Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering
and Applied Science

EngiNet™

WARNING

CS 460/560

Computer Graphics

Professor Richard Eckert

Lecture # 13

March 19, 2001

Lecture 13

A. Raster Methods for Transformations
B. Windows BitBlitting
C. Animation
D. Window to Viewport Transformation
E. Windows Mapping Modes

Raster Methods for 2-D Geometric (Translations)

- See CS-460/560 Notes:
  http://www.cs.binghamton.edu/~reckert/460/bitmaps.htm
- Can translate image in frame buffer by:
  - Embed it in a rectangle
  - Copy bit pattern of each pixel in rectangle to a destination rectangle
  - A simple double loop

- Assume coordinates of opposite corners of embedding rectangle are (x1,y1) and (x2,y2)
- Want to translate enclosed image by (tx,ty)
- Following pseudocode will do the job:

```
for (x=x1 to x2)
  for (y=y1 to y2)
    {
    color = GetPixel(x,y);  //get pixel color
    SetPixel(x+tx, y+ty, color);  //paint pixel
    }
```

- Known as a Bit Block Transfer (BiT BLT, or "blitting")
- Can also be used to move offscreen images to screen
- Can be very fast if we have direct access to the frame buffer
- Windows API uses Device Dependent Bitmaps to achieve effect of Bit Blitting
  - Also provides functions to stretch bitmaps

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 w/ paint pgm
- 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) & 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

- 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_IMG BITMAP image.bmp

– easier to: "Insert | Resource | Bitmap | Import"

B. Instantiate a CBitmap object

CBitmap bmp1;

C. Load it from the pgm's resources:

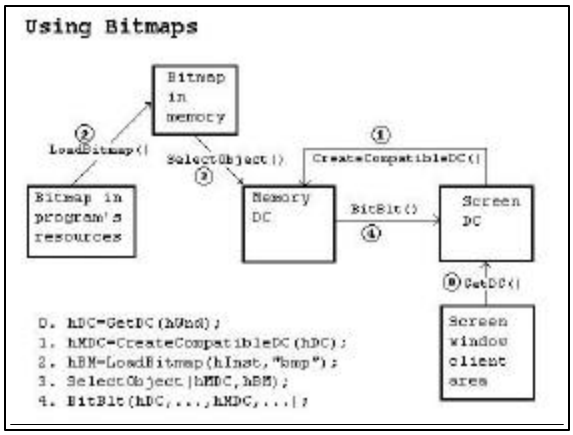
bmp1.LoadBitmap(IDB_IMG);

- 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



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
  – copying from memory DC is fast since data sequence is same as on the device

Blitting 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

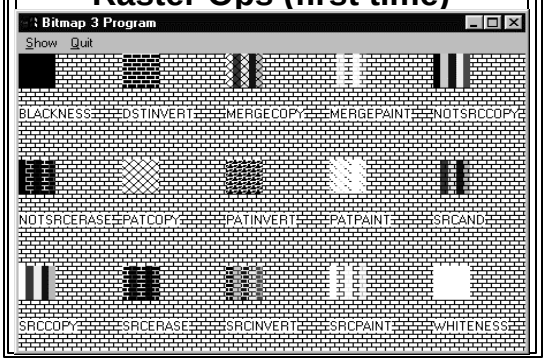
- How source pixel colors combine with current pixel colors
- Boolean logic combinations (AND, NOT, OR, XOR, etc.)
  - Currently-selected brush pattern also can be combined
  - so 256 different possible combinations
  - 15 are named
- Useful for special effects

Named Raster Ops

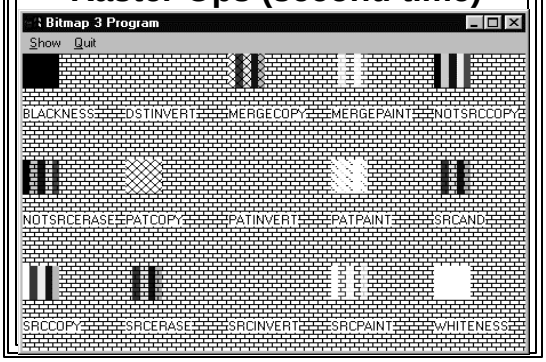
- (S=source bitmap, D=destination, P=currently-selected brush, i.e., the current Pattern)

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);

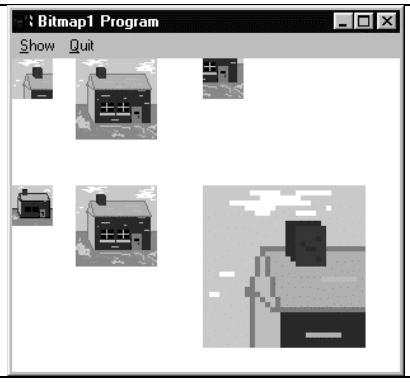
PatBlt()

