65535 values. An integer stored in the frame buffer now becomes an index into the color map, and changing the color map will change the colors displayed on the associated monitor. Some systems allow applications to have private color maps, but in most systems only one color map can be active at once; for this reason, most systems provide a default color map, which is shared by all applications.

Display Modes

Since the capabilities of a display device are largely a function of the amount of memory contained in the frame buffer, many such devices provide different operating *modes* in which they trade resolution for color. For example, a device with 1.4 megabytes of video memory can provide true color (depth 24) at a resolution of 800 x 600, but only 256 colors (depth 8) at a resolution of 1024 x 1024.

Color Planes

So far we have been discussing frame buffers as though they were (at least virtually) two dimensional in nature, but, as suggested by **Figure 8-9** they are actually three dimensional. For a

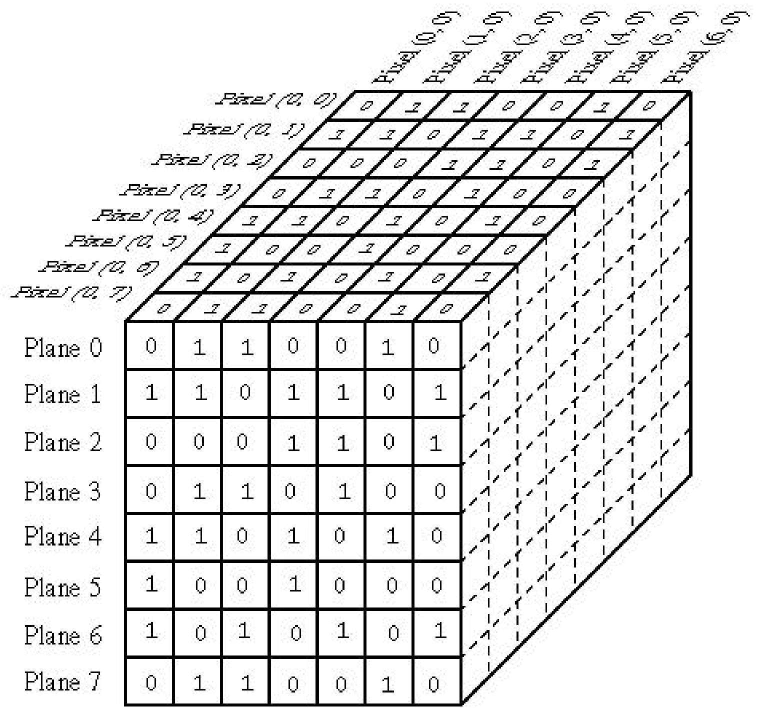


Figure 8-9 Color Planes

particular location in the frame buffer, imagine that the individual bits assigned to the location are lined up along the Z-axis of a cube (if you think about it, this explains why the number of bits assigned to a location in the frame buffer is referred to as the frame buffer's depth). Now imagine a slice taken out of the cube consisting of bit number 0 for every frame buffer location; this would be the frame buffer's zeroth *color plane*.

A frame buffer has as many planes as it has depth, and some video systems are capable of addressing individual planes in the frame buffer. For example, with the right hardware you could store, say, 8 different monochrome images in eight different planes of the frame buffer, and instruct your system to display images from a single plane at a time; then by rapidly switches planes you could perform a sort of animation.

The most common use of color planes is made in systems employing so-called twelve plane display devices. In these systems, the planes of the device are divided into one set of four planes, and one set of eight, and the system assigns a different color map to each. Typically, the four plane color map is declared public, and used for things like window decorations, and the twelve plane color map becomes the private domain of a large graphics application; then the graphics application can change its color map in any way that best suits its purpose without affecting peripheral applications using the public color map.

Dithering

Often a system will attempt to display more colors than the hardware actually supports. This is done via the process of *dithering*, which combines different colors in a small area to simulate the effect of producing a third color.

