Programming in Graphical Environment

Windows API Lecture 4

Paweł Aszklar
`P.Aszklar@mini.pw.edu.pl`

Faculty of Mathematics and Information Science
Warsaw Univeristy of Technology

Warsaw 2021

Graphics Device Interface

- Abstract interface for producing graphics and text on various media
- Drawing on: displays, bitmaps, printers, …
- Core system component from the earliest Windows versions
- Integrates well with message-driven GUI paradigm
- Stateful
  - Prefers modifying state before drawing over drawing function parameters
  - Simpler function calls, but harder to reason about
- Limited resource pools, difficult management make accidental leaks easier and more severe

Limitations:

- Hardware acceleration
  - Only for bit-block transfers
  - Far below Direct2D/DirectWrite capabilities
  - Still superior to GDI+ (which is entirely in software)
- Anti-aliasing only for text, bitmap stretching
- Transparency: Alpha blending available only for selected operations

Basic Types

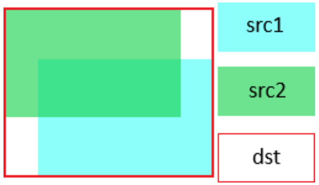
- COLORREF — RGB color
  - From lowest byte: blue, green, red channels
  - High byte unused (sometimes alpha channel)
  - RGB(r, g, b) — combine channel values
  - GetRValue(c), GetGValue(c), GetBValue(c) — extract
- POINT — 2D integer coordinate
- RECT — Upright (axis-aligned) rectangle
  - Coordinates: *X* of left and right, and *Y* of top and bottom edge
  - BOOL SetRectEmpty(RECT *rc) — all coordinates set to 0
  - BOOL SetRect(RECT* rc, int left, int top, int right, int bottom)
  - BOOL IsRectEmpty(const RECT *rc) — if width and height are 0
  - BOOL InflateRect(RECT *rc, int dx, int dy) — increase width by 2dx and height by 2dy (dx subtracted from left and added to right, dy subtracted from top and added to bottom coordinates)
  - BOOL OffsetRect(RECT *rc, int dx, int dy) — moves rectangle (dx added to left and right, dy added to top and bottom coordinates)
  - BOOL CopyRect(RECT *dst, const RECT *src) — copies coordinates

```
typedef DWORD COLORREF;
struct POINT {
    LONG x, y;
};
struct RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
};
```

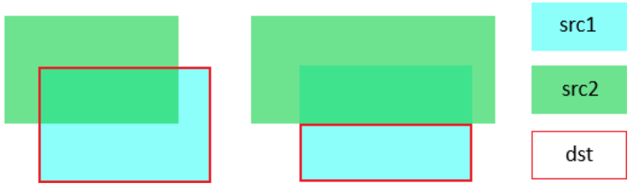
Basic Types

- `RECT` — Upright (axis-aligned) rectangle
  - `BOOL EqualRect(const RECT *rc1, const RECT rc2)` — checks if coordinates are equal
  - `BOOL PtInRect(const RECT *rc, POINT pt)` — checks if `pt` is inside `rc` (left, top egde or interior only, `rc` height and width must not be negative)
  - Bounding box of set intersection of rectangle areas: `BOOL IntersectRect(RECT *dst, const RECT *src1, const RECT *src2)`
  - Bounding box of set union of rectangle areas: `BOOL UnionRect(RECT *dst, const RECT src1, const RECT *src2)`
  - Bounding box of set difference of rectangle areas: `BOOL SubtractRect(RECT *dst, const RECT src1, const RECT *src2)`

Set Union:



Set Difference:



Device Context

- Core of GDI abstraction
- HDC — handle to opaque device context object
- Stores state, links drawing to particular surface
- Provides way to query capabilities of a device
- Device context types:
  - Display, printer — tied to a given device
  - Memory — allow drawing on bitmaps
  - Information context — context of a display, printer, which can retrieve device properties and capabilities, but cannot draw.