- 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.GetObject(sizeof(BITMAP), &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: Bouncing Ball Animation using PeekMessage() (ball.cpp)
http://www.cs.binghamton.edu/~reckert/360/ball_cpp.htm
Example 4-4: Bouncing Ball Animation with Bitblt() to Preserve Background (ballblt.cpp)
http://www.cs.binghamton.edu/~reckert/360/ballblt_cpp.htm
Example 4-5: Bouncing Ball Animation using a Timer (balltime.cpp)
http://www.cs.binghamton.edu/~reckert/360/balltime_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
- One method:
  - Use a Windows Timer

Timers -- One 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:

  HWnd::SetTimer(timerID, interval, NULL);
  - Interval in milliseconds

- 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);

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
  - Windows: use off-screen memory bitmaps

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
  - Fast, so no flicker in animations

Getting a Bitmap to Draw on

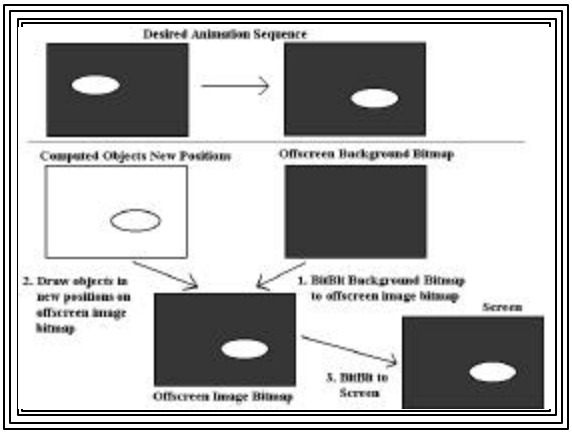
- Create a blank bitmap in memory with:

  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
      --> 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
- 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



- 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

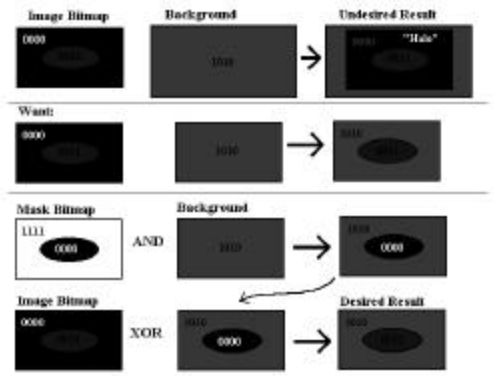
Sprites

- 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

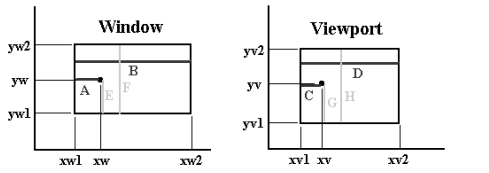
   Result will make sprite move to its new location with the background around it intact



2-D Viewing Transformation

- Convert from Window Coordinates to Viewport Coordinates
- (xw, yw) --> (xv,yv)
- Maps a world coordinate window to a screen coordinate viewport
- Window defined by: (xw1,yw1), (xw2,yw2)
- Viewport defined by: (xv1,yv1), (xv2,yv2)
- Basic idea is to maintain proportionality

Window to Viewport Transformation



$A/B = C/D$

$$\frac{xw-xw1}{xw2-xw1} = \frac{xv-xv1}{xv2-xv1}$$

$$xv=(Wv/Ww)*xw + xv1-(Wv/Ww)*xw1$$

where:
Ww=xw2-xw1 (window width)
Wv=xv2-xv1 (viewport width)

$E/F = G/H$

$$\frac{yw-yw1}{yw2-yw1} = \frac{yv-yv1}{yv2-yv1}$$

$$yv=(Hv/Hw)*yw + yv1-(Hv/Hw)*yw1$$

where:
Hw=yw2-yw1 (window height)
Hv=yv2-yv1 (viewport height)

Viewing Transformation Matrix

- Can cast into homogeneous matrix form:

$$Pv = \begin{vmatrix} xv \\ yv \\ 1 \end{vmatrix} \quad Pw = \begin{vmatrix} xw \\ yw \\ 1 \end{vmatrix} \quad Pv = Tv * Pw$$

$$Tv = \begin{vmatrix} C1 & 0 & C2 \\ 0 & C3 & C4 \\ 0 & 0 & 1 \end{vmatrix}$$

$C1 = Wv/Ww, \quad C2 = xv1 - (Wv/Ww)*xw1$

$C3 = Hv/Hw, \quad C4 = yv1 - (Hv/Hw)*yw1$

Be Careful with y-axis down

- y equation will be different!