Programming in Graphical Environment

Windows API Lecture 4

Paweł Aszklar
pawel.aszklar@pw.edu.pl

Faculty of Mathematics and Information Science
Warsaw Univeristy of Technology

Warsaw 2024

Graphics Device Interface

- Abstract interface for producing graphics and text on: displays, bitmaps, printers, …
- Core system component, 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

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.

Device Context State

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

- Selected objects (one of each type)
  - Pen, Brush, Font
  - optional: Palette, Clip Region, Path
  - for memory context: Bitmap
- Modes:
  - graphics mode, layout, text alignment
  - drawing modes: polygon filling, arc direction
  - mixing modes: foreground, background, stretch
- Positioning:
  - Current position — where certain drawing operations start
  - Transformations — map logical points to screen (world→page→device→screen)
- Other properties:
  - colors used for: background, text, DC Pen, DC Brush
  - brush origin, pen miter limit, arc direction, text spacing, LTR/RTL layout, halftone stretching color adjustment …

Obtaining Display Device Context

- Device context can be obtained for any window, primary display or entire virtual screen.
- Usually reused contexts from a common pool with state reset upon retrieval.
- Private context obtained only if window's class styles includes:
  - `CS_OWNDC` — each window has its own private context, or
  - `CS_CLASSDC` — all windows of a class share a private context (should be avoided!)
- Context's *visible region*, drawings outside are invisible, e.g.:
  - Window's client area
  - Entire window including frame
  - `WS_CLIPCHILDREN` — for parent, excludes areas covered by children
  - `WS_CLIPSIBLINGS` — for child, excludes areas covered by siblings
- 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` — *visible region* intersected with/excludes `clip`
  - `DCX_CLIPCHILDREN`, `DCX_CLIPSIBLINGS` — as if `WS_CLIPCHILDREN`, `WS_CLIPSIBLINGS` styles were used
  - `DCX_PARENTCLIP` — uses parent's visible region (similar to `CS_PARENTDC`)
  - `DCX_CACHE` — common context (overrides `CS_OWNDC`, `CS_CLASSDC`)
  - `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)

Where to Draw

- Parts of a window need to be redrawn, e.g. when windows move/resize/change z-order.
- Any such areas are automatically marked for update
- Manually mark parts of client area with (changes are cumulative):

```
BOOL InvalidateRect(HWND hWnd, const RECT *rc, BOOL erase)
BOOL InvalidateRgn (HWND hWnd, HRGN rgn, BOOL erase)
```

  - Pass `nullptr` as `rc`/`rgn` to mark the whole client area
  - `erase` controls if background should be erased
- Manually unmark parts of client area:

```
BOOL ValidateRect(HWND hWnd, const RECT *rc)
BOOL ValidateRgn (HWND hWnd, HRGN rgn)
```

  Passing `nullptr` as `rc`/`rgn` validates the whole client area
- Check current update region or its bounding box:

```
BOOL GetUpdateRect(HWND hWnd, LPRECT rc, BOOL erase)
```

  Pass `nullptr` as `rc` to just check if it's not empty

```
int GetUpdateRgn(HWND hWnd, HRGN rgn, BOOL erase)
```

When to Draw

- You can draw anytime and anywhere w/ `GetDC`, `GetWindowDC`, `GetDCEx`, however,
- Drawing should generally be done in response to messages:
  - `WM_PAINT` generated by message queue if update region not empty (low priority)
  - `WM_NCPAINT` sent if any part of window frame needs to be redrawn
  - `WM_ERASEBKGND` sent if any part of client area background needs to be erased
- Beware of fragmentation of painting logic!
- If you need to paint something immediately, invalidate an area (see previous slide), then send `WM_PAINT` message with:

  ```
  BOOL UpdateWindow(HWND hwnd)
  ```
- Related messages that might affect painting: any window positioning message, `WM_SYSCOLORCHANGE`, `WM_DISPLAYCHANGE`, `WM_DPICHANGED`, `WM_DWMCOLORIZATIONCOLORCHANGED`

When and Where to Draw

```
BOOL RedrawWindow(HWND hWnd, const RECT *rc, HRGN rgn, UINT flags)
```

- Offers functionality of all `Invalidate-`, `Validate-`, `Update-` functions and more
- `rc` or `rgn` (if used) specify the part of a window affected, only one can be non-`nullptr`
- if both are `nullptr`, entire window is affected
- `flags` control the behavior:
  - `RDW_INVALIDATE` — invalidate affected client area
  - `RDW_ERASE` — also mark the area for erasure (must be used w/ `RDW_INVALIDATE`)
  - `RDW_FRAME` — also invalidate affected non-client area (must be used w/ `RDW_INVALIDATE`)
  - `RDW_VALIDATE` — validate affected client area
  - `RDW_NOFRAME` — also suppress pending `WM_NCPAINT` messages (must be used w/ `RDW_VALIDATE`)
  - `RDW_NOERASE` — suppress pending `WM_ERASEBKGND` messages
  - `RDW_ERASENOW` — immediately sends pending `WM_NCPAINT`, `WM_ERASEBKGND`
  - `RDW_UPDATENOW` — immediately sends pending `WM_PAINT`
  - `RDW_ALLCHILDREN`, `RDW_NOCHILDREN` — control if child windows are included in the operation
  - `RDW_INTERNALPAINT`, `RDW_NOINTERNALPAINT` — control the internal flag, that causes `WM_PAINT` to be pending even if invalid area is empty.

WM_PAINT

- `HDC BeginPaint(HWND hWnd, LPPAINTSTRUCT ps)`
  - Should be called before any painting
  - Can only be called in response to `WM_PAINT`
  - Sends `WM_NCPAINT`, `WM_ERASEBKGND` if they are still pending
  - Obtains device context for client area
  - Limits drawing to the update region and validates it
  - Hides caret (if one's present)

```
struct PAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
};
```

- Afterwards `PAINTSTRUCT` contains
  - `hdc` — device context that should be used for painting, the same that `BeginPaint` returns
  - `fErase` — `true` if attempts to erase background failed,
    e.g. window class's background brush was `nullptr` or custom `WM_ERASEBKGND` returned 0
  - `rcPaint` — bounding box of the update area
  - other fields are reserved for internal system use
- `BOOL EndPaint(HWND hWnd, const PAINTSTRUCT *ps)`
  - Must be called after drawing before the end of `WM_PAINT` handler
  - Releases the device context, restores caret (if it was hidden)

WM_ERASEBKGND

- Received when window's background needs repainting
- `wParam` is the device context, you should not release it here
- Return 1 or 0 to indicate if background was erased (`fErase` of `PAINTSTRUCT`)
- `DefWindowProcW` will erase with class background brush if it's not `nullptr`
  - Reminder: background brush `hbrBackground` is set when registering a class
  - Use `GetClassLongPtrW`, `SetClassLongPtrW` w/ `GCLP_HBRBACKGROUND` to retrieve or change it
  - Instead of `HBRUSH` handle, can be a color constant incremented by 1, e.g.: `reinterpret_cast<HBRUSH>(COLOR_WINDOW + 1)`
- If custom erasing needed, it's often more convenient to just return 0 and erase background in `WM_PAINT` handler.