Obtaining Display Context

- Display context can be obtained for any window, primary display or entire virtual screen
- Usually retrieved from common pool
- Common contexts have state reset upon retrieval
- Private context can be requested for window/class, usually controlled by class style:
  - `CS_OWNDC` — each window has own private context
  - `CS_CLASSDC` — all windows of a class share a private context (should be avoided!)
- Obtained context can limit the area where drawing is visible, i.e. *clipping region* (depending on window styles and function used) to visible part of:
  - Window’s client area
  - Windows area including frame
  - `WS_CLIPCHILDREN` — additionally excludes areas of child windows
  - `WS_CLIPSIBLINGS` — additionally excludes areas covered by siblings drawn on top
- Child window can request to use parent’s context

Obtaining Display Context

- window’s client area: `HDC GetDC(HWND hWnd)` (pass `nullptr` for entire screen)
- window area (incl. frame): `HDC GetWindowDC(HWND hWnd)` (pass `nullptr` for primary display)
- window’s (client) area: `HDC GetDCEx(HWND hWnd, HRGN clip, DWORD flags)` (pass `nullptr` for entire screen), depending on `flags`:
  - `DCX_INTERSECTRGN`, `DCX_EXCLUDERGN` — drawing on visible area intersected with `clip`/with `clip` excluded
  - `DCX_CLIPCHILDREN`, `DCX_CLIPSIBLINGS` — as if corresponding class style was set
  - `DCX_PARENTCLIP` — context of parent (parent’s `CS_PARENTDC`, `WS_CLIPCHILDREN` ignored)
  - `DCX_CACHE` — common context (regardless of class styles)
  - `DCX_WINDOW` — entire window’s visible area instead of just client

Releasing context:

- Contexts acquired by above function released by `ReleaseDC`
- Common contexts need to be freed as soon as possible
- Private context don’t need to be released immediately (unless shared by whole class), but it’s recommended for consistency (they can always be retrieved again unchanged)

Creating Device Context

Display (any display, entire screen), printer context:

- `CreateDCW` — drawing context
- `CreateICW` — information context (no drawing)

Memory context:

- `HDC CreateCompatibleDC(HDC hdc)`
  - Context created with default attribute
  - Compatible with `hdc`'s device, but with default attributes
  - Bound to monochrome 1×1 bitmap (needs to be rebound)

Destroying contexts

- Functions above create context owned by calling thread
- Must be destroyed by calling `DestroyDC` when no longer needed
- Bitmap bound to memory context isn't released with it!

  (Although the default bitmap memory context is created with doesn't need releasing)

Device Context State

Context stores various states, drawing modes and bound (*selected*) object used for all relevant drawing operations:

- Positioning:
  - Current position — where certain drawing operations start
  - Transformations — map logical points to screen (world$\rightarrow$page$\rightarrow$device$\rightarrow$screen)
- Selected objects (one of each)
  - Pen, Brush, Font
  - optional: Palette, Clip Region, Path
  - memory context: Bitmap
- Modes:
  - graphics mode, layout, text alignment
  - drawing modes: polygon filling, arc direction
  - mixing modes: foreground, background, stretch
- Other properties:
  - colors: background, text, DC Pen, DC Brush (DC Pen and Brush not selected by default)
  - text spacing, brush origin (for non-solid brushes), miter limit (for line joints), …

Bitmaps

- Image stored as continuous binary data
- Additional information needed to interpret and display image data
- How to extract a pixel values:
  - Image resolution: width w, height h
  - Bits per pixel count bpp (usually 24 or 32bpp)
    (e.g. 4bpp – one byte describes two pixels; 24bpp — 3 bytes describe one pixel)

  Optionally:
  - Scan-line (row of pixels) byte width — not always $w * bpp$ because of alignment requirements
  - Compression type — image data might need to be decompressed before accessing pixels
  - Row order — bottom-up (default) or top-down
- How to interpret pixel values (pixel format):
  - Indexed colors — values indicate an index in a color table
  - RGB colors — value is a bitfield of three channel intensities
- How to reproduce the image (optional):
  - Intended physical dimensions
  - Color table (RGB values or indices in device's current palette)
  - Color profile image was created with, preferred color profile matching technique

