MFC Graphics

WM_PAINT messages are sent when the window needs to be updated:
restores, exposes, or when
`CWnd::Invalidate()` is called to force repainting

WM_PAINT is mapped to the window's OnPaint() method.

Application painting is done inside a window's OnPaint() method.

Basic painting process:

get a DC, device context

set the current foreground, background, pen, brush, and font objects

make draw requests

Device Contexts

CDC encapsulates the device context, how to draw

CDC	handler	usage
CPaintDC	OnPaint	drawing windows client area draw on WM_PAINT message
CClientDC	any	drawing windows client area ! w/ OnPaint() draw w/ other event handler (mouse keyboard)
CWindowDC	any	draw non-client area, border, title bar
CMetaFileDC	any	draw to GDI metafile (display lists)

```
CPaintDC dc(this);
```

create dc as local variables in handler functions, "run time stack"
automatic dc delete (memory reclaimed) when function loses scope.

dc are a GDI resource, you can run out.

MFC Graphics

CDC has Set* and Get* methods for manipulating dc attributes

`GetTextColor, SetBkColor, SetROP2`

CDC specifies the drawing mode with ROP2 (Raster Operation To)

ROP2 specifies graphics function for drawing the requested pixel on the current pixel

default `R2_COPYPEN` replaces current pixel with new pixel

`R2_XORPEN` dest = src XOR dest (used with animations, cursors)

CDC SelectObject can set the current pen, brush, and font objects

CPen objects are used for drawing

CBrush objects are for painting, filling

CFont objects are for text

Drawing a rectangle uses the current pen to draw the border and brush to fill with the current drawing mode.

CDC has a mapping mode to translate logical coordinates to device coordinates.

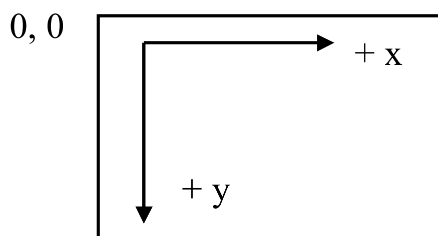
default mapping mode is MM_TEXT 1 unit == 1 pixel

other defined modes and user programmable modes.

Window and Viewport origins are in the upper left corner.

Both can be changed (usually do only one) SetWindowOrg(..)

```
// move origin to center of viewport
CRect rect;    // CRect defines rectangle objects
GetClientRect (*rect);  // get window rect values
dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);
```



Drawing Requests

MoveTo sets current drawing position
LineTo draws line from current position to specified position

```
dc.MoveTo(0,0);
dc.LineTo(0, 100);
```

PolyLine draws connected line segments between specified points

```
POINT aPoint[5] = {0,0, 0,100, 100,100, 0,100, 0, 0};
dc.PolyLine(aPoint, 5);
```

Arc draws an arc, PolyBezier draws a Bezier curve ...
Chord, Ellipse, Pie, Polygon, Rectangle, RoundRect ....

Pen

CPen pen(int style, int width, COLORREF color);
 style PS_SOLID, PS_DASH, PS_NULL ...
 width in units (default pixel)
 color an RGB value

```
CPen pen (PS_DASHDOT, 4, RGB(0, 128, 128));
```


Color

Windows uses a 24 bit pixel (true color) with 256 levels of RGB
 higher numbers are brighter (more saturated w/ color).

Color	R	G	B
back	0	0	0
light gray	192	192	192
cyan	0	192	0
magenta	192	0	0
yellow	192	192	0
bright yellow	255	255	0

RGB macro creates a single COLORREF structure from 3 rgb values.
COLORREF is a 32 bit representation of a color

```
const COLORREF  myColors[3] = {
   RGB (255,   255,  0),      //    yellow
   RGB (255,   0,    0),      //    red
   RGB (0,     255,  255)};   //    cyan
```

Brush


```
CBrush brush (COLORREF  color);
CBrush brush (int HatchStyle, COLORREF color);
   HS_DIAGCROSS, HS_CROSS, HS_DIAGCROSS, HS_VERTICAL,...
CBrush brush (CBITMAP * bitmap);
```

Text

DrawText(LPCTSTR *string,* int nChar, RECT &rect, int style)
  nChar count of characters in str, if -1 str must be null terminated
  style   DT_CENTER, DT_VCENTER format in rect

TextOut(int x, int y, CString str);
  TabbedTextOut(...) if str contains '\t' symbols

specify text alignment SetTextAlign(...) TA_RIGHT,

SetTextColor(...), SetBkColor(...)

Fonts


```
CFont font:
font.CreatePointFont( 140, "Arial");
```

Creates a CFont object for a 14 point Arial font.
Many ways to create fonts.

Point sizes are specified as multiples of 10. So there is a 14.1 point size (141).