WM_NCPAINT

- Received when window frame needs repainting
- `wParam` is update region (`HRGN` handle, always rectangular)
- To obtain device context and limit drawing to the update region, call:
  `GetDCEx(hwnd, reinterpret_cast<HRGN>(wParam), DCX_WINDOW | DCX_INTERSECTRGN)`
- Pass to `DefWindowProcW` if you want the regular frame to be drawn first
- Nowadays frame usually hidden for top-level windows, covered by a frame created by Desktop Window Manager

Flicker-Free Drawing

- All drawing operations are immediately reflected on the window
- If whole window is erased and repainted often, the area might flicker
- To avoid it, block background erasure
  (set class background brush to `nullptr`, intercept `WM_ERASEBKGND` and return 0)
- When painting use so called *double-buffering*
  (`hdc` - original device context; `x`, `y`, `width`, `height` - update area bounding box)

```
//Create in-memory buffer and an associated memory 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, offset positions if (x,y) not (0,0)
...

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

Basic Types

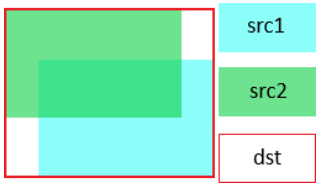
- `COLORREF` — RGB color
  - From lowest byte: blue, green, red channels
  - High byte unused (sometimes alpha channel)
  - `RGB(r, g, b)` — combine channel values
  - `GetRValue(c)`, `GetGValue(c)`, `GetBValue(c)` — extract
- `POINT` — 2D integer coordinate
- `RECT` — Upright (axis-aligned) rectangle
  - 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 2`dx` and height by 2`dy`
    (`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:

Lines and Curves

- Straight or curved line segments
- Outlined using selected pen (stock black pen by default)
- Shapes are not filled
- One set of functions uses context's current position
  - All `-To` functions, `PolyDraw`, `AngleArc`
  - Drawing starts at current position
  - Current position changed to shape's last point
  - Get current position:
    ```
    BOOL GetCurrentPositionEx(HDC hdc, LPPOINT ppt)
    ```
  - Set current position:
    ```
    BOOL MoveToEx(HDC hdc, int x, int y, LPPOINT ppt)
    ```
    `ppt` receives previous value, pass `nullptr` to ignore
- Other functions ignore current position entirely

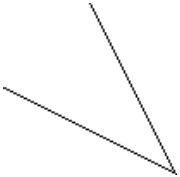
Lines To

- `BOOL LineTo(HDC hdc, int x, int y)`
  Line segment from current position to specified point.

```
MoveToEx(hdc, 50, 50, nullptr);
LineTo(hdc, 150, 100);
LineTo(hdc, 100, 0);
```

- `BOOL PolylineTo(HDC hdc, const POINT *apt, DWORD cpt)`
  Polyline from current position through `cpt` points from array `apt`.

```
POINT pts[2] = { {150, 100}, {100, 0} };
MoveToEx(hdc, 50, 50, nullptr);
PolylineTo(hdc, pts, 2);
```

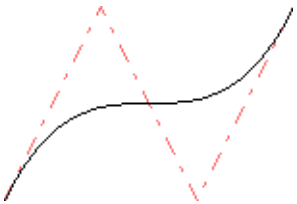


Curves To

`BOOL PolyBezierTo(HDC hdc, const POINT *apt, int cpt)`

- Draws *n* Bézier segments
- `cpt` must be $3n$, where *n* - number of Bézier curve segments
- Current position and first 3 points control first segment
- Last point of the previous segment and next 3 points control each subsequent one

```
auto oldp = SelectObject(hdc,
    CreatePen(PS_DASHDOT, 1, RGB(255, 100, 100)));
POINT pts[3] = { {50, 0}, {100, 100}, {150, 0} };
MoveToEx(hdc, 0, 100, nullptr);
PolylineTo(hdc, pts, 3);
DeleteObject(SelectObject(hdc, oldp));
MoveToEx(hdc, 0, 100, nullptr);
PolyBezierTo(hdc, pts, 3);
```



Lines and Curves To

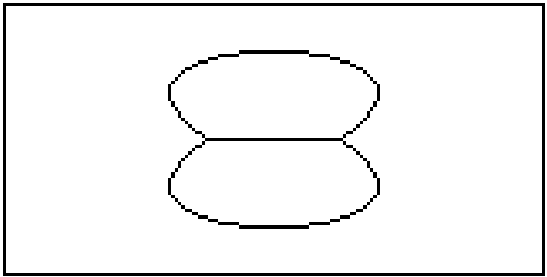
```
BOOL PolyDraw(HDC hdc, const POINT *apt, BYTE *aj, int cpt)
```

- Combines multiple `MoveToEx`, `PolylineTo` and `PolyBezierTo` calls
- `apt` stores point positions, `aj` their annotations, both with `cpt` elements
- Each point annotated as `PT_MOVETO`, `PT_LINETO`, or `PT_BEZIERTO`
- `PT_BEZIERTO` points must come in sequences of 3
- Combine with `PT_CLOSEFIGURE` to also draw a line segment from the point to the start of current shape — even if closed, shape is not filled.
- `PT_MOVETO` begins a new shape

Lines and Curves To

```
BOOL PolyDraw(HDC hdc, const POINT *apt, BYTE *aj, int cpt)
```

```
POINT pts[10] = {
    {160, 80},   {160,  0},   {  0,  0},              //Rectangle
    { 60, 40},   { 10,  5},   {150,  5},   {100, 40}, //Top curve
    { 10, 75},   {150, 75},   {100, 40} };            //Bottom curve
BYTE types[10] = {
    PT_LINETO,   PT_LINETO,   PT_LINETO | PT_CLOSEFIGURE,
    PT_MOVETO,   PT_BEZIERTO, PT_BEZIERTO, PT_BEZIERTO | PT_CLOSEFIGURE,
    PT_BEZIERTO, PT_BEZIERTO, PT_BEZIERTO
};
MoveToEx(hdc, 0, 80, nullptr);
PolyDraw(hdc, pts, types, 10);
```



Elliptic Arcs To

```
BOOL ArcTo(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
```

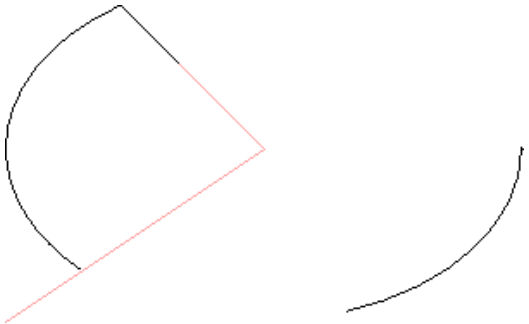
- Draws arc of ellipsis inscribed in $[\texttt{l}, \texttt{r}] \times [\texttt{t}, \texttt{b}]$ rectangle
- Delimited by two radials (half-lines from center) through $(\texttt{xr1}, \texttt{yr1})$ and $(\texttt{xr2}, \texttt{yr2})$
- Direction controlled by context's arc direction:
  - `int GetArcDirection(HDC hdc)` — to get current
  - `int SetArcDirection(HDC hdc, int dir)` — to change it (returns old direction)
  - can be: `AD_COUNTERCLOCKWISE` (default) or `AD_CLOCKWISE`
- Additional line drawn from context's current position to the start of the arc.

Elliptic Arcs To

```
BOOL ArcTo(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
```

```
//Draw radials of the first arc in red
auto old = SelectObject(hdc, CreatePen(PS_SOLID, 1, RGB(255, 160, 160)));
MoveToEx(hdc, 0, 200, nullptr);
LineTo(hdc, 150, 100);
LineTo(hdc, 100,  50);
DeleteObject(SelectObject(hdc, old));