Device-Dependent (Compatible) Bitmaps (DDB)

- Bottom-up, uncompressed
- Only describes how to extract pixel values
- Interpretation, reproduction depends on device context
- `HBITMAP CreateCompatibleBitmap(`
  `HDC hdc, int cx, int cy)`
  - Creates compatible bitmap of given resolution
  - *Bpp*, row alignment matches `hdc`’s surface
  - If `cx` or `cy` is 0, creates 1×1 monochrome bitmap (1*bpp*)
  - If `hdc` is a memory context and has a device-independent bitmap selected, device-independent bitmap (with the same attributes) is created instead.

```
struct BITMAP{
    LONG bmType; //always 0
    LONG bmWidth;  //cx
    LONG bmHeight; //cy
    LONG bmWidthBytes;
    WORD bmPlanes; //always 1
    WORD bmBitsPixel; //bpp
    LPVOID bmBits; //bits
};
```

- `HBITMAP CreateBitmap(int cx, iny cy, UINT planes, UINT bpp, const void *bits)`
  `HBITMAP CreateBitmapIndirect(const BITMAP *bmp)`
  - As above, but *bpp* specified directly, row always aligned to 2 bytes
  - If `bits` not `nullptr`, must point to bitmap data (including row padding)

Device-Independent Bitmaps (DIB)

- Attributes described by bitmap header (Note! Header doesn't point to pixel data):
  `BITMAPCOREHEADER`, `BITMAPINFOHEADER`, `BITMAPV4HEADER`, `BITMAPV5HEADER`
- Negative height indicated top-down bitmap
- Variable-length color table follows header immediately, if it is needed
  Note! Check docs to see: when needed, required size and layout!
- In *packed* bitmaps, pixel data immediately follows header (and color table, if present)

-
  ```
  HBITMAP CreateDIBSection(HDC hdc, const BITMAPINFO *info, UINT usage,
                           void **pbits, HANDLE hSection, DWORD offset)
  ```
  - `info` — despite stated type, can point to memory containing header of any type followed by color table (if needed)
  - `usage` — contents of color table: `DIB_RGB_COLORS` for RGB values; `DIB_PAL_COLORS` for `WORD` indices into `hdc` current palette (rarely used).
  - `handle`, `offset` — handle to and offset into memory-mapped bitmap file, pass `nullptr` to allocate new bitmap instead
  - `pbits` — output parameter, receives pointer to pixel data (can be `nullptr`)
- `GetDIBits`, `SetDIBits` — Device-Dependent to/from Device-Independent Bitmap conversion

Device-Independent Bitmap Headers

```
struct BITMAPHEADER { /*Note: exact field names and types vary between header structs*/
    /*BITMAPCOREHEADER - basic pixel data layout*/
    DWORD size;                  // Header struct size in bytes
    LONG  width, height;         // Image width and height (WORD in CORE header, LONG in others)
    WORD  planes;                // Number of color planes (always 1)
    WORD  bits;                  // Bits per pixel
    /*BITMAPINFOHEADER - pixel data interpretation parameters*/
    DWORD compression;           // Compression type (BI_RGB - uncompressed)
    DWORD imagesize;             // Pixel data size, can be 0 if uncompressed
    LONG  xppm, yppm;            // Pixels per meter (for physical size)
    DWORD ncolours;              // Number of entries in color table (can be 0 if color table unused)
    DWORD importantcolours;      // Number of significant color table entries (can be 0)
    /*BITMAPV4HEADER - color profile attributes (ICM 1.0)*/
    DWORD rMask, bMask, gMask, aMask; // Channel masks (BI_BITFIELDS compression)
    DWORD colorSpaceType;        // Indicates if Color Space is provided
    CIEXYZTRIPLE endpoints;      // 2.30 Fixed-point CIEXYZ coordinates of RGB primary colors
    DWORD gammaR, gammaG, gammaB;     // 16.16 Fixed-point gamma coefficients
    /*BITMAPV5HEADER - additional/alternative color profile attributes (ICM 2.0)*/
    DWORD intent;                // Intended color space conversion method
    DWORD profileData;           // Offset in bytes to color profile data
    DWORD profileSize;           // Size in bytes of color profile data
    DWORD reserved;              // Unused, always 0
};
```

