Bitmaps, Animation, Timers, DirectX

Introduction to Windows Bitmaps

- See CS-360, CS-460/560 Notes & Programs:

http://www.cs.binghamton.edu/~reckert/460/bitmaps.htm
http://www.cs.binghamton.edu/~reckert/360/class4a.htm
http://www.cs.binghamton.edu/~reckert/360/bitmap1_cpp.htm
http://www.cs.binghamton.edu/~reckert/360/bitmap3_cpp.htm

Bitmap: An Off-screen Canvas

- Rectangular image, can be created with almost any paint program
- Data structure that stores a matrix of pixel values in memory
- Pixel value stored determines color of pixel
- Windows supports 4-bit, 8-bit (indirect) and 16 or 24-bit (direct) pixel values
- Can be stored as .bmp file (static resource data)
- Can be edited; can save any picture
- Takes up lots of space (no compression)

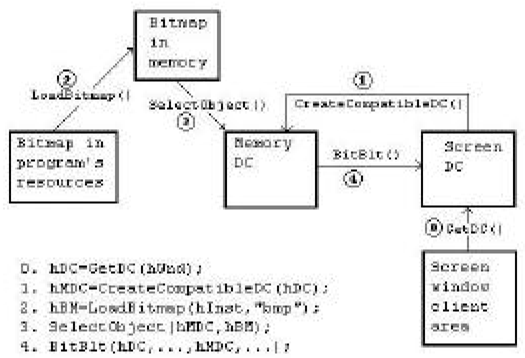
- A GDI object, must be selected into a DC to be used
- Think of as the canvas of a DC upon which drawing takes place
- Must be compatible with a video display or printer
- Can be manipulated invisibly and apart from physical display device
- Fast transfer to/from physical device ==> flicker free animation
- Does not store info on drawing commands

Using Device Dependent Bitmaps

A. Create and save bitmap using a paint editor --> image.bmp file
   Add to program's resource script file
   – e.g.: IDB_HOUSE BITMAP "HOUSE.BMP"
   – easier to select: “Project | Add Resource | Bitmap | Import”
B. In Program, Instantiate a CBitmap object
```
CBitmap bmp1;
```
C. Load it from the program's resources:
```
bmp1.LoadBitmap(IDB_HOUSE);
```

- D. Display the bitmap
  0. Get a ptr to the screen DC (as usual), pDC
  1. Create a memory device context compatible with the screen DC
```
CDC dcMem;
dcMem.CreateCompatibleDC(pDC);
```
  2. Select bitmap into the memory DC
```
CBitmap* pbmpold = dcMem.SelectObject(&bmp1);
```
  3. Copy bitmap from memory DC to device DC using BitBlt() or StretchBlt()
  4. Select bitmap out of memory DC

Using Bitmaps



```
0. hDC=GetDC(hWnd);
1. hMDC=CreateCompatibleDC(hDC);
2. hBM=LoadBitmap(hInst,"bmp");
3. SelectObject(hMDC,hBM);
4. BitBlt(hDC,...,hMDC,...);
```

A Memory DC

- Like a DC for a physical device, but not tied to device
- Used to access a bitmap
- Bitmap must be selected into a memory DC before displayable on physical device
- CreateCompatibleDC(pDC) --> memory DC with same attributes as device DC
- SelectObject() selects bitmap into DC
  - Subsequent copying from memory DC is fast since data sequence is same as on the device

Bit Block Transfer in Windows

- pDC->BitBlt (x, y, w, h, &dcMem,
  xsrc, ysrc, dwRop)
  - Copies pixels from bitmap in source DC (dcMem) to destination DC (pDC)
  - x,y: upper left hand corner of destination rectangle
  - w,h: width, height of rectangle to be copied
  - xsrc, ysrc – upper left hand corner of source bitmap
  - dwRop -- raster operation for copy

Raster Ops

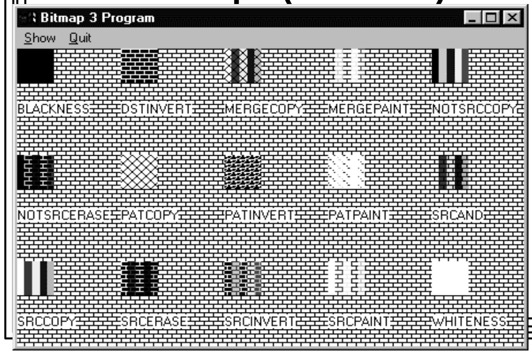
- How source pixel colors combine with current pixel colors
- Boolean logic combinations (AND, NOT, OR, XOR, etc.)
  - Currently-selected brush pattern also can be combined
  - So 256 different possible combinations
  - 15 are named
- Useful for special effects