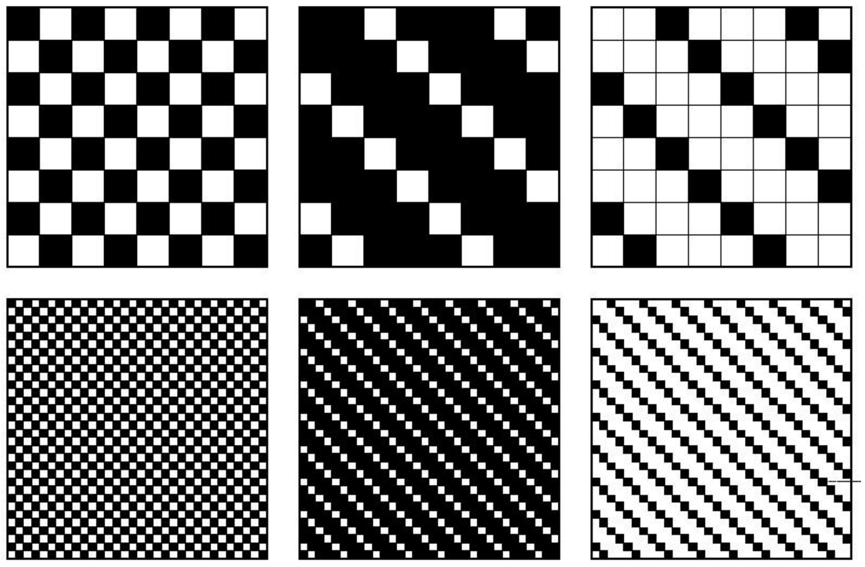


Figure 8-10 Dithering

8.5 Assembling Colors

In order to be able to specify a color in some system, you have to know how to assemble an RGB value in that system. In DOS first you have to know what kind of hardware you are addressing, and how a color is represented using that hardware; brown in CGA, for example, is 0x6 (red and green on at half intensity) and light magenta in EGA is 0x3D (red and blue on at full intensity).

DOS/CGA	i	R	G	B	
Brown	0	1	1	0	(0x6)
Light Blue	1	0	0	1	(0x9)

DOS/EGA	r	g	b	R	G	B	
Light Red	1	1	1	1	0	0	(0x3C)
Light Magenta	1	1	1	1	0	1	(0x3D)

Windows	m	G	B	R
Light Red	0x00	00	00	00
Dark Red	0x00	00	00	77
Light Magenta	0x00	FF	00	FF

Figure 8-11 RGB Values in DOS and Windows

MS Windows uses a more flexible and portable approach. RGB values are encoded as scaled 255 values in the low order three bytes of a long integer, so that, to the programmer, 0x00FF00FF is light magenta no matter what type of hardware you have; the system takes responsibility for converting the value to a device specific representation when necessary.

In the X Window System, RGB values are encoded as scaled 65535 values in three fields of an XColor structure.

```
XColor brown,
       magenta;

brown.red = 32768;
brown.green = 32768;
brown.blue = 0;

magenta.red = 65535
magenta.green = 0;
magenta.blue = 65535;
```

Figure 8-12 RGB Values in X

In Windows colors are extracted from a *palette*, usually the system palette, but possibly a custom or *logical* palette created for a specific application. Windows' default or *stock* palette contains at least 20 colors: the 16 colors normally supported by CGA systems, plus four additional colors, dithered if necessary. (If you would like to investigate building a logical palette for your own application, check your reference pages for *CreatePalette*, *SelectPalette* and *RealizePalette*.)

A Windows palette is addressed using the *COLORREF* data type. This is a 32 bit integer which can be used in three different contexts, or *modes*. The high order byte of a COLORREF value encodes the mode, and the low order bytes are interpreted as follows:

COLORREF

	Name	Value	Macro
MODE0	Absolute	0x00000000	RGB
MODE1	Palette Index	0x01000000	PALETTEINDEX
MODE2	Palette RGB	0x02000000	PALETTERGB

(MODE0 | 0x00FFFF)	Light magenta
(MODE1 | 0x000010)	The seventeenth color in a logical palette
(MODE2 | 0x770077)	The color closest to brown in a logical palette

Figure 8-13 Windows COLORREF Modes

Absolute Mode (mode 0): The low order three bytes encode a scaled 255 RGB value. This value is compared to the values in the stock palette, and the closest color is used.

Palette Index Mode (mode 1): The low order three bytes encode an index into an application's logical palette.