//Draw first arc counter-clockwise
ArcTo(hdc,   0,  0, 300, 200,
          100, 50,   0, 200);
SetArcDirection(hdc, AD_CLOCKWISE);
MoveToEx(hdc, 300, 100, nullptr);
//Draw second arc clockwise
ArcTo(hdc,   0,   0, 300, 200,
          300, 100, 200, 200);
```

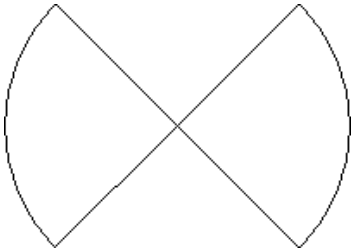


Circular Arcs To

`BOOL AngleArc(HDC hdc, int x, int y, DWORD r, FLOAT a_start, FLOAT a_sweep)`

- Draws circular arc centred at (x, y) with radius `r`
- `a_start` — angle in degrees counter-clockwise from circle's x-axis
- `a_sweep` — angle in degrees, determines arc length
- Ignores context's arc direction — negative angles for clockwise arcs
- Additional line drawn from context's current position to arc start

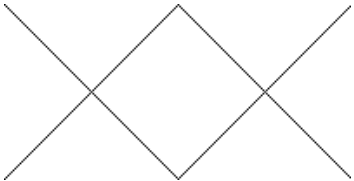
```
//Move to center
MoveToEx(hdc, 100, 100, nullptr);
//Line from center and left, CCW arc
AngleArc(hdc, 100, 100, 100, 135, 90);
//Line back to center
LineTo(hdc, 100, 100);
//Line from center and right, CW arc
AngleArc(hdc, 100, 100, 100, 45, -90);
//Line back to center
LineTo(hdc, 100, 100);
```



Lines and Curves From–To

- Context's current position isn't used and doesn't change
- `Polyline`, `PolyBezier` — same as `-To` variants, but with additional element at the beginning of arrays for the start point.
- `Arc` — Same as `ArcTo` but no line drawn to arc's starting point
- No counterparts to `LineTo`, `AngleArc`, `PolyDraw`
- `BOOL PolyPolyline(HDC hdc, const POINT *apt, const DWORD *asz, DWORD csz)`
  Draws a number of disjointed polylines.
  - `csz` — number of polylines
  - `asz` — number of points in each polyline (needs `csz` elements, each > 1)
  - `apt` — points forming all polylines (size must be the sum of `asz` values)

```
POINT pt[6] = { {0,   0}, {100, 100}, {200,   0},
                {0, 100}, {100,   0}, {200, 100} };
DWORD pl[2] = { 3, 3 };
PolyPolyline(hdc, pt, pl, 2);
```



Closed Figures

- Drawing:
  - polygons,
  - rectangles, rounded rectangles, ellipses
  - elliptic segments and sectors
- Shapes outlined with context's current pen
  (stock black pen by default, use `NULL_PEN` to omit)
- Interior filled with context's current brush
  (stock white brush by default, use `NULL_BRUSH` to omit)

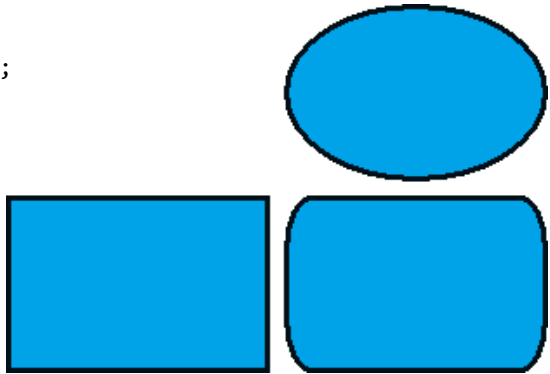
Rectangles, Ellipses

- `BOOL Rectangle(HDC hdc, int left, int top, int right, int bottom)`
  Draws a $[\texttt{left}, \texttt{right}] \times [\texttt{top}, \texttt{bottom}]$ rectangle
- `BOOL Ellipse(HDC hdc, int left, int top, int right, int bottom)`
  Draws an ellipse inscribed in a $[\texttt{left}, \texttt{right}] \times [\texttt{top}, \texttt{bottom}]$ rectangle
- `BOOL RoundRect(HDC hdc, int l, int t, int r, int b, int w, int h)`
  - Draws a $[\texttt{l}, \texttt{r}] \times [\texttt{t}, \texttt{b}]$ rectangle with rounded corners
  - Quarters of ellipse with height `h` and width `w` used for corners
  - `h` and `w` clamped to rectangle's width and height
  - `h` and `w` equal to rectangle's width and height results in an ellipse

Rectangles, Ellipses

```
BOOL Rectangle(HDC hdc, int left, int top, int right, int bottom)
BOOL Ellipse(HDC hdc, int left, int top, int right, int bottom)
BOOL RoundRect(HDC hdc, int l, int t, int r, int b, int w, int h)
```

```
auto oldbr = SelectObject(hdc, CreateSolidBrush(RGB(0, 162, 232)));
auto oldpn = SelectObject(hdc, CreatePen(PS_SOLID, 3, RGB(0, 16, 25)));
Rectangle(hdc,  10, 120, 160, 220);
Ellipse  (hdc, 170,  10, 320, 110);
RoundRect(hdc, 170, 120, 320, 220, 30, 60);
DeleteObject(SelectObject(hdc, oldpn));
DeleteObject(SelectObject(hdc, oldbr));
```



Elliptic Segments & Sectors

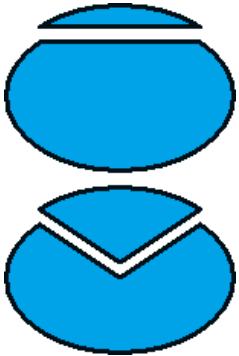
```
BOOL Chord(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
BOOL Pie(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
```

- Draw elliptical segment (`Chord`) or sector (`Pie`).
- Segment/Sector of ellipse inscribed in a $[\texttt{l}, \texttt{r}] \times [\texttt{t}, \texttt{b}]$ rectangle
- Arc delimited by intersections with two radials through $(\texttt{xr1}, \texttt{yr1})$ and $(\texttt{xr2}, \texttt{yr2})$
- Direction controlled by context's arc direction
- Arc start, length and direction work just like with `Arc` and `ArcTo`

Elliptic Segments & Sectors

```
BOOL Chord(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
BOOL Pie(HDC hdc, int l, int t, int r, int b, int xr1, int yr1, int xr2, int yr2)
```