Named Raster Ops

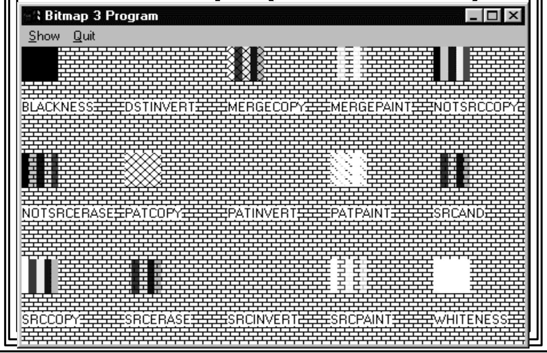
- (S=source bitmap, D=destination, P=currently-selected brush, i.e., the current Pattern)

```
BLACKNESS  0 (all black)   DSTINVERT   ~D
MERGECOPY  P & S            MERGEPAINT  ~S | D
NOTSRCCOPY ~S               NOTSRCERASE ~(S | D)
PATCOPY    P               PATINVERT   P ^ D
PATPAINT   (~S | P) | D    SRCAND      S & D
SRCCOPY    S                SRCERASE    S & ~D
SRCINVERT  S ^ D           SRCPAINT    S | D
WHITENESS  1 (all white)
```

Raster Ops (first time)



Raster Ops (second time)



StretchBlt()

- Same as BitBlt() except size of copied bitmap can be changed
- Source & destination width/height given

  pDC->StretchBlt (x,y,w,h,&dcMem, xsrc,ysrc,wsrc,hsrc,RasterOp);

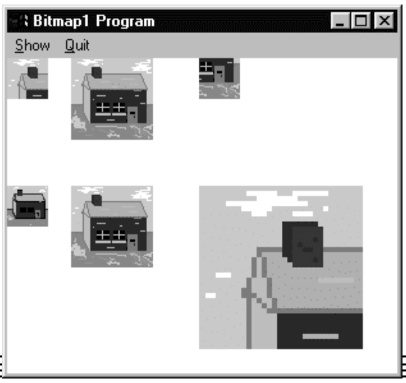
PatBlt()

- pDC->PatBlt(x,y,w,,h,dwRop);
  - Paints a bit pattern on specified DC
  - Pattern is a combination of currently-selected brush and pattern already on destination DC
  - x,y,w,h determine rectangular area
  - dwRop (raster op) specifies how pattern combines with destination pixels:

    BLACKNESS (0), DSTINVERT (~D), PATCOPY (P), PATINVERT (P^D), WHITENESS (1)
  - Pattern is tiled across specified area

Examples of BitBlt & StretchBlt

```
CBitmap bmpHouse; CDC dcMem;
BITMAP bm; int w,h;
bmpHouse.LoadBitmap(IDB_HOUSE);
bmpHouse.GetBitmap(&bm);
w = bm.bmWidth; h = bm.bmHeight;
dcMem.CreateCompatibleDC(pDC);
CBitmap* pbmpOld= dcMem.SelectObject(&bmpHouse);
pDC->BitBlt(0, 0, w/2, h/2, &dcMem, 0, 0, SRCCOPY);
pDC->BitBlt(50, 0, w, h, &dcMem, 0, 0, SRCCOPY);
pDC->BitBlt(150, 0, w/2, h/2, &dcMem, w/2, h/2, SRCCOPY);
pDC->StretchBlt(0,100,w/2,h/2,&dcMem,0,0,w,h,SRCCOPY);
pDC->StretchBlt(50,100,w,h,&dcMem,0,0,w,h,SRCCOPY);
pDC->StretchBlt(150,100,2*w,2*h,&dcMem,0,0,w/2,h/2,SRCCOPY);
dcMem.SelectObject(pbmpOld);
```



Animated Graphics

Notes from CS-360 Web Pages

Course Notes:

Class 4 -- Windows Bitmaps,Animation, and Timers

http://www.cs.binghamton.edu/~reckert/360/class4a.htm

Sample Programs:

Example 4-3: API Bouncing Ball Animation using PeekMessage()

http://www.cs.binghamton.edu/~reckert/360/ball_cpp.htm

Example 4-4: API Bouncing Ball Animation with Bitblt() to Preserve Background

http://www.cs.binghamton.edu/~reckert/360/ballblt_cpp.htm

Example 4-5: API Bouncing Ball Animation using a Timer

http://www.cs.binghamton.edu/~reckert/360/balltime_cpp.htm

Example 4-6: MFC Bouncing Ball Animation Using a Timer

http://www.cs.binghamton.edu/~reckert/360/mfcballtime_cpp.htm

Animated Graphics

- Creating a moving picture
  - Give illusion of motion by continual draw/erase/redraw
  - If done fast, eye perceives moving image
- In a single-user (DOS) application, we could do the following:

```
Do Forever{
  /* compute new location of object */
  /* erase old object image */
  /* draw object at new location */  }
```

- In Windows, other programs can't run while this loop is executing
- Need to keep giving control back to Windows so other programs can operate
- Two methods:
  - Use PeekMessage() Loop (for Win32 API)
    - Override OnIdle() (for MFC)
  - Use a Windows Timer

PeekMessage() vs. GetMessage()

- ***GetMessage()*** only returns control if a message is waiting for calling program
- ***PeekMessage()*** returns (with 0 value) if no active messages are in the system
- ***PeekMessage()*** loop can take action (redraw image) if ***PeekMessage()*** returns 0
- ***PeekMessage()*** doesn't return zero for WM_QUIT (like ***GetMessage()***)
  - So App must explicitly check for a WM_QUIT message to exit the program

- ***PeekMessage(lpMsg, hWnd, uFilterFirst, uFilterLast, wRemove);***
- The first 4 parameters are same as GetMessage.
- Last one: specifies whether message should be removed from the Queue

PeekMessage() message loop

```
while (TRUE)    // Do forever
{
 if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE) )
    { // non-zero (TRUE) means we must handle msg
    if (msg.message = = WM_QUIT)
       return (int)msg.wParam;
    else{
         TranslateMessage (&msg);
         DispatchMessage (&msg); }
    }
 else // { zero (idle); do other stuff - draw next animation frame }
}
```

BALL Example Animation App

- **A Win32 API -- (Ball bouncing off walls)**
- BALL.H
  - Define menu item constants
  - Define ball constants: (VELOCITY, BALLRAD, MINRAD)
- BALL.CPP
  - Global variables:
    - _dDrawOn: toggle animation on or off
    - _nXSize,_nYSize: window width, height--to determine if ball is inside
    - if window is resized, we need to change these

WM_SIZE message

- Sent anytime window is resized by the user
- Vertical/horizontal size of client area encoded in lParam
  - Least significant two bytes = horizontal size in pixels
  - Most significant two bytes = vertical size
- Helper function *DrawBall()*
  - Draws ball in new position for each new frame
  - Called each time PeekMessage() returns 0

MFC OnIdle() Virtual Function

- In CWinThread::Run() there is code like:

```
Do forever
    while {!::PeekMessage(…)}
        if (!OnIdle (lIdleCount++) )
            bIdle=FALSE;
    PumpMessage loop
```

- OnIdle() is called whenever program is idle
- So a derived app class can override OnIdle() to enact its own idle-processing functionality

- Parameter lCount: # of times OnIdle() has been called since last message was processed
  - Framework does its processing when lCount is 0 or 1
- Return value: non-zero means OnIdle() will be called again if message queue is still empty
- So structure the override as follows:

```
BOOL CMyApp::OnIdle(LONG lCount)
{   CWinApp::OnIdle (lCount);
    if (lCount = = 2)
        // Do our own idle processing (i.e., next fame of anim ation)
    return TRUE  }
```

  - Note call to base class function so framework can do necessary idle processing first

Timers -- Another Way to do Windows Animation

- An input device that notifies an app when a time interval has elapsed
  - Application tells Windows the interval
  - Windows sends WM_TIMER message each time interval elapses

Using a Timer

- Allocate and set a timer with:

  SetTimer (hWnd, timerID, interval, NULL);
    - Win32 API

  CWnd::SetTimer (timerID, interval, NULL);
    - MFC
  - Interval in milliseconds
  - Last parameter the address of a “timer procedure” that will receive the WM_TIMER messages
    - NULL means message goes to application’s queue
    - I.e., to application’s WndProc()