Palette RGB Mode (mode 2): The low order three bytes encode a scaled 255 RGB value. This value is compared to the values in an application's logical palette, and the closest color is used.

In X, colors are stored in a *color map*. This is usually a resource that is shared among all applications. It is an array of 16 bit RGB triples, big enough to store all the values a display system is capable of handling (2^{depth}). An element in the array is called a *color cell*.

Depending on device and operating system, an X color map may be *read/write* or *read only*. A read only color map is fully loaded when the system starts, and can't be changed. Indexes into a read only color map can be requested by calling a system routine and passing an RGB value; the routine interrogates the color map, and returns an index to the color cell that most closely matches the requested color.

Status	Red	Green	Blue
shared	0x7777	0x7777	0x0000
shared	0xFFFF	0x0000	0xFFFF
free			
shared	0x0000	0xFFFF	0x0000
private	0x0210	0x7575	0x7300
. . .	. . .	. . .	. . .

Figure 8-14 A Read/Write X Color Map

A read/write color map is initially empty, and is loaded as needed by allocating color cells as *shared* or *private*. An application can request allocation of a private color cell (which will be denied if the color map is full) and then load it with any RGB value. More often an application will request allocation of a shared color cell by calling a system routine and passing an RGB value, to which X responds as follows:

- If the color is already allocated in a shared color cell, X returns an index to the existing cell.
- If the color is not already allocated, and there is a free cell in the color map, X allocates a free cell, stores the requested value in the cell, and returns an index to it.
- If the color is not already allocated and the color map is full, X returns an index to the shared cell that contains the color that most closely matches the requested color.

As seen in **Figure 8-15**, X also allows you to request a color using a name in the X *color name data base*. A name in the data base is associated with a scaled 255 RGB triple.

Depending on your hardware, X may allow you to create and select different color maps. However, except in the case of unusual hardware, you can only have one color map (X) or one logical palette (Windows) selected, so switching a color map or logical palette is usually considered antisocial.

```
typedef struct
{
    unsigned long  pixel;
    unsigned short red;
    unsigend short green;
    unsigned short blue;
    char           flags;
} XColor;
void change_colors( Widget widget )
{
    Display  *display = XtDisplay( widget );
    Colormap colormap = None;
    XColor   backg,
             foreg_actual,
             foreg_exact;
    XtVaGetValues( widget, XmNcolormap, &colormap, NULL );

    /* Background: light gray */
    backg.red = 0x7777;
    backg.green = 0x7777;
    backg.blue = 0x7777;
    if ( !XAllocColor( display, colormap, &backg ) )
        raise_error( “Couldn’t allocate color” );

    if ( !XAllocNamedColor( display,
                            colormap,
                            “Aquamarine”,
                            &foreg_actual,
                            &foreg_exact
                          )
       )
        raise_error( “Couldn’t allocate Aquamarine” );

    XtVaSetValues( widget, XmNbackground, backg.pixel,
                           XmNforeground, foreg_actual.pixel,
                           NULL
                 );
}
```

Figure 8-15 Allocating Color Cells in X

9. Color Models

As we have seen, the RGB color model can be visualized as a unit cube (a cube with a side of length 1). The RGB model is *additive*; new colors are obtained by summing red, green and blue pigment.

Additive color models work well for luminescent devices, such as monitors, where the value of a color component can be controlled by the intensity of a light source. Additive color models do not work well for *reflective* media, such as a color printer or plotter; nor does the RGB model work well for humans attempting to devise new color schemes.

9.1 The CMY Color Model

The *cyan, magenta and yellow*, or *CMY* color model is useful for producing colors on reflective media. Based on the RGB model it, too, can be visualized as a unit cube, but with cyan, magenta and yellow placed at the unit positions on the X, Y and Z axes.

