Courses IPCx, C1: Bitmap, FAQs

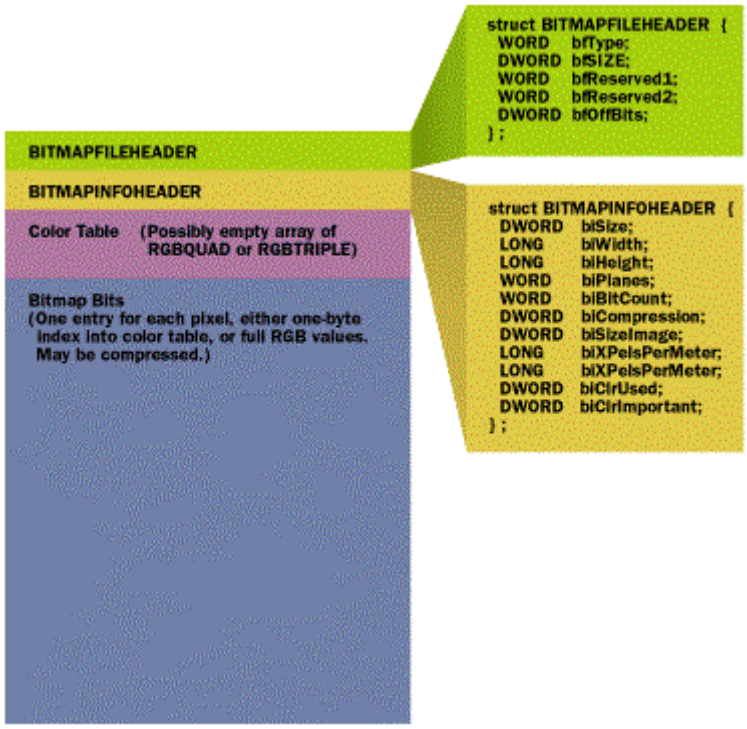
Was ist ein Bitmap ?

"Bitmap" im weiten Sinn ist ein altmodischer aber populärer Ausdruck für ein Rasterbild.
"Bitmap" im engeren Sinn (IBM 1988) ist eine Bezeichnung für eine Familie einfacher Rasterbild-Datenformate aus der Spätzeit von DOS und der Frühzeit von OS/2 und Windows.
Die Qualifizierung der Bitmap-Datenformate als "einfach" ist allerdings relativ.
Eine gute Vertiefung dieses Stoffes finden Sie bei Charles Petzold: Windows-Programmierung. 5. Auflage, Seiten 609 - 775, Microsoft Press, ISBN 3-86063-487-9, DM 119.--
deutsch:
http://mspress.microsoft.de/press/product.asp?cl=&sku=3%2D86063%2D487%2D9&dept%5Fid=1000&mscssid=38XSSP2PW7S92LB800LHRNXB27XX1E2C
english: http://mspress.microsoft.com/prod/books/2344.htm

Problem 1: Die Bezeichnung "Bitmap" ist zweifach irreführend:
1. Rasterpixel hatten früher oft den Datentyp Bit, das war sinnvoll in Zeiten von CGA (IBMColorGraphicsAdapter) und HGC (HerculesGraphicsCard). Solche Graphikkarten sind aber ausgestorben. Pixel haben heute den Datentyp BYTE oder RGBTRIPLE. Das Bit als Datenstruktur für ein Pixel ist selten geworden (noch vorhanden bei Bildmasken).
2. Unter Maps versteht man heute Baumstrukturen. Raster haben aber keine Baumstruktur sondern die Form eindimensionaler (z.B. Bild[xSize*ySize]) oder zweidimensionaler Arrays (z.B. Bild[xSize][ySize]).
3. Zusammenfassung: Bitmap hat weder mit Bit noch mit Map etwas zu tun, sondern bezeichnet einen BYTE-Array.

