The architecture of DirectX and COM

GDI, the windows graphics device interface is much too slow for creating real time Computer Aided Design graphics and games, sure you can write an embedded graphics programs or adventure games, but that's about it. That's why DirectX was created: to give the PC programmer the tools necessary to write CAD applications and games.

DirectX fulfills the dream of a low-level API that's integrated smoothly with windows and the WIN32 API. By using DirectX, you can access video, audio, input devices, and networking capabilities without writing one line of GDI or using the standard Win32 libraries. And if you use DirectX to work with any of these systems, you won't conflict with GDI, windows or Win32.

DirectX & COM fundamentals

Windows is a shared, cooperative, multitasking operating system, which means that all applications have to share such resources as the mouse, the video display, the sound card and so on. CAD applications and video game usually takes over everything, and because of CAD applications need for high performance.

DirectX gives you shortcut to the hardware without going through normal Windows channels. DirectX is a set of Dynamic Link Libraries (DLL), and low level device drivers that have the ability to control all aspects of the PC without much help from GDI or the standard Win32 libraries. And to create a DirectX application, all you need are the header files, the DirectX libraries, and the DLLs on your machine.

To make the magical DirectX technology work, Microsoft had to come up with some new technologies and conventions to make DirectX very robust. In other words a DirectX application written for DirectX 1.0 should be able to run on a computer with DirectX 3.0 or 8.0 installed. In addition, Microsoft knew that a technology like DirectX would get out of hand very quickly if it was written without a great deal of foresight and planning. What was need was a way of writing software that was object oriented, upgradeable, capable of working with multiple languages, and black-box like to programmer.

COM – Component Object Model is a technology invented a few years back as nothing more than a magazine article that described a set of programming techniques to create component software, much like computer chips. When you are designing with digital chips, you don't care what's inside the chip, whether it be a silicon or arsenic. All you care about is that if you fallow the rules of the chip's interface, the chip works.

Further more, if you connect the output of one chip to another, and as long as the inputs and the outputs in the right format, the chips works together. This analogy is the basis for component software and COM. The idea COM is to create software components that are like computer chips or Lego blocks, that you can just plug in together. As long as you fallow the rules, they work.

The Components of DirectX

The various components of DirectX delve into each aspect of game design; graphics, sound, input, 3D, and networking, Figure 1 illustrates all DirectX components and their relationship to Win32, GDI, and hardware. Notice that GDI and DirectX are on the different sides of the border: Each has access to the other and to the hardware. The blocks called the HAL (Hardware Abstraction Layer) and HEL (Hardware Emulation Layer) are also very important.

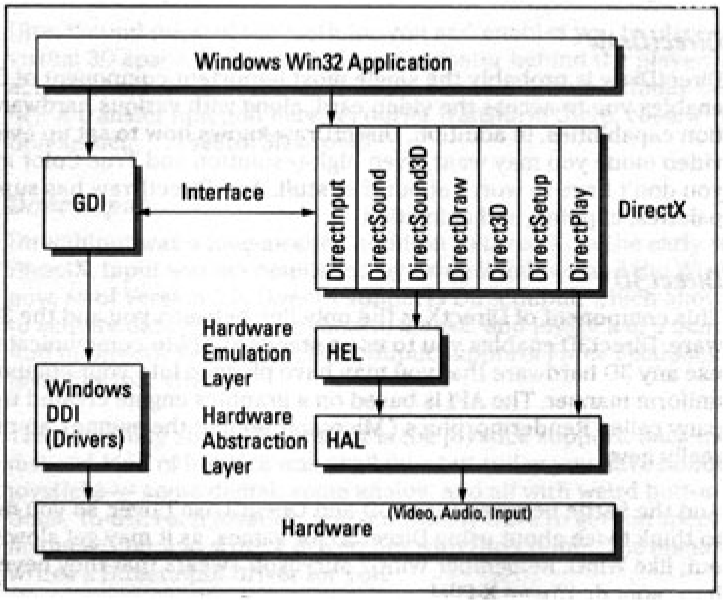

Figure 1: DirectX and its components

HAL: The Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is the lowest level of software in DirectX, consisting of the hardware drivers provided by the manufacturer to control the hardware directly. This layer of software gives you the utmost performance because it talks directly to the hardware. Of course, you don’t actually make calls to the HAL yourself; DirectX does that for you.