Palettes

- Array of colors that can drawn/displayed on a device
- Most devices don’t support palettes any more.
- Used mostly for memory contexts operating on bitmaps with indexed colors
- Creating logical palette: `CreatePalette`
- Modification: `ResizePalette`, `SetPaletteEntries`
- Applying palette to context: `SelectPalette`→`RealizePalette`
- If realized palette is modified: `UnrealizeObject`→`RealizePalette`
- Freeing palette: `DeleteObject`

Brushes

- Used to fill interiors of closed figures: polygons, ellipses, paths, ...
- Represent a pattern used for filling
- Pattern is repeated (tiled)
- Tiling origin defined by context's brush origin: `SetBrushOrgEx`, `GetBrushOrgEx`

  Note: that means pattern will not move if object is drawn in different position
- Brush origin in device coordinates (default: (0,0), i.e. top-left corner of drawing area)
- Pattern position and size will not change with context's coordinate mapping/transformations
- Obtaining stock brushes: `GetStockObject`
  - `WHITE_BRUSH`, `LTGRAY_BRUSH`, `GRAY_BRUSH`, `DKGRAY_BRUSH`, `BLACK_BRUSH` — grayscale, solid
  - `DC_BRUSH` — solid brush, uses context's current DC brush color
    `GetDCBrushColor`, `SetDCBrushColor`, can be changed while selected
  - `NULL_BRUSH` — draws nothing
- Obtaining stock system color brushes: `GetSysColorBrush`
  - any symbolic constant with `COLOR_` prefix
  - colors used by system for drawing different parts of a window

Brushes

- Creating solid brush — fills with constant color: `HBRUSH CreateSolidBrush(COLORREF color)`
- Creating hatch pattern brush — fills with tiling hatches
  - Type: `HS_HORIZONTAL`, `HS_VERTICAL`, `HS_FDIAGONAL`, `HS_BDIAGONAL`, `HS_CROSS`, `HS_DIAGCROSS`
  - Hatches use constant color, gaps use background (depends background mixing mode)
  - `HBRUSH CreateHatchBrush(int hatch, COLORREF color)`
- Creating bitmap pattern brush — fills with tiling bitmap
  - `HBRUSH CreatePatternBrush(HBITMAP bmp)` — from DDB or DIB handle
  - `HBRUSH CreateDIBPatternBrushPt(const void *packedDIB, int usage)`:

`packedDIB`	pointer to *packed* device-independent bitmap
`usage`	color table type (see `CreateDIBSection` here)

- `HBRUSH CreateBrushIndirect(const LOGBRUSH *br)`

`lbStyle`	`lbHatch`	`lbColor`	type
`BS_NULL`	ignored	ignored	empty brush
`BS_SOLID`	ignored	`color`	solid brush
`BS_HATCHED`	`hatch`	`color`	hatch pattern
`BS_PATTERN`	`bmp`	ignored	bitmap pattern
`BS_DIBPATTERNPT`	`packedDIB`	`usage`	bitmap pattern

```
struct LOGBRUSH {
    UINT       lbStyle;
    COLORREF   lbColor;
    ULONG_PTR  lbHatch;
};
```

- Delete brush: `DeleteObject` (not necessary for stock brushes, but not harmful either)

Pens

- Used for drawing lines, curves, outlines of filled shapes
- Attributes:
  - Width
  - Brush (sometimes only color — equivalent to using solid brush)
  - Join and end cap styles
  - Dash pattern
- Simple pens: `CreatePen`, `CreatePenIndirect`
- Extended — cosmetic and geometric pens: `ExtCreatePen`
- Stock pens: `GetStockObject`
  - `WHITE_PEN`, `BLACK_PEN` — solid white/black cosmetic pen
  - `DC_PEN` — solid cosmetic pen, uses context's current DC pen color `GetDCPenColor`, `SetDCPenColor`, can be changed while selected
  - `NULL_PEN` — draws nothing