- From that point on, timer repeatedly generates WM_TIMER messages and resets itself each time it times out
  - Could be used to signal drawing the next frame of an animation!!!
- WM_TIMER handler: OnTimer(timerID)
- When app is done using a timer, stop timer messages and remove it with:
  - KillTimer(timerID);

BALLTIME MFC Application

- Same as BALL, but using MFC Doc/View
- All the action in CView derived class
- Menu item “Ball On/Off” handler: OnShow()
  - Toggles BOOL m_bDrawOn
- WM_CREATE handler: OnCreate()
  - Calls SetTimer() to start timer
- WM_SIZE handler: OnSize()
  - Sets m_nXSize, m_nYSize
- WM_DESTROY handler: OnDestroy()
  - Calls KillTimer() to stop timer

- WM_TIMER handler: OnTimer()
  - Calls helper function DrawBall() to draw ball in new position if m_bDrawOn is TRUE
    - Gets a pointer to a DC using GetDC()
    - Constructs a broad white pen for exterior of ball using CPen
    - Constructs a red brush for interior of ball using CBrush
    - pDC->SelectObject() to select pen an brush into DC
    - PDC->Ellipse() to draw ball in new position
    - pDC->SelectObject() to select pen/brush out of DC

Disadvantages to Using Timers

- WM_TIMER message are very low priority
- Fastest: 18 times per second (55 msec.)

Drawing on a Memory Bitmap (Improving an Animation)

- If many objects are drawn during each frame of an animation, we get flicker
  - Because of multiple accesses to frame buffer during each frame
- Best way to eliminate flicker:
  - Just one access to frame buffer per frame
  - Use off-screen memory bitmaps
  - This is double buffering

Drawing on Off-screen Bitmaps

- Use GDI functions to "draw" on a bitmap selected into a memory DC
- Just like using a "real" DC
  - So we can do many drawing operations
- When done, BitBlt() result to real DC
  - Only one access to fame buffer, so no flicker in animations

Getting a Bitmap to Draw on

- Create a blank bitmap in memory with:

CBitmap::CreateCompatibleBitmap (pDC, w, h);

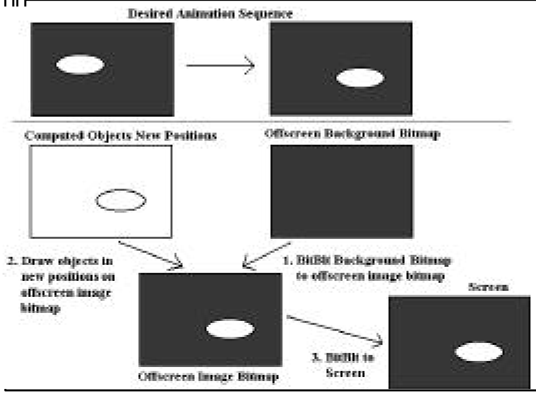
  - An alternative to LoadBitmap()
- After selected into a memory DC, use GDI graphics functions to draw on it without affecting real device screen
  - All the GDI drawing operations are now invisible to the user

---

- When drawing is all done, BitBlt() it to real device
  - so just one screen access
  - No flicker
    - (drawing directly to screen device context ==> many accesses to screen which produces flicker for complex images)

Animation of moving objects over a stationary background

- Set up an offscreen image bitmap and select it into a memory DC
- Set up an offscreen background bitmap and select it into another memory DC
- For each frame (each timer timeout):
  - Calculate new positions of objects
  - BitBlt() background bitmap to the offscreen image bitmap
  - Redraw objects (in new positions) on the offscreen image bitmap
  - BitBlt() entire offscreen image bitmap to screen


Desired Animation Sequence
Computed Objects New Positions
Offscreen Background Bitmap
2. Draw objects in new positions on offscreen image bitmap
1. BitBlt Background Bitmap to offscreen image bitmap
Screen
3. BitBlt to Screen
Offscreen Image Bitmap

- For a large image field, this BitBlt() covers a large area
  - could be too slow
- Better method: compute affected area
  - (rectangle encompassing old and new object positions)
- BitBlt() to that area only