HEL: The Hardware Emulation Layer

The Hardware emulation layer(HEL) is built on top of the HAL. In general, DirectX is designed to take the advantage of hardware if the hardware is there, but DirectX still works if hardware isn’t available. For example, suppose that you write some graphics code, assuming that the hardware you are running on supports bitmap rotation and scaling. You therefore make calls to DirectX to scale and rotate bitmaps. On hardware that supports scaling and rotation, your code runs at full speed and uses the hardware, but if you run on hardware that does not support scaling and rotation, that’s when the HEL kicks in. The HEL emulates the functionality of the HAL with software algorithms so that you don’t know the difference. Of course, the code runs more slowly because it’s being emulated, but it does run. That’s the reason for the HEL.

DirectDraw

DirectDraw is probably the single most important component of DirectX. It enables you to access the video card, along with various hardware acceleration capabilities. In addition, DirectDraw knows how to set up every single video mode you may want, even high-resolution and true color modes. And DirectDraw has support for palettes, clipping, and animation.

Direct3D

This component of DirectX is the only link between you and the 3D hardware. Direct3D enables you to use a standard API to communicate with and use any 3D hardware that you may have plugged into your computer, in a uniform manner. The API is based on a graphics engine created by a company called Rendermorphics (Microsoft bought the engine), so that API isn’t really new.

DirectSound

Writing sound drivers for the PC is nearly impossible; so writing sound software is a full-time job. In the past most game programmers licensed their sound engines from a third party, such as John Miles or Diamondware Sound Toolkit. And these sound engines weren’t cheap. But with DirectSound, this situation is no more.

DirectSound works with every sound card. It supports pure digital mixing of multiple channels in real-time. In addition, the newer versions of DirectSound support MIDI music. But MIDI has been dying off slowly, now that pure digital is CD quality and memory has become so cheap. On the other hand, new wave-table and wave-guide synthesizers are making MIDI comeback, so supporting MIDI is nice insurance of compatibility.

DirectSound3D

DirectSound3D is based on DirectSound and is an implementation of 3D sound. The theory behind 3D sound is that you can simulate how a real object would sound at any position in a space as long as you can control the input into each ear. You control the input by shifting the frequency of the sound and its amplitude, harmonics, and timing, based on mathematical models of how sound interacts with geometry of your head along with how it travels through space. DirectSound does all the math for you and enables you to place sounds in a virtual 3D space.

DirectInput

DirectInput was a long-awaited addition to DirectX. In the early releases of DirectX, input was accomplished by using Windows and the Win32 API. But now, as of version 3.0, DirectX supports DirectInput, which allows a program to acquire data from the keyboard, mouse, and joystick in a uniform manner.

DirectDraw Applications

DirectDraw is the drawing component of DirectX and is the most important of the components. Not only does it let you create 32-bit high-resolution games, but with it, you can almost circumvent Windows and get rid of GDI.
DirectDraw is implemented in run-time as a .DLL or a set of .DLLs. But as long as DirectX has been installed on the computer, you don’t need to worry about the physical implementation of the DirectX COM objects. To write application that uses DirectDraw, you need only two files in your project:
DDRAW.H: The header file for DirectDraw.
DDRAW.LIB: The Library file that contains all the code, imports, and COM Obeject.dll loaders.

Interfacing to DirectDraw

Every COM component has a number of interfaces, and DirectDraw is no different. You must communicate with the component through these interfaces, period.
The interfaces of DirectDraw are a bit arbitrary in their names and functionality from a software-design perspective. The designers could have selected the interfaces differently, but all crank calls late at night to the designers did not seem to have an effect; so this is what we are stuck with. Each interface is supposed to model, or represent, a different part of the video system:
IUnknown: The interface class that all interfaces must be derived from.
IDirectDraw: Represents the video card. This interface is used to select video modes and set the overall system-cooperation level. It is the main interface(the core) of the COM object that we create; from it we request other interfaces.
IDirectDrawSurface: Represents the video memory or the drawing surface(s) that you draw on.
IDirectDrawPalette: Represents the color palette associated with the drawing surface.
IDirectDrawClipper: Represents a DirectDraw clipper, which is a set of rectangles that DirectDraw can draw into.

Of course, we need to know all the functions within each interface, we will have a look at all the important functions or methods.

1. Create a DirectDraw object and obtain access to the main interface IDirectDraw.
2. Set a video mode and cooperation level; from there, create one or more DirectDraw surfaces (IDirectDrawSurface) to draw on.
3. Depending on the color depth you may create a palette (IDirectDrawPalette).
4. Create a clipper (IDirectDrawClipper), if desired.