Simple Pens

```
HPEN CreatePen(int lopnStyle, int lopnWidth,
               COLORREF lopnColor)
HPEN CreatePenIndirect(LOGPEN *pen)
```

```
struct LOGPEN{
    UINT     lopnStyle;
    POINT    lopnWidth; //y unused
    COLORREF lopnColor;
};
```

- `lopnWidth`
  - pen width in world units
  - effective width (in pixels) depends on all transformations
  - if 0, effective width always $1px$
- `lopnStyle` — line style, one of:
  - PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT
  - if effective width $> 1px$ pen always solid (transformations change pen's appearance)
  - PS_NULL — draws nothing
  - PS_INSIDEFRAME — solid pen, entire width inside the shape (only some closed figures)
- `lopnColor` — pen color
- Simple pens have round caps and joins

Cosmetic and Geometric Pens

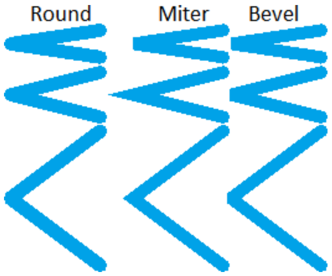
```
HPEN ExtCreatePen(DWORD style, DWORD width, const LOGBRUSH *brush,
                  DWORD dashCount, const DWORD *dashes)
```

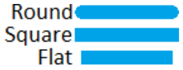
- `style` — combination of:
  - pen type – PS_COSMETIC or PS_GEOMETRIC
  - line style — one of simple pen styles, PS_ALTERNATE (draws every other pixel) or PS_USERSTYLE (user defined dash style)
  - join style (geometric pens only) — one of:

PS_JOIN_ROUND	round
PS_JOIN_MITER	sharp (mitered) if within context's miter limit, otherwise beveled
PS_JOIN_BEVEL	flat (beveled)

  - cap style (geometric pens only) — one of:

PS_ENDCAP_ROUND	round
PS_ENDCAP_SQUARE	square (extended half the width past the end)
PS_ENDCAP_FLAT	flat





Cosmetic and Geometric Pens

```
HPEN ExtCreatePen(DWORD style, DWORD width, const LOGBRUSH *brush,
                  DWORD dashCount, const DWORD *dashes)
```

- `width`:
  - Geometric — pen width in world units (undergoes transformations), must be > 0
  - Cosmetic — must be 1, effective width always 1*px*
- `brush`:
  - Geometric — describes brush pattern used to draw lines
  - Cosmetic — describes line color (i.e. `brush` must describe solid brush)
- `dashCount`, `dashes` — custom dash style array and it's count
  - Only for PS_USERSTYLE pens, otherwise both must be 0
  - First value — first dash length; second value — first space length, ...
  - Geometric — lengths in world units
  - Cosmetic — lengths in device dependant *style* units (unit length of 3*px* on my screen)
  - Max count 16, pattern repeats for even counts or is reversed for odd
- Extended pens ignore background color

  (Draw as if with transparent background mixing mode, regardless of actual mode of the context)

Pens — Summary

- Simple pens with 0 width almost like cosmetic extended pens, except:
  - Ones with dash pattern use context's background mixing mode for gaps (gaps always transparent for extended pens)
  - Must use solid color
- Simple pens with width ≥ 1 behave almost like extended geometric pens, except:
  - Dash pattern used only if effective width is 1 (geometric pens always use dash pattern)
  - Dash pattern uses context's background mixing mode for gaps (gaps always transparent for extended pens)
  - Must use solid color, can't change join and end cap styles
- Sharp joins appearance controlled by miter limit:
  - Miter length — distance between intersection of line walls on the inside and outside of a join
  - Miter limit — maximum ratio between miter length and pen width, above which join is beveled
  - `GetMiterLimit`, `SetMiterLimit` — check/set context's miter limit (default: `10.0`)
- Created pens need to be released: `DeleteObject` (not necessary for stock pens, but not harmful either)