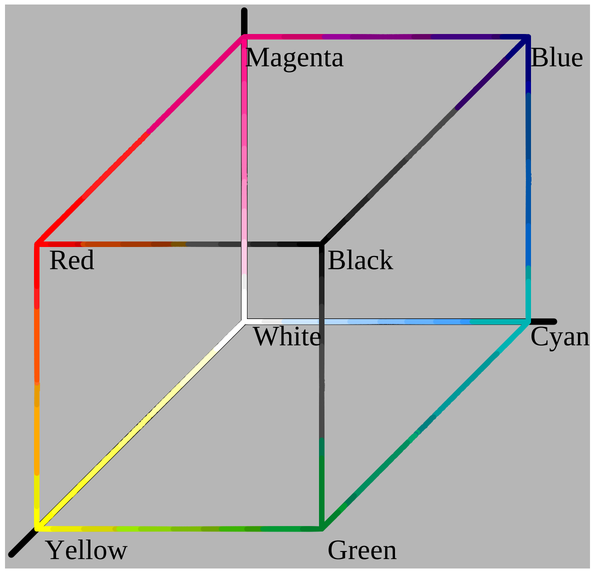


Figure 9-1 The CMY Color Cube

When light is reflected by magenta ink, green hues are completely absorbed, or *subtracted*; therefor we say that the CMY model is a *subtractive* color model. Hard copy color output devices use cyan, magenta and yellow inks to paint a surface with closely spaced dots, or to spray the surface to produce the target color (most such devices use black ink as well).

Within your application you do not typically need to know whether your output device is reflective or luminescent, and choose the appropriate color model to assemble a color. Usually you assemble a color using the RGB color model, then the system or device will translate to the CMY model, if necessary.

Translating between RGB and CMY is straightforward; if color intensities are idealized as floating point values between 0 and 1, then the conversions can be performed as shown in **Figure 9-2**.

RGB to CMY	CMY to RGB
cyan = 1 - red	red = 1 - cyan
magenta = 1 - green	green = 1 - magenta
yellow = 1 - blue	blue = 1 - yellow

Figure 9-2 Converting Between RGB and CMY

9.2 Intuitive Color Models

Humans typically have difficulty selecting a color by mixing various amounts of red, green and blue (or cyan, magenta and yellow). Intuitively, a painter will select a hue, than lighten it by adding white, or darken it by adding black. Adding white to a hue produces a *tint*; adding black desaturates the color to produce a *shade*; adding both black and white produces a *tone*.

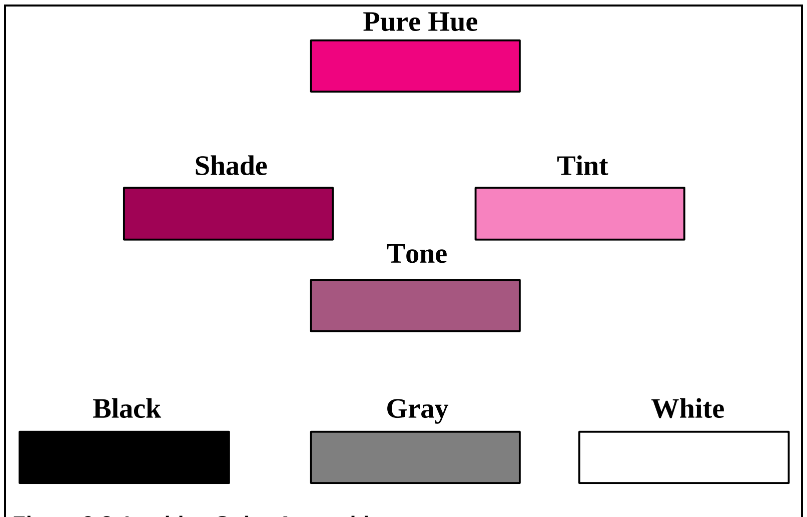


Figure 9-3 Intuitive Color Assembly

Several color models are based on this intuitive concept. The *hue, saturation and value*, or *HSV* model, derived from the RGB model, can be visualized as a hexagonal cone, or *hexcone*.