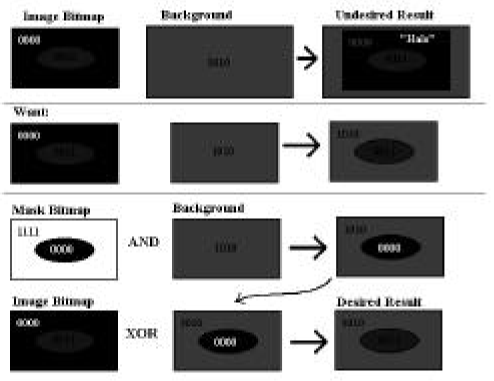
Sprites

- Little bitmaps that move on screen
- Frequently used in game programs
- Could restore background and just BitBlt() the sprite over it
- But there's a problem
  - sprite consists of desired image enclosed in a rectangle
  - so when blitting is done, background color inside enclosing rectangle will wipe out the background area on destination bitmap
  - moving object will have a "halo" around it
  - will also always have a rectangular shape

Solution (Sprite Animation)

1. Set up a "mask bitmap" in which sprite pixels are black and rest of enclosing rectangle is white
2. BitBlt() this over background using SRCAND (AND) raster op
3. Set up an "image bitmap" in which sprite pixels are set to image colors they should be (whatever colors are in the sprite object) and rest of enclosing rectangle pixels are black
4. BitBlt() this to the result of step 2 using the SRCINVERT (XOR) raster op

    Result will make sprite move to its new location with the background around it intact



DirectX and Windows Game Programming

- Game Programming
  - No "good" (fast) Windows games before 1995
  - Only DOS games for PCs--
    - Direct access to video memory permitted
    - Fast

Windows GDI

- Device independence
- Useful but slow functions
- No access to video hardware
- ==> High-speed games almost impossible

Demanding Requirements for High-resolution Animated Graphics

- 640 X 480 X 256-color ✍
  - 307,200 bytes
  - If background changes in each frame (e.g., flight simulator)
  - 307K bytes must be moved to screen
  - At least 15 times a second
  - Almost 5 Megabytes/second

- Also sprites move on a background
- Must be transferred one-by-one to screen memory too
- To avoid flicker--
  - Compose Scene in memory
  - Then transfer it to screen
    - twice as much data must be moved
  - GDI BitBlt() and StrecthBlt() can't cope with this task in real time

Microsoft Remedy (1995): the "Game SDK" (DirectX)

- A series of components called "COM objects"
  - Component Object Model
  - COM: an object-oriented interface
  - Creating/using A COM interface closely resembles creating/using a C++ class

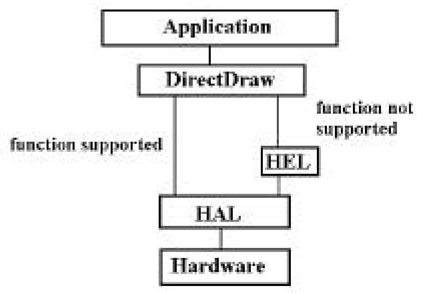
Some DirectX Components

- Direct Draw
- DirectSound
- DirectInput
- DirectPlay
- Direct3D

DirectDraw

- Provides direct control over the computer's video hardware
- Enables programs to quickly transfer graphics between memory and screen
- Takes advantage of hardware capabilities of the video card
- Capabilities not available on video card are emulated in software

DirectDraw Hardware Access



DirectSound

- Provides device-independent way of directly accessing computer's sound card
- Enables programmer to add sound effects and music to games
- Can synchronize sound effects with events occurring on the screen
- Can handle 3D sound effects

DirectInput

- Provides for easy use of joystick/game controller devices
- Done in a device-independent way

DirectPlay

- Provides for implementation of multi-user games over a network or modem
- Transport-independent, protocol-independent, on-line-service-independent
- Allows Windows games to communicate with each other

Direct3D

- Provides optimized three-dimensional capabilities to Windows games
  - polygon modeling, 3D transformations, projections, clipping, surface properties, lighting, texturing, shadowing, hidden surface removal, animation
- Takes advantage of 3D acceleration hardware, if available
- No additional coding required of the game developer

DirectX Games Run Under Windows

- Benefit from all the built-in Windows functionality
- Can use:
  - GDI graphics functions
  - All Windows user interface capabilities
  - All of fonts and other standard Windows drawing objects
  - In general, the entire Windows Win32 API

DirectDraw

- Main purpose:
  - To provide directly -accessible drawing "surfaces" in memory
  - To transfer drawing surfaces quickly to screen
- A surface:
  - A block of memory used for drawing
  - Separate surfaces used to hold each sprite in an animated scene
  - Another surface used to hold the background
  - Surfaces are composed into a final image
  - Which is transferred to the primary screen surface