```
//Top part of the image show segments (chords)
//Bottom part shows sectors (pies)
auto oldbr = SelectObject(hdc,
    CreateSolidBrush(RGB(0, 162, 232)));
auto oldpn = SelectObject(hdc,
    CreatePen(PS_SOLID, 3, RGB(0, 16, 25)));
//Top, CCW segment & sector
Chord(hdc,  10,  10, 160, 110, 160,  10,  10,  10);
Pie  (hdc,  10, 130, 160, 230, 160, 130,  10, 130);
SetArcDirection(hdc, AD_CLOCKWISE);
//Bottom, CW segment & sector
Chord(hdc,  10,  20, 160, 120, 160,  20,  10,  20);
Pie  (hdc,  10, 140, 160, 240, 160, 140,  10, 140);
DeleteObject(SelectObject(hdc, oldpn));
DeleteObject(SelectObject(hdc, oldbr));
```



Polygons

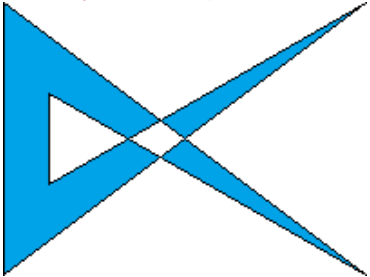
- `BOOL Polygon(HDC hdc, const POINT *apt, int cpt)`
  Draws a polygon with `cpt` corners stored in array `apt`
- `BOOL PolyPolygon(HDC hdc, const POINT *apt, const INT *asz, int csz)`
  - Draws `csz` polygons
  - `asz` contains number of vertices for each polygon (`csz` elements)
  - `apt` contains vertex positions for all polygons (size equal to sum of `asz` elements)

Polygons

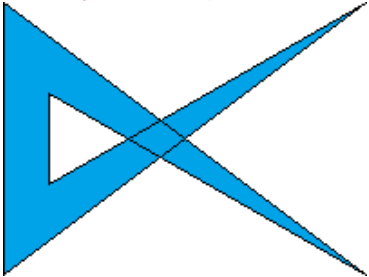
Interior of self-intersecting boundary or shapes w/ holes determined by context's fill mode:

- `int GetPolyFillMode(HDC hdc)` — check current fill mode
- `int SetPolyFillMode(HDC hdc, int mode)` — change it (returns previous)
- `ALTERNATE` — crossing outline flips from outside to inside and v.v.
- `WINDING` — influenced by direction of the outline. Crossing outline drawn relatively clockwise adds 1, counter-clockwise subtracts 1. Outside is 0, non-zero is inside.

`SetPolyFillMode(hdc, ALTERNATE)`



`SetPolyFillMode(hdc, WINDING)`



Polygons

All polygons drawn in one `PolyPolygon` call are treated as outline of a single shape

```
SelectObject(hdc, CreateSolidBrush(RGB(0, 162, 232)));
SelectObject(hdc, CreatePen(PS_SOLID, 3, RGB(0, 16, 25)));
POINT p1[3] = { {110,  10}, {210, 150}, { 10, 150} };
//Outer triangle filled blue
Polygon(hdc, p1, 3);
DeleteObject(SelectObject(hdc,
    CreateSolidBrush(RGB(162, 128, 242))));
POINT p2[10] = {
    {110, 30}, { 150, 86}, {70, 86}, //Small upper triangle
    {63, 98}, {158, 98}, {187, 138}, {33, 138}, //Lower trapezoid
    {110, 50}, {56, 126}, {164, 126} }; //Inner triangle
INT c2[3] = { 3, 4, 3 };
//Shape with three outlines filled purple
PolyPolygon(hdc, p2, c2, 3);
//Test swapping last two points in p2 w/ different fill modes
```

Drawing Paths

Paths store a collection of lines, curves and closed shapes, their creation is discussed below.

- `BOOL StrokePath(HDC hdc)`
  Outlines figures contained in current path with current pen
- `BOOL FillPath(HDC hdc)`
  - Closes any open figures in current path with a straight line segment.
  - Fills path interior with current brush and fill-mode
  - All figures treated as outline of one shape. (see `PolyPolygon` example above)
  - Path discarded afterwards!
- `BOOL StrokeAndFillPath(HDC hdc)`
  - Closes figures, fills and outlines them, then discards the path.
  - Same as `StrokePath` followed by `FillPath`, but the outline is drawn on top.

Drawing Regions

Regions represent an area (union of a bunch of rectangles), their creation is discussed below.

- `BOOL PaintRgn(HDC hdc, HRGN rgn)`
  `BOOL FillRgn (HDC hdc, HRGN rgn, HBRUSH brush)`
  Fill region w/ current or supplied brush
- `BOOL FrameRgn(HDC hdc, HRGN rgn, HBRUSH brush, int w, int h)`
  Outline region w/ supplied brush
  `w` and `h` specify width and height of vertical and horizontal brush strokes
- Similar function:
  `int FrameRect(HDC hdc, const RECT *prc, HBRUSH brush)`
  Outlines rectangle w/ supplied brush (outline thickness is 1)
- `BOOL InvertRgn(HDC hdc, HRGN rgn)`
  Bitwise invert colors within region

Filling

- `BOOL FloodFill(HDC hdc, int x, int y, COLORREF color)`
  `BOOL ExtFloodFill(HDC hdc, int x, int y, COLORREF color, UINT type)`
  Perform flood-fill (think: bucket tool from MS Paint) from point (x,y), using `color` as:
  - for `FloodFill` or if `type` is `FLOODFILLBORDER` — boundary blocking filling
  - if `type` is `FLOODFILLSURFACE` — color of the surface that should be filled
- `GdiGradientFill` — fills area with a gradient (see docs for this one)
  The same function is also available as `GradientFill` msimg32.dll (not linked by default)

Pattern Block Transfer

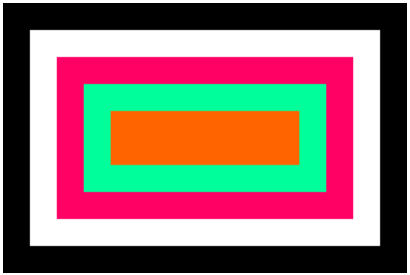
```
BOOL PatBlt(HDC hdc, int x, int y, int w, int h, DWORD rop)
```

- Fills a rectangle with top-left corner in (x,y) and size $w \times h$
- `rop` (binary raster-operation code) determines the result based on initial colors of pixels in the destination area and the current brush
  - `WHITENESS` — Fill white (more specifically, 0^{th} palette color)
  - `BLACKNESS` — Fill black (more specifically, 1^{st} palette color)
  - `DSTINVERT` — Bitwise inversion of existing colors
  - `PATCOPY` — Fill with context's current brush
  - `PATINVERT` — Bitwise XOR of existing colors and context's current brush

Pattern Block Transfer

```
BOOL PatBlt(HDC hdc, int x, int y, int w, int h, DWORD rop)
```

```
//Fills with (0,0,0)
PatBlt(hdc,  0,  0, 300, 200, BLACKNESS);
//Fills with (255,255,255)
PatBlt(hdc, 20, 20, 260, 160, WHITENESS);
auto old_br = SelectObject(hdc,
    CreateSolidBrush(RGB(255, 0, 100)));
//Fills with (255,0,100)
PatBlt(hdc, 40, 40, 220, 120, PATCOPY);
//Inverts to (0,255,155)
PatBlt(hdc, 60, 60, 180, 80, DSTINVERT);
DeleteObject(SelectObject(hdc,
    CreateSolidBrush(RGB(255, 155, 155))));
//XORs to (255,100,0)
PatBlt(hdc, 80, 80, 140, 40, PATINVERT);
DeleteObject(SelectObject(hdc, old_br));
```



Bit Block Transfer

```
BOOL BitBlt(HDC dst, int xd, int yd, int w, int h, HDC src, int xs, int ys, DWORD rop)
```

- Fills a rectangle in `dst` which top-left corner in $(\texttt{xd},\texttt{yd})$ and size $\texttt{w}\times\texttt{h}$
- Source area in `src` is a rectangle with top-left corner in $(\texttt{xs},\texttt{ys})$ and size $\texttt{w}\times\texttt{h}$
- `rop` (ternary raster-operation code) determines the result based on initial colors of pixels in the source and destination area, and the current brush:
  - values listed for `PatBlt` achieve the same result here (source area is ignored)
  - `SRCCOPY` — Copies the source area to the destination
  - `SRCAND` — Bitwise ANDs source and destination
  - `SRCPAINT` — Bitwise ORs source and destination
  - `SRCINVERT` — Bitwise XORS source and destination
  - `MERGECOPY` — Bitwise ANDs source and brush
  - More named codes in the docs
  - Not all codes have names, see appendix to see how they are constructed

Masked Block Transfer