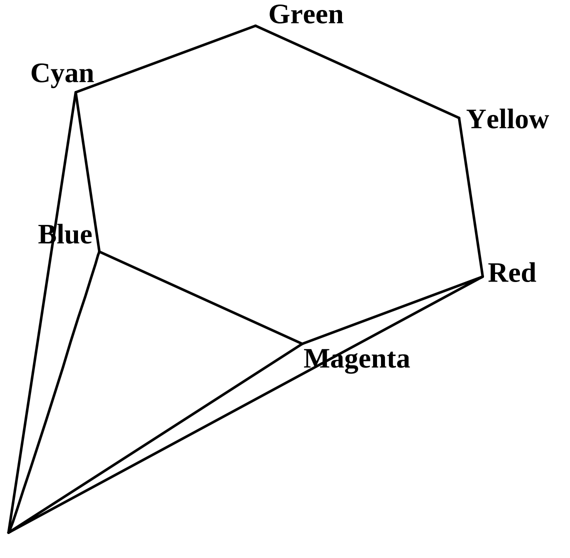


Figure 9-4 The HSV Hexcone

In the HSV model, *hue* is expressed in degrees, between 0 and 360, around the perimeter of the upper surface of the hexcone. *Value*, expressed as a real value between 0 and 1, represents the distance between the origin of the hexcone and the upper surface. *Saturation*, expressed as a real value between 0 and 1, represents the distance between the center of the surface of the hexcone and the hexcone's perimeter.

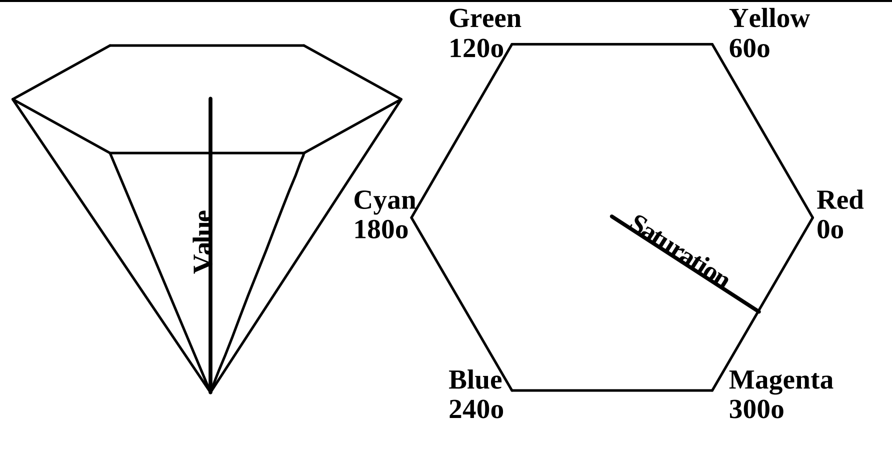


Figure 9-5 Hue, Saturation and Value

Value controls the intensity of the selected hue, and saturation controls hue purity; when value and saturation are both 1, the hue is said to be pure. Decreasing value has the effect of mixing in black, producing a shade; decreasing saturation has the effect of mixing in white, producing a tint. When saturation is 0, hue is undefined and value controls the gray scale: 1 is white, 0 is black, .5 is medium gray, etc.

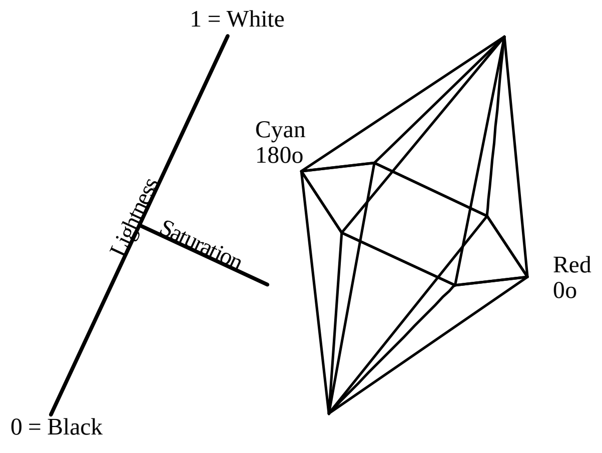


Figure 9-6 The HLS Double Hexcone

The *hue, lightness and saturation,* or *HLS* color model, also based on the RGB model, is similar to the HSV model, but is visualized as a double hexcone. Like the HSV model, hues are represented in degrees around the edge of the surface where the two hexcones meet. Lightness, a value between 0 and 1, is the distance between the origins of the two hexcones; saturation, also a value between 0 and 1, is the distance from the center of the surface to the edge. Pure hues are found when lightness is .5 and saturation is 1. Decreasing lightness to 0 darkens the selected hue to black; increasing it to 1 lightens the color to white. Decreasing saturation decreases the purity of the selected hue. When saturation is 0, hue is undefined and lightness controls the gray scale.