Steps in Using DirectDraw in a Windows Program

- (Check the online help for details on the use of each DirectDraw function)

1. Call DirectDrawCreate() to create a DirectDraw object
2. Call the DirectDraw object's SetCooperativeLevel() member function==>

   Get control over screen and restrict access by other applications
3. Call DirectDraw object's SetDisplayMode() member function==>

   Set screen's resolution and color depth
4. Call DirectDraw object's CreateSurface() member ftn==>

   Create a primary surface object + one or more secondary surfaces (back buffers)
5. Call the primary DirectDrawSurface object's GetAttachedSurface() member function==>

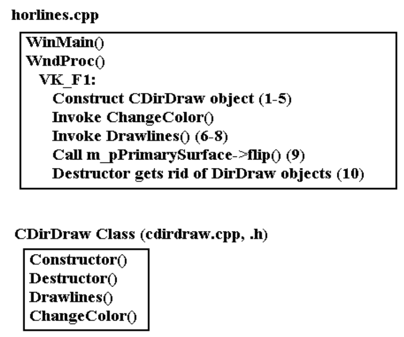
   Get a pointer to a back buffer

6. Call the back buffer’s DirectDrawSurface object's Lock() member function✍
   – Get a pointer to the back buffer surface's memory
7. Draw an image on the back buffer
   – **Access its memory directly**
8. Call the back buffer’s DirectDrawSurface object's Unlock() member function==>
   – Tell DirectDraw that the program is done with the back buffer

9. Call the primary DirectDrawSurface object's Flip() member function✍
   • Swap surface memory associated with the primary surface and that of the next back buffer surface, thus displaying the newly-drawn image
10. When terminating the application, all direct draw objects should be removed by calling their Release() member functions

The lineminimum DirectDraw Example Application

✍ Creates a 640 X 480 X 8-bit-color primary surface
✍ Draws 256 horizontal lines (using the current palette) on a back buffer surface
✍ Flips surfaces so the lines are displayed on screen
✍ A Win32 API program that has no menu
✍ Drawing action occurs in response to the user pressing the <F1> keyboard key



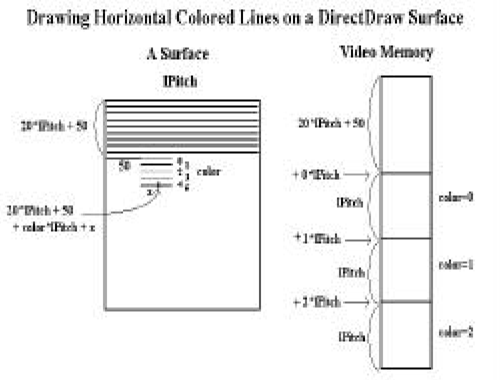
horlines.cpp

```
WinMain()
WndProc()
  VK_F1:
    Construct CDirDraw object (1-5)
    Invoke ChangeColor()
    Invoke Drawlines() (6-8)
    Call m_pPrimarySurface->flip() (9)
    Destructor gets rid of DirDraw objects (10)
```

CDirDraw Class (cdirdraw.cpp, .h)

```
Constructor()
Destructor()
Drawlines()
ChangeColor()
```

✍ The Program’s CDirDraw class:
   – Specification in cdirdraw .h
   – Implementation in cdirdraw .cpp
      • Does most of the work in this example
      • Defines a pointer to a DirectDraw object and two pointers to DirectDraw surfaces
      • Its constructor performs steps 1 through 5 (above)
      • Its destructor performs step 10
      • Member functions ChangeColor() & DrawLines() do rest (steps 6 through 8)



- <ESC> key terminates the application
- Application keeps track of/displays time required to draw lines on back buffer
- And time required for the surface switch
  - Uses GDI TextOut() to do the display
- Program uses DirectDraw member functions in the simplest ways possible
- A robust application would do extensive error checking after most of function calls (Refer to the references.)

DirectX and lineminimum Details

- See following CS-360 Web Pages:
- Course Notes
- Class 4x--DirectX and Windows Game Programming

http://www.cs.binghamton.edu/~reckert/360/class4x.htm

- Sample Programs
- Example 4-6: lineminimum DirectX Example

http://www.cs.binghamton.edu/~reckert/360/horlines_cpp.htm
http://www.cs.binghamton.edu/~reckert/360/cdirdraw_cpp.htm
http://www.cs.binghamton.edu/~reckert/360/cdirdraw_h.htm