Creating a DirectDraw Object

Because DirectX and Windows are so well integrated, all we need to do to get DirectDraw working is create a DirectDraw object by creating a minimum windows application and a single window for DirectDraw to anchor itself to.

Before we create a DirectDraw object, we should be aware of the following about data structures:

> DirectX and, hence, DirectDraw have a veritable plethora of data structures, and these data structures are nested, with lots of fields in them.
>
> The most important rule about DirectX data structures is that just about every one of them has a *dwSize* field, which indicates the size of the data structure and is used to compute the actual length of variant-length data structures that DirectX uses.

To create a DirectDraw object, use the function DirectDrawCreate(), which has the following prototype:

```
HRESULT DirectDrawCreate(GUID FAR *lpGUID, LPDIRECTDRAW FAR *lplpDD,
                                    IUnknown FAR *pUnkOuter);
```

lpGuid: A GUID(Globally Unique Identifier) that selects the type of video driver you want to use. NULL selects the default driver.
lplpDD: Where the function places the address of the COM interface, if successful.
pUnkOuter: An advanced feature; always set it to NULL.

Here is the code for creating DirectDraw object:

```
LPDIRECTDRAW lpdd;                    // pointer to interface object
// create object and test for error
if(DirectDrawCreate(NULL,&lpdd,NULL)!=DD_OK)
            { // error }
```

If the call was successful, lpdd points to a valid DirectDraw object interface, and you are free to use lpdd to call functions.

Selecting Video Modes

Changing the video mode is one of the most important feature of DirectDraw. In the Win32 API, changing the video mode is possible, but it is like mixing matter with antimatter.
The function to change video modes is SetDisplayMode(), and you use it to select horizontal and vertical resolution along with the color depth in bits per pixel.

Here is the prototype of SetDisplayMode;

```
HRESULT SetDisplayMode(
            DWORD dwWidth,                // Width of the mode in pixels
            DWORD dwHeight,               // Height of mode in pixels
            DWORD dwBPP,                  // Bits Per Pixel
            DWORD dwRefreshRate,          // Refresh rate, set to zero
            DWORD dwFlags);               // Flags, set to zero
```