Regions

- Represents arbitrary area
- Stored as set of axis-aligned rectangles
- All coordinates as 27-bit signed integers
- Referred to by HRGN handle
- When created, usually represent the interior of given shape
- When passed to a function, handle must be a valid region, even if it's used as output
- Contrary to other GDI objects, all region handles need to be destroyed (`DeleteObject`)
  Operations such as selecting a region into device context create copies instead of assuming ownership like with other objects

Creating Regions

- Rectangular Region:
  ```
  HRGN CreateRectRgn(int x1, int y1, int x2, int y2)
  HRGN CreateRectRgnIndirect(const RECT * rect)
  ```
  - `x1, y1` — Top-left corner
  - `x2, y2` – Bottom-right corner
  - `rect` — `RECT` structure specifying upper-left and lower-right corners
- Rounded Rectangle Region:
  ```
  HRGN CreateRoundedRectRgn(int x1, int y1, int x2, int y2, int w, int h)
  ```
  - `x1, y1` — Top-left corner
  - `x2, y2` — Bottom-left corner
  - `w, h` — Width and height of ellipse used to round the corners
- Elliptical Region:
  ```
  HRGN CreateEllipticRgn(int x1, int y1, int x2, int y2)
  HRGN CreateEllipticRgnIndirect(const RECT * rect)
  ```
  - `rect` — Bounding rectangle of the ellipse
  - `x1, y1` — Upper-left corner of ellipse’s bounding rectangle
  - `x2, y2` — Lower-left corner of ellipse’s bounding rectangle

Creating Regions

- Polygonal Region:

```
HRGN CreatePolygonRgn(const POINT * ptList, int ptCount, int mode);
HRGN CreatePolyPolygonRgn(const POINT * ptList, const INT * ptCounts, int polyC
```

  - `ptList` — array of vertex coordinates of the polygon(s)
  - `ptCount` — number of vertices in a polygon
  - `ptCounts` — array with number of vertices in each polygon (`ptList` contains flat list of points, last vertex of a polygon is immediately followed by first vertex of the next)
  - `mode` — Fill mode:

`ALTERNATE`	alternate mode (odd-even)
`WINDING`	winding mode (non-zero winding value)

    See slides below

Recreating Regions

```
DWORD GetRegionData(HRGN rgn, DWORD size,
                    RGNDATA * data)
```

- `rgn` — region handle
- `size` — size of `data` buffer in bytes
- `data` — output buffer for region data
- If `data` is `nullptr`, returns required `data` buffer size
- If function fails (e.g. `size` to small) returns 0
- Otherwise returns `size`

```
HRGN ExtCreateRegion(const XFORM * mtx,
                     DWORD size,
                     const RGNDATA * data)
```

- `mtx` — region transformation (see slides below)
- `size` — size of `data` buffer in bytes
- `data` — region data

```
struct RGNDATA {
    struct RGNDATAHEADER {
        //header size in bytes
        DWORD dwSize;
        //must be RDH_RECTANGLES
        DWORD iType;
        //number of rectangle
        DWORD nCount;
        //size of Buffer
        DWORD nRgnSize;
        //bounding rectangle
        RECT  rcBound;
    } rdh;
    char Buffer[];
};
```

Region Operations

- Comparing regions: `BOOL EqualRgn(HRGN rgn1, HRGN rgn2)`
- Replace with rectangular region (`rgn` must be valid):

  `BOOL SetRect(HRGN rgn, int x1, int y1, int x2, int y2)`
- Combining regions:

  `int CombineRgn(HRGN dst, HRGN src1, HRGN src2, int mode)`
  - `dst` — must already exist, area replaced with te result
  - `mode`:

RGN_COPY	Copy of `src1`
RGN_OR	Set union (`src1`∩`src2`)
RGN_AND	Set intersection (`src1`∪`src2`)
RGN_DIFF	Set difference (`src1`\\`src2`)
RGN_XOR	Set symmetric difference ((`src1`\\`src2`)∪(`src2`\\`src1`))