Problem 2: Microsoft verwendet den Begriff Bitmap für zwei verschiedene Formatgruppen:
1. Es gibt ein Bitmap-Fileformat *.BMP (genauer: eine Familie von Bitmap-Fileformaten), mit dem man Rasterbilder an Empfänger senden kann, deren Hardware man nicht kennt. Microsoft bezeichnet dieses Format zutreffend als **Device Independent Bitmap Format DIB**. Dieses Format hat sich aus dem Windows-Betriebssystem heraus zum populärsten Standard für den Austausch von Rasterbildern entwickelt. Es ist seit langem ein W3W-Internet-Standard. Wenn man vom Bitmap-Format spricht oder verkürzt von Bitmaps, meint man fast immer dieses DIB-Fileformat. Auch in diesem Kurs bedeutet das Wort "Bitmap" ein File im DIB-Format (erkenntlich an der Filename-Extension *.BMP).
2. Innerhalb des Betriebssystems Windows gibt es eine Datenstruktur BITMAP als Teil der mächtigen Datenstruktur Device Context DC. Diese BITMAP-Datenstruktur verpackt Rasterdaten in eine Form, die man schnell und effektiv innerhalb des Hauptspeichers und zwischen Hauptspeicher = Main Memory und Graphikkarte über den Bus kopieren kann. Das Format heißt zutreffend **Device Dependent Bitmap Format DDB.** Dieses Format ist nur relevant innerhalb einer Maschine (genauer eines Windows-Betriebssystems, das seine Graphikkarte genau kennt). Wenn man ein *.BMP-File als Bild sehen will, muss man das DIB in den Hauptpeicher einlesen, in ein DDB umwandeln und in die Graphikkarte kopieren. Das ist der Zusammenhang zwischen den beiden ganz verschiedenen Datenstukturen DIBitmap und DDBitmap.

Aufbau des Device Independent Bitmap Formats (DIB)



Aus historischen Gründen hat das DIBitmap-Format zwei Header, einen ersten BITMAPFILEHEADER und einen zweiten BITMAPINFOHEADER.
Bei 8-Bit-Bitmaps folgt nach dem BITMAPINFOHEADER in der Regel die Palette = Color Table.
Die Palette = Color Table ist ein Array variabler Länge (zwischen 1 und 256 Elemente vom Typ RGBQUAD mit 4 Byte).
BITMAPINFOHEADER und Palette sind zusammengefasst in eine gemeinsame Datenstruktur namens BITMAPINFO (nicht im Bild).
Ein BITMAPINFO enthält also immer einen BITMAPINFOHEADER und eine Palette (gelb und violett), wobei letztere auch die Länge Null haben kann.
Nach BITMAPINFO kommen die sogenannten Bitmap Bits (die eigentlich Bytes sind) anfangend mit dem linkesten Pixel der untersten Zeile, danach die nächsthöhere Zeile bis zur obersten Zeile.

In den Headern gibt es wichtige, weniger wichtige und unwichtige Einträge.
Jeder Hersteller einer *.BMP entscheidet selbst, was er in die Header schreibt.

Wenn ein Filename mit der Extension *.BMP endet, können Sie davon ausgehen, dass der Inhalt des Files ein DIB ist.
Sicher ist das nicht.
Folgende minimale Bedingungen müssen erfüllt sein, sonst ist ein *.BMP- File definitv unbrauchbar:
1. Das erste Byte des Files ist der ASCII-Code des Buchstabens B.
2. Das zweite Byte des Files ist der ASCII-Code des Buchstabens M.
3. Die Filelänge ist größer als 54 Byte. // =gemeinsame Länge beider Header
4. bfOffBits ist größer gleich 54. // =Position des 1. Pixels im File
5. biBitCount ist gleich 1 oder 4 oder 8 oder 24 // =Anzahl Bit pro Pixel

Einteilung der DIBs nach biBitCount

Bit pro Pixel	Byte pro Pixel	Farben	Paletten-einträge	Palettenlänge in Bytes	Bezeichnung
1	1/8	keine	0	0	Binärbild
4	1/2	16	16	64	EGA-Bild
8	1	keine	0	0	S/W-Bild
8	1	256	256	1024	VGA-Bild
16	2	32.768	0	0	HighColor-Bild
24	3	16.777.216	0	0	Echtfarbbild
32	4	16.777.216	0	0	Echtfarbe+Transparenz

In den folgenden Absätzen werden die DIBs in der Reihenfolge ihrer Wichtigkeit behandlet.
Weil sie selten sind, werden die DIBs mit 1, 4, 16 und 32 Bit pro Pixel überhaupt nicht behandelt.