The Source Code of DirectX Project

Project.h

```
#define SCREEN_WIDTH    800  // size of screen
#define SCREEN_HEIGHT   600
#define SCREEN_BPP      8    // bits per pixel
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
// Externals
extern HWND main_window_handle; // save the window handle
extern int screen_width,                         // width of screen
           screen_height,                        // height of screen
           screen_bpp;                           // bits per pixel
int screen_width  = SCREEN_WIDTH,            // width of screen
    screen_height = SCREEN_HEIGHT,           // height of screen
    screen_bpp    = SCREEN_BPP;              // bits per pixel
//Globals
extern UCHAR            *back_buffer;        // secondary back buffer
LPDIRECTDRAWSURFACE          lpddsprimary = NULL;  // dd primary surface
LPDIRECTDRAWSURFACE          lpddsback    = NULL;  // dd back surface
int                  back_lpitch     = 0;    // memory line pitch
DDSURFACEDESC            ddsd;               // a direct draw surface description struct
LPDIRECTDRAW             lpdd        = NULL;  // dd object
DDSCAPS              ddscaps ;           // a direct draw surface capabilities struct
extern LPDIRECTDRAWPALETTE  lpddpal;             // a pointer to the created dd palette
LPDIRECTDRAWPALETTE  lpddpal      = NULL;  // a pointer to the created dd palette
PALETTEENTRY         palette[256];         // color palette
extern PALETTEENTRY      palette[256];         // color palette
DWORD                start_clock_count = 0; // used for timing
LPDIRECTDRAWCLIPPER  lpddclipper  = NULL;  // dd clipper
UCHAR                *primary_buffer = NULL; // primary video buffer
UCHAR                *back_buffer    = NULL; // secondary back buffer
//Macros
#define DD_INIT_STRUCT(ddstruct) { memset(&ddstruct,0,sizeof(ddstruct));
ddstruct.dwSize=sizeof(ddstruct); }
DWORD Get_Clock(void)
{
// this function returns the current tick count
// return time
return(GetTickCount());
} // end Get_Clock
DWORD Wait_Clock(DWORD count)
{
// this function is used to wait for a specific number of clicks
// since the call to Start_Clock
while((Get_Clock() - start_clock_count) < count);
return(Get_Clock());
} // end Wait_Clock

//DDraw functions
UCHAR *DD_Lock_Back_Surface(void)
```

```
{
// this function locks the secondary back surface and returns a pointer to it
// and updates the global variables secondary buffer, and back_lpitch
// is this surface already locked
if (back_buffer)
   {
   // return to current lock
   return(back_buffer);
   } // end if
// lock the primary surface
DD_INIT_STRUCT(ddsd);
lpddsback->Lock(NULL, &ddsd, DDLOCK_WAIT |
DDLOCK_SURFACEMEMORYPTR,NULL);
// set globals
back_buffer = (UCHAR *)ddsd. lpSurface;
back_lpitch = ddsd.lPitch;
// return pointer to surface
return(back_buffer);
} // end DD_Lock_Back_Surface

int Draw_Pixel(int x, int y, int color,
               UCHAR *video_buffer, int lpitch)
{
// this function plots a single pixel at x, y with color
video_buffer[x + y*lpitch] = color;
// return success
return(1);
} // end Draw_Pixel

int DD_Unlock_Back_Surface(void)
{
// this unlocks the secondary
// is this surface valid
if (!back_buffer)
   return(0);
// unlock the secondary surface
lpddsback->Unlock(back_buffer);
// reset the secondary surface
back_buffer = NULL;
back_lpitch = 0;
// return success
return(1);
} // end DD_Unlock_Back_Surface
int DD_Fill_Surface(LPDIRECTDRAWSURFACE lpdds, int color)
{
DDBLTFX ddbltfx; // this contains the DDBLTFX structure
// clear out the structure and set the size field
DD_INIT_STRUCT(ddbltfx);
// set the dwfillcolor field to the desired color
ddbltfx.dwFillColor = color;
```

```
// ready to blt to surface
lpdds->Blt(NULL,      // ptr to dest rectangle
          NULL,       // ptr to source surface, NA
          NULL,       // ptr to source rectangle, NA
          DDBLT_COLORFILL | DDBLT_WAIT,   // fill and wait
          &ddbltfx);  // ptr to DDBLTFX structure
// return success
return(1);
} // end DD_Fill_Surface

int DD_Init(int width, int height, int bpp)
{
// this function initializes directdraw
int index; // looping variable
// create object and test for error
if (DirectDrawCreate(NULL,&lpdd,NULL)!=DD_OK)
   return(0);
// set cooperation level to windowed mode normal
if (lpdd->SetCooperativeLevel(main_window_handle,
          DDSCL_ALLOWMODEX | DDSCL_FULLSCREEN |
          DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)!=DD_OK)
    return(0);
// set the display mode
if (lpdd->SetDisplayMode(width, height, bpp)!=DD_OK)
   return(0);
// set globals
screen_height = height;
screen_width = width;
screen_bpp = bpp;
// Create the primary surface
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
// we need to let dd know that we want a complex
// flippable surface structure, set flags for that
ddsd.ddsCaps.dwCaps =
  DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP | DDSCAPS_COMPLEX;
// set the backbuffer count to 1
ddsd.dwBackBufferCount = 1;
// create the primary surface
lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL);
// query for the backbuffer i.e. the secondary surface
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
lpddsprimary->GetAttachedSurface(&ddscaps, &lpddsback);
// create and attach palette
// create palette data
// clear all entries defensive programming
memset(palette,0,256*sizeof(PALETTEENTRY));
// create a R,G,B,GR gradient palette
for (index=0; index < 256; index++)
```

```
    {
    // set each entry
    if (index < 64)
        palette[index].peRed = index*4;
    else          // shades of green
    if (index >= 64 && index < 128)
        palette[index].peGreen = (index-64)*4;
    else          // shades of blue
    if (index >= 128 && index < 192)
       palette[index].peBlue = (index-128)*4;
    else          // shades of gray
    if (index >= 192 && index < 256)
        palette[index].peRed = palette[index].peGreen =
        palette[index].peBlue = (index-192)*4;
   // set flags
    palette[index].peFlags = PC_NOCOLLAPSE;
    } // end for index
// now create the palette object
if (lpdd->CreatePalette(DDPCAPS_8BIT | DDPCAPS_INITIALIZE |
DDPCAPS_ALLOW256,
                         palette, &lpddpal, NULL)!=DD_OK)
   return(0);
// attach the palette to the primary
if (lpddsprimary->SetPalette(lpddpal)!=DD_OK)
   return(0);
// clear out both primary and secondary surfaces
DD_Fill_Surface(lpddsprimary,0);
DD_Fill_Surface(lpddsback,0);
// return success
return(1);
} // end DD_Init

DWORD Start_Clock(void)
{
// this function starts the clock, that is, saves the current
// count, use in conjunction with Wait_Clock()
return(start_clock_count = Get_Clock());
} // end Start_Clock

int DD_Shutdown(void)
{
// this function release all the resources directdraw
// allocated, mainly to com objects
// release the clipper first
if (lpddclipper)
    lpddclipper->Release();
// release the palette
if (lpddpal)
   lpddpal->Release();
// release the secondary surface
```

```
if (lpddsback)
    lpddsback->Release();
// release the primary surface
if (lpddsprimary)
   lpddsprimary->Release();
// finally, the main dd object
if (lpdd)
    lpdd->Release();
// return success
return(1);
} // end DD_Shutdown

int Draw_Text_GDI(char *text, int x, int y, COLORREF color,
LPDIRECTDRAWSURFACE lpdds)
{
// this function draws the sent text on the sent surface
// using color index as the color in the palette
HDC xdc; // the working dc
// get the dc from surface
if (lpdds->GetDC(&xdc)!=DD_OK)
   return(0);
// set the colors for the text up
SetTextColor(xdc, color);
// set background mode to transparent so black isn't copied
SetBkMode(xdc, TRANSPARENT);
// draw the text a
TextOut(xdc, x, y, text, strlen(text));
// release the dc
lpdds->ReleaseDC(xdc);
// return success
return(1);
} // end Draw_Text_GDI

int DD_Flip(void)
{
// this function flip the primary surface with the secondary surface
// test if either of the buffers are locked
if (primary_buffer || back_buffer)
   return(0);
// flip pages
while(lpddsprimary->Flip(NULL, DDFLIP_WAIT)!=DD_OK);
// flip the surface
// return success
return(1);
} // end DD_Flip
```