- Move region area: `int OffsetRgn(HRGN rgn, int x, int y)`
- Retrieve region bounding box: `int GetRgnBox(HRGN rgn, RECT * rc)`
- Hit-testing: `BOOL PtInRegion(HRGN rgn, int x, int y)`

  `BOOL RectInRegion(HRGN rgh, const RECT *rc)`

Paths

Fonts

Coordinate Spaces

- World space
- Page space
- Device (Context) space
- Physical Device space

World to Page Space Transformations

Page to Device Space Transformation

Device to Physical Device Transformation

Clipping Regions

- System Region
  - Window rectangle (CreateWindow,SetWindowPos,GetWindowPos,etc.)
  - Window region (SetWindowRgn,GetWindowRgn,GetWindowRgnBox) - don't set on windows with any frame (caption bar, border)
  - Window visibility (Minimized, WS_CLIPCHILDREN,WS_CLIPSIBLINGS)
  - Client area (WM_PAINT, WM_ERASEBKGND)
  - Update region (InvalidateRect, InvalidateRgn, ValidateRect, ValidateRgn, GetUpdateRect, GetUpdateRgn, ExcludeUpdateRgn)
- Meta region
  - SetMetaRgn (calculates intersection clip/existing meta, replaces meta, clears clip, no way to expand w/o resetting DC), GetMetaRgn
- Clip region: ExtSelectClipRgn, GetClipRgn, SelectClipRgn(same-ish as SelectObject w/ region), SelectClipPath, OffsetClipRgn, ExcludeClipRect, IntersetClipRect, GetClipBox
- GetRandomRgn - Random access to System (4, SYSRGN); Meta (2); Clip (1); and *API* (3, clip$\cap$meta) regions

When to Draw

- Parts of a window need redrawing when it or other windows move/resize/change z-order/etc.
- Windows mark for update any such region
- InvalidateRect, InvalidateRgn mark for update (e.g. redrawing entire window when resizing, otherwise only new part repainted)
- WM_PAINT generated if update region not empty (low priority)
- Force immediate repaint w/ RedrawWindow,UpdateWindow
- prevent w/ ValidateRect,ValidateRgn
- Paint anytime w/ GetDC, GetWindowDC, GetDCEx - might cause fragmentation of painting logic
- other messages that might affect painting: WM_SYSCOLORCHANGE, WM_DISPLAYCHANGE

WM_PAINT

- BeginPaint - sends WM_NCPAINT, obtains DC for client area$\cap$update region (conceptually: GetDCEx(hwnd, GetUpdateRgn, DCX_INTERSECTRGN)), sends WM_ERASEBKGND, fills PAINTSTRUCT, validates update entire region (preventing duplicated WM_PAINTS),hides caret
- EndPaint - releases dc, restores caret (if it was hidden)

WM_NCPAINT

- Sent when window frame needs repainting
- wParam is update region (always rectangle)
- `GetDCEx(hwnd, wParam, DCX_WINDOW|DCX_INTERSECTRGN)`
- Pass to DefWindowProc, YMMV with painting on window frame of top-level windows

WM_ERASEBKGND

- Indicates window’s background needs repainting
- wParam is HDC
- if handled return 1 or 0 to indicate background was erased (fErase of PAINTSTRUCT)
- DefWindowProc will erase with class background brush (hbrBackground) if it’s not null
- Set when registering class, SetClassLongPtr, GetClassLongPtr - either assign a brush or system color constant incremented by 1 (COLOR_XXX + 1)

Lines and Curves

using DC Current Position

- MoveToEx, GetCurrentPositionEx
- AngleArc, ArcTo, LineTo, PolyLineTo, PolyBezierTo, PolyDraw
- use current pen (SelectObject - returns old pen, either restore it - preferable - or release/destroy) for outline
- Simple dashed pens: GetBkMode, SetBkMode (OPAQUE - GetBkColor, SetBkColor; TRANSPARENT)
- All pens: foreground mixing mode (GetROP2, SetROP2) - many different bitwise operations between 1, 0, source (pen), destination (screen) colours using NOT, AND, OR, XOR
- GetMiterLimit, SetMiterLimit
- implicit starting point at current position, afterwards current position moved to the last point of the shape.
- shape not filled