```
BOOL MaskBlt(HDC      dst, int xd, int yd, int w, int h,
             HDC      src, int xs, int ys,
             HBITMAP mask, int xm, int ym, DWORD rop)
```

- Fills a rectangle in `dst` which top-left corner in $(\texttt{xd},\texttt{yd})$ and size $\texttt{w}\times\texttt{h}$
- Source area in `src` is a rectangle with top-left corner in $(\texttt{xs},\texttt{ys})$ and size $\texttt{w}\times\texttt{h}$
- `mask` is an optional monochrome (black-and-white) bitmap
- Mask area in `mask` is a rectangle with top-left corner in $(\texttt{xm},\texttt{ym})$ and size $\texttt{w}\times\texttt{h}$
- `rop` (quaternary raster-operation code) determines the result based on initial colors of pixels in the source, destination, and mask area, and the current brush:
  - use `MAKEROP4(fore, back)` to combine two ternary raster-operation codes
  - `fore` — operation code under the mask (where mask values are non-zero)
  - `back` — operation code outside of the mask
- If `mask` is `nullptr`, works as `BitBlt` with `fore` raster-operation code.

Stretched Block Transfer

```
BOOL StretchBlt(HDC dst, int xd, int yd, int wd, int hd,
                HDC src, int xs, int ys, int ws, int hs, DWORD rop)
```

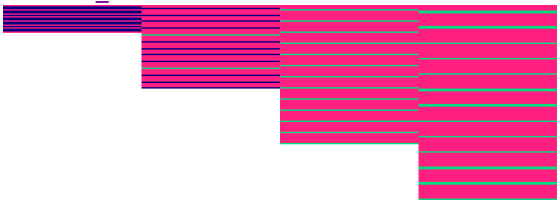
- Fills a rectangle in `dst` which top-left corner in (`xd`,`yd`) and size `wd`×`hd`
- Source area in `src` from (`xs`,`ys`) and size `ws`×`hs` stretched over the destination
- Enlarging always duplicates rows and/or columns of pixels
- Compressing controlled by `dst`'s stretching mode:
  - `int GetStretchBltMode(HDC hdc)` — check the current mode
  - `int SetStretchBltMode(HDC hdc, int mode)` — change it (returns previous)
  - `STRETCH_DELETESCANS` — removes some rows/columns
  - `STRETCH_ANDSCANS` — bitwise ANDs removed rows/columns with remaining ones
  - `STRETCH_ORSCANS` — bitwise ORs removed rows/columns with remaining ones
  - `STRETCH_HALFTONE` — resizes with averaging (often best result)
- `rop` determines the result (same as `BitBlt`), colors are mixed after the stretch

`StretchDIBits` — similar function for stretching a bitmap

Stretched Block Transfer

```
BOOL StretchBlt(HDC dst, int xd, int yd, int wd, int hd,
                HDC src, int xs, int ys, int ws, int hs, DWORD rop)
```

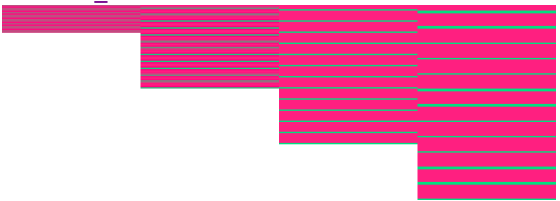
STRETCH_ANDSCANS



STRETCH_DELETESCANS

STRETCH_ORSCANS

STRETCH_HALFTONE



Parallelogram Block Transfer

```
BOOL PlgBlt(HDC dst, const POINT *dstPts,
            HDC src, int xs, int ys, int ws, int hs,
            HBITMAP mask, int xm, int ym)
```

- `dstPts` must contain 3 `POINT`s for upper-left, upper-right and lower-left corner of a parallelogram which is the destination area in `dst`.
- Source area in `src` is a rectangle with top-left corner in $(\texttt{xs},\texttt{ys})$ and size $\texttt{ws}\times\texttt{hs}$
- `mask` is an optional monochrome (black-and-white) bitmap
- Mask area in `mask` starts from $(\texttt{xm},\texttt{ym})$, if `mask` is to small to cover source area, it is repeated.
- Source area is stretched, compressed, sheared and/or rotated to fit the destination.
- No parameter for raster-operation code
- Pixels under `mask` (all pixels if `mask` is `nullptr`) overwrite the destination (compare to `MaskBlt`)
- Reshaping governed by `dst`'s current stretch mode (see `StretchBlt`)

Transparent Block Transfer

```
BOOL GdiTransparentBlt(HDC dst, int xd, int yd, int wd, int hd,
                       HDC src, int xs, int ys, int ws, int hs,
                       UINT crTransparent)
```

- First 10 parameters describe the source and destination areas, just like `StretchBlt`
- Stretches the source over the destination, just like `StretchBlt`
- No parameter for raster-operation code
- Pixels which color is different than `crTransparent` overwrite the destination
- Stretching controlled by `dst`'s stretch mode, but `STRETCH_ANDSCANS` and `STRETCH_ORSCANS` treated as `STRETCH_DELETESCANS`.
- Newer function, supports 32bpp colors, but *alpha* (opacity) value is simply copied over.

The same function is also available as `TransparentBlt` in msimg32.dll (not linked by default).

Block Transfer

```
BOOL GdiAlphaBlend(HDC dst, int xd, int yd, int wd, int hd
                   HDC src, int xs, int ys, int ws, int hs, BLENDFUNCTION fn)
```

- Almost the same as `GdiTransparentBlt` (see prev. slide)
- `src` and `dst` colors combined instead of overwriting
- Controlled by `fn`:
  - `BlendOp` must be `AC_SRC_OVER`, `BlendFlags` must be 0
  - `AlphaFormat` — set to `AC_SRC_ALPHA` if source has 32bpp and the *alpha* channel should be used for per-pixel opacity
  - `SourceConstantAlpha` — additional opacity used for entire source (set to 255 to ignore)

```
struct BLENDFUNCTION {
    BYTE BlendOp;
    BYTE BlendFlags;
    BYTE SourceConstantAlpha;
    BYTE AlphaFormat;
};
```

- Red channel of `dst` r_d updated with red and *alpha* channels (r_s, a_s) of the source color ($a_s = 255$ if no per-pixel opacity) and the constant opacity a_c:

$$r_d = r_s * \frac{a_s}{255.0} * \frac{a_c}{255.0} + r_d * \frac{a_c}{255.0}\left(1 - \frac{a_s}{255.0} * \frac{a_c}{255.0}\right)$$

- Green and blue channels (and *alpha* if it exists in `dst`) handled accordingly

The same function is also available as `AlphaBlend` in msimg32.dll (not linked by default)

Drawing Text

Text and font handling is the most complex part of GDI, we'll only cover some basics

- Text drawn using context's current text color:
  - `COLORREF GetTextColor(HDC hdc)` — check current
  - `COLORREF SetTextColor(HDC hdc, COLORREF color)` — change it (returns previous)
- Context's current text alignment flags used to align text against a reference point
  - `UINT GetTextAlign(HDC hdc)` — check current flags
  - `UINT SetTextAlign(HDC hdc, UINT align)` — change them (returns previous)
  - `TA_TOP`, `TA_BOTTOM`, `TA_BASELINE` — vertical alignment (default: `TA_TOP`), reference point will be on the top, bottom of text bounding box or on text's baseline
  - `TA_LEFT`, `TA_CENTER`, `TA_RIGHT` — horizontal alignment (default: `TA_LEFT`), reference point will be on the left, in the middle or on the right of text bounding box
  - `TA_UPDATECP` — if set, current position used as reference and is updated when drawing text
  - For flags related to RTL and vertical scripts, check the docs

Drawing Text