Since the HLS and HSV color models are transformations of the RGB color model, values in one model can be translated to values in any of the other models. Consult your text book for details.

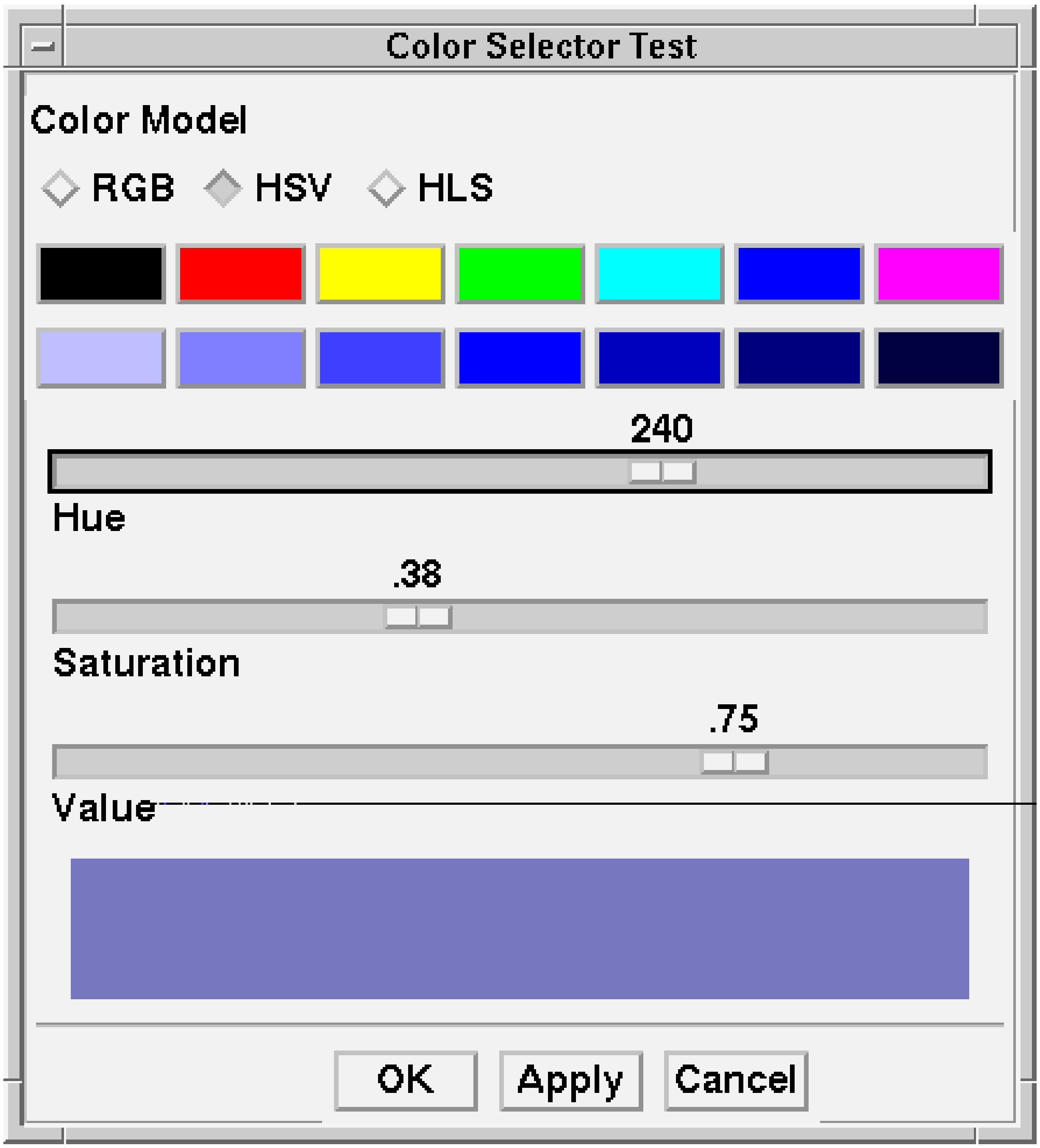


Figure 9-7 A Color Selector