Project.c

```
// INCLUDES //////////////////////////////////////////////////
#define WIN32_LEAN_AND_MEAN
#define INITGUID
#include <windows.h>   // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <objbase.h>
#include <iostream.h>   // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
#include <ddraw.h>     // DirectX includes
#include "stars.h"
// DEFINES ///////////////////////////////////////////////////
// defines for windows
#define WINDOW_CLASS_NAME "WINXCLASS"  // class name
#define WINDOW_WIDTH    64   // size of window
#define WINDOW_HEIGHT   48
// starfield defines
#define MAX_STARS          400
// PROTOTYPES ////////////////////////////////////////////////
// program console
int Prog_Init(void *parms=NULL);
int Prog_Shutdown(void *parms=NULL);
int Prog_Main(void *parms=NULL);
void Move_Stars(void);
void Draw_Stars(void);
void Init_Stars(void);

// TYPES /////////////////////////////////////////////////////
// used to contain a single star
typedef struct STAR_TYP
    {
    UCHAR color;
    int x, y;
    int velocity;
    } STAR, *STAR_PTR;

// GLOBALS ///////////////////////////////////////////////////

HWND main_window_handle = NULL; // save the window handle
```

```
HINSTANCE main_instance = NULL; // save the instance
char buffer[80];                // used to print text
int moving_up = 0;
STAR stars[MAX_STARS]; // the star field


// PROTOTYPES ////////////////////////////////////////////
int Color_Scan(int x1, int y1, int x2, int y2,
    UCHAR scan_start, UCHAR scan_end,
    UCHAR *scan_buffer, int scan_lpitch);


// FUNCTIONS ////////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wparam,
LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT     ps;             // used in WM_PAINT
HDC             hdc;       // handle to a device context
// what is the message
switch(msg)
        {
        case WM_CREATE:
      {
                // do initialization stuff here
                return(0);
                } break;
   case WM_PAINT:
       {
       // start painting
       hdc = BeginPaint(hwnd,&ps);
       // end painting
       EndPaint(hwnd,&ps);
       return(0);
      } break;
        case WM_DESTROY:
                {
                // kill the application
                PostQuitMessage(0);
                return(0);
                } break;
        default:break;
   } // end switch
// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