```
BOOL TextOutW(HDC hdc, int x, int y, LPCWSTR text, int c)
```

- (x,y) — reference point for alignment (ignored if `TA_UPDATECP` is set)
- `text` and it's length `c` — string to be drawn (doesn't need to be zero-terminated)

```
BOOL ExtTextOutW(HDC hdc, int x, int y, UINT opt, const RECT *rc,
                 LPCWSTR text UINT c, const INT *dx)
```

- (x,y) — reference point for alignment (ignored if `TA_UPDATECP` is set)
- `text` and it's length `c` — string to be drawn (doesn't need to be zero-terminated)
- `opt` flags control behaviour, incl. if and how `rc` is used:
  - `ETO_CLIPPED` — text is clipped to `rc`
  - `ETO_OPAQUE` — fill `rc` with context's current background color (see below)
    contrary to `TextOutW`, here text background is transparent by default regardless of current background color.
  - For other flags, check the docs
- `dx` — used for character spacing (check docs), pass `nullptr` for default spacing.

Drawing Text

```
int DrawTextW(HDC hdc, LPCWSTR text, int c, LPRECT rc, UINT format)
```

- Current text alignment must be `TA_LEFT` and `TA_TOP` without `TA_UPDATECP`
- `text` and it's length `c` (can pass -1 for `c` if `text` zero-terminated)
- `rc` — rectangle in which the text is laid out
- `format` flags control the output:
  - `DT_LEFT`, `DT_RIGHT`, `DT_CENTER` — align text horizontally to the left, right or in the center of `rc`
  - `DT_SINGLELINE` — outputs `text` in a single line, ignoring new-lines and carriage-returns
  - `DT_TOP`, `DT_BOTTOM`, `DT_VCENTER` — align text vertically to the top, bottom or in the center of `rc` (only for `DT_SINGLELINE`)
  - `DT_WORDBREAK` — automatically brakes lines before words that do not fit in `rc`
  - `DT_CALCRECT` – used to measure the text output without drawing (see docs)
  - Many other options, check the docs!

```
int DrawTextExW(HDC hdc, LPWSTR text, int c, LPRECT rc,
                UINT format, LPDRAWTEXTPARAMS params)
```

- Additional parameters controlling margins, tab-stops, etc.

Measuring Text

- `BOOL GetTextExtentPoint32W(HDC hdc, LPCWSTR text, int c, LPSIZE size)`
  - `text` and its length `c`
  - stores in `size` width and height of the text, as if drawn by `TextOutW`
- `DrawTextW` can measure text as well (see previous slide)

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)
```

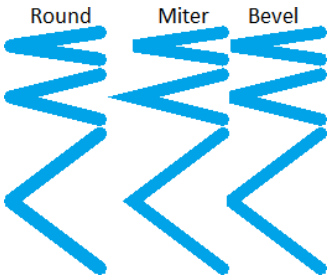
```
struct LOGPEN{
    UINT     lopnStyle;
    POINT    lopnWidth; //y unused
    COLORREF lopnColor;
};
```

- `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 $1px$
- `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 $3px$ 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)

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)`
  - By default creates a compatible bitmap w/ given resolution
  - For memory context `hdc` bound to device-independent bitmap (next slide), creates DIB with the same attributes.
  - *Bpp*, row alignment matches `hdc`'s surface
  - If `cx` or `cy` is 0, creates 1×1 monochrome bitmap (1*bpp*)
- `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)

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

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`

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 polyCounts, int mode);
```

  - `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
- On failure (e.g. `size` too 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

Path is a collection of lines, shapes and close figures (text included)

- Always tied to a device context, no separate handle type
- `BOOL BeginPath(HDC hdc)` creates a new path and binds it to the context, discarding any previous path
- Afterwards, any call to line, curve, closed shape, and text drawing function adds those shapes to the path instead of drawing them
- `BOOL CloseFigure(HDC hdc)`
  Closes the latest figure, usually straight line segment from current position to the most recent `MoveToEx` destination (compare to `PolyDraw`)
- `BOOL EndPath(HDC hdc)` closes the path, all drawing functions return to normal
- `BOOL AbortPath(HDC hdc)` discards any existing path (closed or not)

Path Modifications

- `BOOL FlattenPath(HDC hdc)`
  Converts curves in current path to series of line segments
- `BOOL WidenPath(HDC hdc)`
  - Converts the path to be the boundary of the area that would be painted over if the path was outlined with current pen.
  - Current pen must be a simple pen with width > 1 or a geometric pen.
  - Path is flattened in the process

Using Paths

- Filling and outlining: `FillPath`, `StrokePath`, `StrokeAndFillPath` (see above).
  Remember! Filling a path discards it.
- `HRGN PathToRegion(HDC hdc)`
  Converts the current path to a region and discards it.
- `int GetPath(HDC hdc, LPPOINT apt, LPBYTE aj, int cpt)`
  - Retrieves the path as a sequence of annotated points — same format that `PolyDraw` uses as input (see above)
  - If `cpt` is 0, returns the required size of `apt` and `aj` arrays
- Path can be used to limit the drawing area (`SelectClipPath`, see below)

Fonts

Lines and Curves State

Shapes not filled, only outlined. Outline drawn using context's current pen.

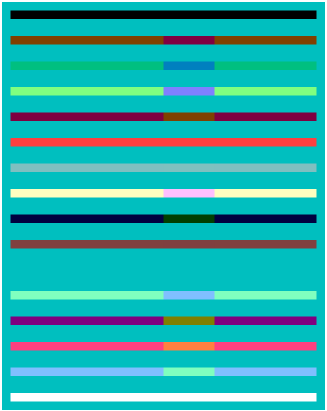
- `HGDIOBJ GetCurrentObject(HDC hdc, UINT type)`
  Pass `OBJ_PEN` as `type` to obtain `HPEN` of current pen (default: stock black pen)
- `HGDIOBJ SelectObject(HDC hdc, HGDIOBJ h)`
  - Pass `HPEN` handle as `h` to change current pen.
  - Previously selected pen returned as a result.
  - Old pen must be destroyed or (preferably) restored once you are done using the new one.
- For simple dashed pens, gaps between dashes filled based on context's background mode
  - `GetBkMode`, `SetBkMode` — check/select background mix mode
  - `TRANSPARENT` — background remains unchanged
  - `OPAQUE` — default, gaps filled with context's background color (not background brush!)
  - `GetBkColor`, `SetBkColor` — check/select background color (default: white)
- Other properties: pen type and style, `GetMiterLimit`, `SetMiterLimit`

Lines and Curves State

New color of a pixel mixed with the old one based on context's foreground mixing mode:

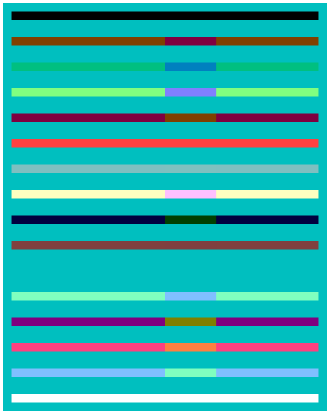
- GetROP2, SetROP2 — check/select foreground mixing mode (default: R2_COPYPEN)
- Combination of source (pen), destination (screen) colors using bitwise operations