Lines and Curves

Ignoring Current Position

- Arc,PolyBezier,Polyline,PolyPolyline
- Starting point provided explicitly
- Current position doesn’t change
- otherwise same as -To variants
- StrokePath

Closed Figures

- Rectangle,RoundRect,Ellipse,Chord,Pie,Polygon,PolyPolygon
- StrokeAndFillPath - will close any open figure
- outline w/ current pen (see prev. slide for DC params)
- filled w/ current brush (SelectObject - same as for pen; GetBrushOrgEx, SetBrushOrgEx)
- current position not modified
- fill mode for self-intersecting boundary or shapes w/ holes (GetPolyFillMode, SetPolyFillMode):
  - alternate - pixel filled if half-line from it in any direction crosses shape boundary odd number of times
  - winding - accounts for drawing direction for each part of the boundary. Each time half-line cast from the point is intersected by the boundary going clockwise add 1, counter-clockwise subtract 1. Fill pixels with non-zero winding value.

Filling

- PatBlt - fill/combine rectangle w/ current brush
- FillPath - fill path (closing opened figures; StrokeandFillPath, but w/o outline)
- PaintRgn, FillRgn - fill region w/ current or supplied brush
- FrameRgn - paint region outline of given thickness w/ supplied brush
- InvertRgn - invert color bits within region
- FloodFill, ExtFloodFill
- GdiGradientFill

GradientFill exisst and is equivalent to Gdi- variant, but defined in msimg32.lib instead of gdi32.lib

Block Transfer

- BitBlt - copy/combine rectangle from source to destination DC w/o scaling
- MaskBlt - copy/combine rectangle from source to destination DC w/o scaling, w/ a mask
- StretchBlt - copy/combine rectangle from source to rectangle (possibly of different size) in destination DC (allows for scaling) - GetStretchBltMode, SetStretchBltMode
- StretchDIBits - same as above, but a source is device-independent bitmap
- GdiTransparentBlt - copy rectangle from source to rectangle (possibly of different size) in destination DC (allows for scaling, but no mirroring) treating specified color in source as transparent
- PlgBlt - copy rectangle from source into parallelogram in destination DC w/ optional mask (allows for scaling and shearing)

AlphaBlend, TransparentBlt exist and are equivalent to Gdi- variants, but defined in msimg32.lib instead of gdi32.lib

Text

- TextOutW, ExtTextOutW, DrawTextW, DrawTextExW
- GetTextColor, SetTextColor, GetBkColor, SetBkColor, GetTextAlign, SetTextAlign, GetTextCharacterExtra, SetTextCharacterExtra, GetTextExtentPoint32W, GetTextMetricsW, SetTextJustification
- GetGraphicsMode, SetGraphicsMode - under *advanced mode* vector/truetype fonts fully transformed

Flicker-Free Drawing

- Avoid flickering when drawing by double-buffering
- Block background erasure (set class background brush to null or intercept WM_ERASEBKGND)
- When painting (hdc - client area device context; width, height - client rectangle size):

```
//Create in-memory buffer and associated device context
HDC memDC = CreateCompatibleDC(hdc);
HBITMAP memBmp = CreateCompatibleBitmap(hdc, width, height);
HBITMAP oldBmp = reinterpret_cast<HBITMAP>(SelectObject(memDC, memBmp));

... //Fill background and draw on memDC

//Clean-up
BitBlt(hdc, 0, 0, width, height, memDC, 0, 0, SRCCOPY);
DeleteObject(SelectObject(memDC, oldBmp));
DeleteDC(memDC);
```

Device Context Attributes

Table of device state default values

End of Windows API Lecture 4

Thank you for listening! ☺