```
// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain(         HINSTANCE hinstance,
                                      HINSTANCE hprevinstance,
                                      LPSTR lpcmdline,
                                      int ncmdshow)
{
// this is the winmain function
WNDCLASS winclass;          // this will hold the class we create
HWND           hwnd;        // generic window handle
MSG              msg;       // generic message
// first fill in the window class stucture
winclass.style                    = CS_DBLCLKS | CS_OWNDC |
                     CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc          = WindowProc;
winclass.cbClsExtra             = 0;
winclass.cbWndExtra            = 0;
winclass.hInstance                = hinstance;
winclass.hIcon                              = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor                 = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground       = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName       = NULL;
winclass.lpszClassName       = WINDOW_CLASS_NAME;
// register the window class
if (!RegisterClass(&winclass))
          return(0);
// create the window, note the use of WS_POPUP
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
                                                      "EE 515 DirectX Project",    // title
                                                      WS_POPUP | WS_VISIBLE,
                                                      0,0,       // x,y
                                                      WINDOW_WIDTH,  // width
                     WINDOW_HEIGHT, // height
                                                      NULL,            // handle to parent
                                                      NULL,            // handle to menu
                                                      hinstance,// instance
                                                      NULL)))       // creation parms
return(0);
// save the window handle and instance in a global
main_window_handle = hwnd;
main_instance      = hinstance;
Prog_Init();
// enter main event loop
while(1)
          {
          if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                    {
                    // test if this is a quit
      if (msg.message == WM_QUIT)
        break;
```

```
                // translate any accelerator keys
                TranslateMessage(&msg);
                // send the message to the window proc
                DispatchMessage(&msg);
                } // end if
    // main program processing goes here
    Prog_Main();

        } // end while
// shutdown program and release all resources
Prog_Shutdown();

// return to Windows like this
return(msg.wParam);

} // end WinMain


void Init_Stars(void)
{
// this function initializes all the stars in such a way
// that their intensity is proportional to their
// velocity

for (int index=0; index<MAX_STARS; index++)
    {
    // random postion
    stars[index].x = rand()%SCREEN_WIDTH;
    stars[index].y = rand()%SCREEN_HEIGHT;

    // select star plane
    int plane = rand()%4; // (1..4)

    // based on plane select velocity and color
    stars[index].velocity = -(1 + plane*2);
    stars[index].color =255; //- (plane*32);

    } // end for index

} // end Init_Stars

//////////////////////////////////////////////////////
void Move_Stars(void)
{
// this function moves all the stars
for (int index=0; index<MAX_STARS; index++)
    {
    // translate upward
        stars[index].y-=(stars[index].velocity+((moving_up*stars[index].velocity)));
        // test for collision with top of screen
     if (stars[index].y >= SCREEN_HEIGHT)
        stars[index].y-=SCREEN_HEIGHT;
     } // end for index

} // end Move_Stars

///////////////////////////////////////////////////////////

void Draw_Stars(void)
{
// this function draws all the stars
// lock back surface
DD_Lock_Back_Surface();
// draw all the stars
for (int index=0; index<MAX_STARS; index++)
     {
           // draw stars
     Draw_Pixel(stars[index].x,stars[index].y, stars[index].color,back_buffer, back_lpitch);
     } // end for index
// unlock the secondary surface
DD_Unlock_Back_Surface();

} // end Draw_Stars

///////////////////////////////////////////////////////////

int Prog_Init(void *parms)
{
// initialize directdraw
DD_Init(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP);
Init_Stars();

// seed random number generate
srand(Start_Clock());

// hide the mouse
ShowCursor(FALSE);

// return success
return(1);

} // end Prog_Init

///////////////////////////////////////////////////////////
int Prog_Shutdown(void *parms)
{
// this function is where you shutdown your program and
// shutdonw directdraw
DD_Shutdown();
```

```
// return success
return(1);
} // end Prog_Shutdown
////////////////////////////////////////////////////////////

int Prog_Main(void *parms)
{
static int ready_counter = 0,
           ready_state   = 0;
 // check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) && ready_state)
    PostMessage(main_window_handle, WM_DESTROY,0,0);
// start the timing clock
Start_Clock();
// reset upward motion flag
moving_up = 0;
// clear the drawing surface
DD_Fill_Surface(lpddsback, 0);
// move the stars
Move_Stars();
// draw the stars
Draw_Stars();
// draw get ready?
if (!ready_state)
   {
   // draw text
   Draw_Text_GDI("EE 515, DirectX Project !",320-8*strlen("EE 515, DirectX Project !")/2,
200,RGB(255,255,0),lpddsback);
   // increment counter
   if (++ready_counter > 50)
      {
      // set state to ready
      ready_state = 1;
      ready_counter = 0;
      } // end if
   } // end if
 // flip the surfaces
DD_Flip();
// sync to 30ish fps
Wait_Clock(30);
// return success
return(1);
} // end Prog_Main
///////////////////////////////////////////////////////////
```