```
//Background brush: RGB(0, 0XBF, 0XBF)
auto oldpn = SelectObject(hdc,
CreatePen(PS_DASH, 1, RGB(0X7F, 0, 0X7F)));
SetBkColor(hdc, RGB(0X7F, 0X7F, 0));
int rop2[16] = {    R2_BLACK,       R2_NOTMERGEPEN
    R2_MASKNOTPEN,  R2_NOTCOPYPEN, R2_MASKPENNOT,
    R2_NOT,         R2_XORPEN,      R2_NOTMASKPEN,
    R2_MASKPEN,     R2_NOTXORPEN,  R2_NOP,
    R2_MERGENOTPEN, R2_COPYPEN,    R2_MERGEPENNOT,
    R2_MERGEPEN,    R2_WHITE };
for(int i = 0; i < 16; ++i) {
    SetROP2(hdc, rop2[i]);
    MoveToEx(hdc, 2, i * 3 + 1, nullptr);
    LineTo(hdc, 38, i * 3 + 1);
} DeleteObject(SelectObject(hdc, oldpn));
```



Lines and Curves State

Background brush (D): #00bfbf
Pen (Dash) color (S): #7f007f
Background (Gap) color (S): #7f7f00



Mix Mode	Dash	Gap	BitOp
R2_BLACK	#000000	#000000	D ^ D
R2_NOTMERGEPEN	#804000	#800040	~S & ~D
R2_MASKNOTPEN	#00bf80	#0080bf	~S & D
R2_NOTCOPYPEN	#80ff80	#8080ff	~S
R2_MASKPENNOT	#7f0040	#7f4000	S & ~D
R2_NOT	#ff4040	#ff4040	~D
R2_XORPEN	#7fbfc0	#7fc0bf	S ^ D
R2_NOTMASKPEN	#ffffc0	#ffc0ff	~S | ~D
R2_MASKPEN	#00003f	#003f00	S & D
R2_NOTXORPEN	#80403f	#803f40	~S ^ D
R2_NOP	#00bfbf	#00bfbf	D
R2_MERGENOTPEN	#80ffbf	#80bfff	~S | D
R2_COPYPEN	#7f007f	#7f7f00	S
R2_MERGEPENNOT	#ff407f	#ff7f40	S | ~D
R2_MERGEPEN	#7fbfff	#7fffbf	S | D
R2_WHITE	#ffffff	#ffffff	D | ~D

Closed Figures State

- All functions outline and fill closed shapes
- Context's current position not used or modified
- Outline drawn with context's current pen (see prev. slides)
- Use `GetStockObject(NULL_PEN)` to omit the outline
- Interior filled with context's current brush
- `HGDIOBJ GetCurrentObject(HDC hdc, UINT type)`
  Pass `OBJ_BRUSH` to obtain `HBRUSH` of current pen (default: stock white brush)
- `HGDIOBJ SelectObject(HDC hdc, HGDIOBJ h)`
  - Pass `HBRUSH` handle as `h` to change current brush.
  - Previously selected brush returned as a result.
  - Old brush must be destroyed or (preferably) restored once you are done using the new one.
  - Use `GetStockObject(NULL_BRUSH)` to omit filling the shape

Closed Figures State

- Fill properties: brush type and style, `GetBrushOrgEx`, `SetBrushOrgEx`

```
HBITMAP bmp = (HBITMAP)LoadImageW(
    GetModuleHandleW(nullptr),
    MAKEINTRESOURCEW(IDB_BMPHELLO),
    IMAGE_BITMAP, 0, 0, LR_SHARED);
auto oldbr = SelectObject(hdc,
    CreatePatternBrush(bmp));
auto oldpn = SelectObject(hdc,
GetStockObject(NULL_PEN));
SetBrushOrgEx(hdc, 25, 25, nullptr);
Rectangle(hdc, 25, 25, 225, 125);
DeleteObject(SelectObject(hdc, oldpn));
DeleteObject(SelectObject(hdc, oldbr));
DeleteObject(bmp);
```

Result:

Result w/o `SetBrushOrgEx`:

- Foreground mix mode used for both outline and interior (`GetROP2`, `SetROP2`)
- Background mix mode and color used for gaps between lines in hatched brushes and simple dashed pens (`GetBkMode`, `SetBkMode`, `GetBkColor`, `SetBkColor`)

Text Drawing State

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

Coordinate Spaces

- All positions and sizes used for drawing are expressed in logical units in a more-or-less abstract World Space coordinate system.
- Before any actual pixels are modified a series of transformations must be performed:
  - World → Page Space
  - Page → Device (Context) Space
  - Device → Physical Device space
- Most of them are one-to-one by default

World to Page Space Transformations

- Default is an identity transformation
- Can only be changed in advance graphics mode
  - `int GetGraphicsMode(HDC hdc)` — check current
  - `int SetGraphicsMode(HDC hdc, int mode)` — change it (returns previous)
  - `mode` - one of: `GM_COMPATIBLE`, `GM_ADVANCED`

- Described as (affine) transformation matrix, that converts world space points (x_w, y_w) to page space points (x_p, y_p)

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} eM_{11} & eM_{21} & eD_x \\ eM_{12} & eM_{22} & eD_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix}$$