Fonts are represented by the LOGFONT structure

True Type fonts are scalable and render well. (like postscript)
  Times New Roman, Arial, Courier New, Symbol (on all systems)

Raster fonts (i.e. MS Sans Serif) are designed for specific sizes.
  they do not scale and render well.

Stock Objects

Stock objects are pens, brushes, fonts, that are predefined in the API. There are many, for example:

```
NULL_PEN, BLACK_PEN, WHITE_PEN,
NULL_BRUSH, BLACK_BRUSH, GRAY_BRUSH,
SYSTEM_FONT, ANSI_FIXED_FONT,
```

For example, to draw a light gray circle w/o a border using stock objects

```
void CMainWindow::OnPaint() {
   CPaintDC dc(this);
   dc.SelectStockObject(NULL_PEN);
   dc.SelectStockObject(LTGRAY_BRUSH);
   dc.Ellipse(0, 0, 100, 100);
   }
```

Alternatively
create a pen, then select the pen
create a brush, then select the brush
draw the circle

Memory Mgmt & DCs

With many calls to OnPaint() and each call creating GDI objects (memory resources) possible memory utilization issues arise.

Simple Soln: Create all GDI objects on the stack
destructors implicitly declared at end of OnPaint() method scope.

Problems arise with **new**.
Objects created in the heap (new) exist past local method scope and must explicitly call their destructors with delete.

Mouse Events

Graphics windows often respond to mouse events.

Client Area Mouse Messages

Message	Message Map Macro	Handling function
`WM_?BUTTONDOWN`	`ON_WM_?BUTTONDOWN`	`On?ButtonDown`
`WM_?BUTTONUP`	`ON_WM_?BUTTONUP`	`On?ButttonUp`
`WM_?BUTTONDBLCLK`	`ON_WM_?BUTTONDBLCLK`	`On?ButtonDblClk`
`WM_MOUSEMOVE`	`ON_WM_MOUSEMOVE`	`OnMouseMove`

? == R M L for right, middle, or left button

Most window systems are two button mice (wheel mouse another issue)

Mouse message handlers are prototyped

```
afx_msg void OnMsgName(UINT nFlags, CPoint point);
void OnLButtonDown(UNIT flag, CPoint position) {....}
```

Nflags describes state of button press

```
MK_?BUTTON     (nFlags & MK_?BUTTON)   only mouse pressed
MK_CONTROL     (nFlags & MK_CONTROL)   control key + mouse
MK_SHIFT       (nFlags & MK_SHIFT)     shift key + mouse
```


CPoint class represents pixels

CPoint encapsulates the windows POINT typedef.
POINT has a long x and y value representing an x y coordinate

```
void CMainWindow::OnLButtonUp (UINT flags, CPoint point) {
   CClientDC dc (this);
   dc.SetTextColor(RGB(255, 0, 0));
   dc.TextOut (point.x, point.y, "Ouch!");
   }
```

The CPoint argument is always in pixels.
With other coordinate mapping use CDC::DPtoLP function

With OnMouseMove(...) point argument is the last point

There are equivalent nonclient-area mouse messages (mouse events in the title bar, frame, etc....

There are also `WM_MOUSELEAVE` and `WM_MOUSEHOVER` messages to use with `WM_MOUSEMOVE` for mouse leaves, hovers, and enters window.

Background processing

An application can engage in background tasks with a timer, idle processing or threads.

Threads are more complex and can have synchronization problems with the primary (user interface) thread. Threads useful for non UI tasks.

Background tasks enable a task to be performed and allow the application to respond to User Interaction.

When a handler function is processing, no User Interaction events can be responded to until that function returns control to main loop!

Window systems use idle time (no event) to handle housekeeping tasks such as resource management...

When using background processing always allow system to also use idle time (i.e. OnIdle() calls `CWinApp::OnIdle(...)` first).

Timer

Set a countdown timer to send a WM_TIMER message or call a function
Kill a set timer.

Timing not precise. Timers do not cascade (only one in event queue for each timerID.

```
SetTimer( UINT timerID, UINT msec, CALLBACK * lpfn)
SetTimer(1, 1000, NULL); // timer1  sends WM_TIMER  in 1 second
```

```
KillTimer( UINT timerID);
KillTimer(1);
```

NULL value for callback function sends WM_TIMER

Message Map macro ON_WM_TIMER () will map to handler
void OnTimer(UINT timerID) see Prosise pg 809-810 wrt P2

ptr to function as 3rd argument will call that function (see Prosise)

OnIdle

The event loop can detect when there are no events to process and send trigger an OnIdle message.

```
BOOL CMainWindow::OnIdle(LONG count) {
   CWinApp::OnIdle(count);  // call base OnIdle
   if (count == threshold) {
      // do simple, fast activity
      }
   if (continueIdleProcessing) return TRUE;
      else return FALSE;
   }
```

Need to call base OnIdle first so that windows can do its normal OnIdle housekeeping activities.

Argument count is the number of times OnIdle has been called since another event was dispatched.