```
struct XFORM {
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
};
```

  - (`eM11`, `eM12`) —X-axis unit vector of World space in Page space
  - (`eM21`, `eM22`) — Y-axis unit vector of World space in Page space
  - (`eDx`, `eDy`) — origin of World space in Page space

World to Page Space Transformation

Basic transformations

- Translation (offset by d_x, d_y):

$$\begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_x \\ 0 & 0 & 1 \end{bmatrix}$$

- Scaling by factors s_x, s_y (enlarge w/ > 1, shrink w/ $\in (0, 1)$, reflect w/ < 0)

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotation around the origin by angle α

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Shear by factors s_x, s_y (use 0 to avoid shearing in a direction)

$$\begin{bmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

World to Page Space Transformation

- `BOOL CombineTransform(LPXFORM xfOut, const XFORM *xf1, const XFORM xf2)`
  - Combines two transformation
  - Result stored in `xfOut`
  - `xfOut` same as transforming with `xf1` first, followed by `xf2`
- `BOOL GetWorldTransform(HDC hdc, LPXFORM xf)` — obtains context's current transform
- `BOOL SetWorldTransform(HDC hdc, const XFORM *xf)` — replaces it
- `BOOL ModifyWorldTransform(HDC hdc, const XFORM *xf, DWORD mode)`
  - Modifes context's current world transform
  - `mode` controls the behaviour
  - `MWT_IDENTITY` — resets current transform to an identity (`xf` ignored)
  - `MWT_LEFTMULTIPLY` — combines current with `xf` (`xf` first, followed by current)
  - `MWT_RIGHTMULTIPLY` — combines current with `xf` (current first, followed `xf`)

Page to Device Space Transformation

- Specifies units used in Page space and their size in device context's pixels
- Can only perform translations and scaling (including flipping axes direction)
- Unit scaling, axes direction controlled by context's mapping mode
- `int GetMapMode(HDC hdc)` — check current mode
- `int SetMapMode(HDC hdc, int mode)` — change it (returns previous)
- Available `mode`s:
  - `MM_TEXT` — 1 page unit = 1 pixel, X axis →, Y axis ↓(default)
  - `MM_LOMETRIC` — 1 page unit = 0.1mm, X axis →, Y axis ↑
  - `MM_HIMETRIC` — 1 page unit = 0.01mm, X axis →, Y axis ↑
  - `MM_LOENGLISH` — 1 page unit = 0.01in, X axis →, Y axis ↑
  - `MM_HIENGLISH` — 1 page unit = 0.001in, X axis →, Y axis ↑
  - `MM_TWIPS` — 1 page unit = 1pt ($1/1440$in), X axis →, Y axis ↑
  - `MM_ISOTROPIC`, `MM_ANISOTROPIC` — custom mappings (see next slide)

Page to Device Space Transformation

- Translation and custom mappings controlled by viewport
- Two complementary sets of functions (usually you only use one set)
- `-Window-` functions specify mapping from window area to page space
  `GetWindowExtEx`, `GetWindowOrgEx`, `OffsetWindowOrgEx`, `ScaleWindowExtEx`, `SetWindowExtEx`, `SetWindowOrgEx`
- `-Viewport-` functions specify mapping from page space to the window area
  `GetViewportExtEx`, `GetViewportOrgEx`, `OffsetViewportOrgEx`, `ScaleViewportExtEx`, `SetViewportExtEx`, `SetViewportOrgEx`

Device to Physical Device Transformation

- Purely automatic, no way to change it
- Offsets positions so they appear in correct positions depending on the physical device

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∩meta) regions

Device Context Attributes

End of Windows API Lecture 4

Thank you for listening! ☺

Raster Operations

Various GDI functions perform bitwise operations on inputs to determine output color

- Inputs include:
  - `D` — destination color, i.e. color initially stored in a output pixel of the destination bitmap,
  - `S` — source color obtained from the source bitmap that corresponds to the output pixel,
  - `P` — pattern color, i.e. color obtained from the pen, brush or background color corresponding to the output pixel when filling or outlining,
  - `M` — mask bit, obtained from a monochrome mask bitmap (same 0 or 1 bit is used for all bits of the output pixel).
- Functions grouped by inputs used:
  - Binary raster operations — use `D` and `P` — line, curve and closed shape functions (any affected by `SetROP2`)
  - Ternary raster operations — use `S`, `D` and `P` — block transfer functions `BitBlt` and `StretchBlt`
  - Quaternary raster operations — use `S`, `D`, `P` and `M` — masked block transfer function `MaskBlt`
- Boolean function used to determine an output bit based on corresponding input bits described by raster operation code

Binary Raster Operations

- Functions $\{0,1\} \times \{0,1\} \to \{0,1\}$
- 4 possible combinations for input bits D, P
- Function uniquely described by the outputs for those 4 combinations
- Total of $2^4 = 16$ unique functions possible
- GDI code for each function: sequence of output values as a binary number + 1
- Many equivalent boolean expressions exist for each function
- GDI code symbolic constant names based on canonical boolean representation in Reverse Polish Notation

	P	1	1	0	0	
	D	1	0	1	0	RPN
R2_BLACK	1	0	0	0	0	0
R2_NOTMERGEPEN	2	0	0	0	1	DP|~
R2_MASKNOTPEN	3	0	0	1	0	DP~&
R2_NOTCOPYPEN	4	0	0	1	1	P~
R2_MASKPENNOT	5	0	1	0	0	PD~&
R2_NOT	6	0	1	0	1	D~
R2_XORPEN	7	0	1	1	0	DP^
R2_NOTMASKPEN	8	0	1	1	1	DP&~
R2_MASKPEN	9	1	0	0	0	DP&
R2_NOTXORPEN	10	1	0	0	1	DP^~
R2_NOP	11	1	0	1	0	D
R2_MERGENOTPEN	12	1	0	1	1	DP~|
R2_COPYPEN	13	1	1	0	0	P
R2_MERGEPENNOT	14	1	1	0	1	PD~|
R2_MERGEPEN	15	1	1	1	0	DP|
R2_WHITE	16	1	1	1	1	1

Ternary Raster Operations

- Functions $\{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1\}$
- 8 possible combinations of input bits S, P and D
- Function uniquely described by the outputs for those 8 combinations
- Total of $2^8 = 256$ unique functions possible
- Raster operation — 4 byte value:

3	2	1	0
Zero	Function Index	Operation Code	

- Raster Operation Code encodes procedure to calculate the function:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op_5		Op_4		Op_3		Op_2		Op_1		~	Parse String			Offset	

Ternary Raster Operations

Raster operation bytes (3 – highest; 0 – lowest):

- Byte 3 equals 0 — allows for combinations with additional flags: `CAPTUREBLT`, `NOMIRRORBITMAP`
- Byte 2 — function index, sequence of output values for each input combination as binary number
- Byte 1 and 0 — raster operation code, encodes equivalent boolean expression in RPN:
  - Bits 15-6 — 5 boolean operator indices, 2 bits per,
  - Bit 5 — flag indicating if ~ (NOT) used as additional (sixth) operator,
  - Bits 4-2 — input parameter string (aka. *parse string*) index
    + and - indicate pushing to and pulling values from a temporary stack. Most expression: sequence of inputs followed by sequence of operators. In few cases an operator needed in between operands — in such instance an intermediate value needs to be temporarily stored on a stack (see examples)
  - Bit 1-0 — offset into parameter string

Index	Bool Op.
0	~ (NOT)
1	^ (XOR)
2	| (OR)
3	& (AND)

Index	Parse String
0	SPDDDDDD
1	SPDSPDSP
2	SDPSDPSD
3	DDDDDDDD
4	DDDDDDDD
5	S+SP-DSS
6	S+SP-PDS
7	S+SD-PDS

Ternary Raster Operations Examples

PATPAINT 0X00FB0A09

- Function index: 0XFB

P	1	1	1	1	0	0	0	0
S	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0
Output	1	1	1	1	1	0	1	1
	F				B			

- Raster Operation Code: 0X0A09

0				A				0				9			
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	1
~		~		|		|		~		0	2: SDPSDPSD		1	
Op_5		Op_4		Op_3		Op_2		Op_1		~		Parse String		Offset	

  - 2 binary operators, so 3 input parameters from parse string 2 starting at offset 1: DPS
  - RPN expression: DPS~||~~ ≡ DPS~||
  - Infix expression: ((~S)|P)|D

Ternary Raster Operations Examples

`0X002916CA`

- Function index: `0X29`

P	1	1	1	1	0	0	0	0
S	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0
Output	0	0	1	0	1	0	0	1
	2				9			

- Raster Operation Code: `0X16CA`

1				6				C				A			
0	0	0	1	0	1	1	0	1	1	0	0	1	0	1	0
~		^		^		|		&		0	2: SDPSDPSD		2	
Op_5		Op_4		Op_3		Op_2			Op_1		~	Parse String		Offset	

  - 4 binary operators, so 5 input parameters from parse string 2 starting at offset 2: `PSDPS`
  - RPN expression: `PSDPS&|^^~`
  - Infix expression: `~((((S&P)|D)^S)^P)`

Ternary Raster Operations Examples

0X00420D5D

- Function index: 0X42

P	1	1	1	1	0	0	0	0
S	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0
Output	0	1	0	0	0	0	1	0
		4				2		

- Raster Operation Code: 0X0D5D

0				D				5				D			
0	0	0	0	1	1	0	1	0	1	0	1	1	1	0	1
~		~		&		^		^		0	7: S+SD-PDS		1		
Op_5		Op_4		Op_3		Op_2		Op_1		~	Parse String		Offset		

  - 3 binary operators, so 4 input parameters from parse string 7 starting at offset 1 (+ and - don't count): +SD-PD
  - expression: +SD-PD^^&
  - Infix expression: ~((((S&P)|D)^S)^P)