Aufbau der 24-Bit-DIBs

Von allen Bitmaps kommen diese DIBs am meisten vor. Sie heißen auch True Color DIBs oder Echtfarben-Bitmaps.
Vorteil 1: Diese Bilder verfügen über den gesamten Farbraum des RGB-Farbmodells (=RGB-Gammut).
Vorteil 2: Diese Bilder brauchen keine Palette und keine Look-Up-Table.
Nachteil 1: Diese Bilder enthalten die 3-fache Redundanz von 8-Bit-DIBs.
Nachteil 2: Look-Up-Table-Animationen sind nicht möglich.

BITMAPFILEHEADER	BitMAPINFOHEADER	24-Bit Pixel Data
14 Byte	>=40 Byte	many

Field	Bytes	BITMAPFILEHEADER for 24-Bit-Bitmaps
bfType	2	Bitmap identifier. Must be 'BM'.
bfSize	4	length of file
bfReserved	2	0
bfReserved	2	0
bfOffbits	4	Specifies the location (in bytes) in the file of the image data. For our 24-bit bitmaps, this will be sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER).
Field	**Bytes**	**BITMAPINFOHEADER for 24-Bit-Bitmaps**
biSize	4	size of INFOHEADER structure. Thus, it is >=40.
biWidth	4	The width of the bitmap, in pixels
biHeight	4	The height of the bitmap, in pixels
biPlanes	2	1
biBitCount	2	24
biCompression	4	If bitmaps are uncompressed, this field is set to zero.
biSizeImage	4	The size of the padded image, in bytes
biXPelsPerMeter	4	Horizontal resolution, in pixels per meter, of device displaying bitmap
biYPelsPerMeter	4	Vertical resolution, in pixels per meter, of device displaying bitmap
biClrUsed	4	This number does not apply to 24-bit bitmaps. Set to zero.
biClrImportant	4	This number does not apply to 24-bit bitmaps. Set to zero.
Field	**Bytes**	**24 - Bit Pixel Data**
pixel[0]	3	Here begins an 1D array of blue, green, red values.
.....	.	.....

Aufbau der 8-Bit-DIBs

8-Bit-DIBs verwendet man in folgenden Fällen anstelle von 24-Bit-DIBs:
1. Es handelt sich um Bilder ohne Farbe. (z.B. Röntgen, Ultraschall, S/W-Photos)
2. Es handelt sich um Bilder mit wenigen künstlichen Farben. (z.B. aus Zeichenprogrammen wie Paint, Illustrator etc.)
3. Man kann sich den Speicherplatz und die Ladezeiten von 24-Bit-DIBs nicht leisten. (z.B. im Internet)
4. Man braucht die Animationseffekte, die mit einer Palette möglich sind. (z.B. bei Games)

BITMAPFILEHEADER	BITMAPINFOHEADER	Palette	8-Bit Pixel Data
14 Byte	>=40 Byte	0, 4, 8, ... max 1024 Byte	many

Field	Bytes	BITMAPFILEHEADER for 8-Bit-Bitmaps
bfType	2	Bitmap identifier. Must be 'BM'.
bfSize	4	length of file
bfReserved	2	0
bfReserved	2	0
bfOffbits	4	Specifies the location (in bytes) in the file of the image data. For our 8-bit bitmaps, this will be sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * numPaletteEntries.
Field	**Bytes**	**BITMAPINFOHEADER for 8-Bit-Bitmaps**
biSize	4	size of the INFOHEADER structure. Thus, it is >=40.
biWidth	4	The width of the bitmap, in pixels
biHeight	4	The height of the bitmap, in pixels
biPlanes	2	1
biBitCount	2	8
biCompression	4	If bitmaps are uncompressed, this field is set to 0.
biSizeImage	4	The size of the padded image, in bytes
biXPelsPerMeter	4	Horizontal resolution, in pixels per meter, of device displaying bitmap
biYPelsPerMeter	4	Vertical resolution, in pixels per meter, of device displaying bitmap
biClrUsed	4	This indicates how many colors are in the palette.
biClrImportant	4	This indicates how many colors are needed to display the bitmap. Set it to 0, indicating all colors should be used.
Field	**Bytes**	**Palette = Color Table = bmiColors[i]**
		256 repetitions of the following 4 lines: i=0.....255
rgbRed[0]	1	Red value
rgbGreen[0]	1	Green value
rgbBlue[0]	1	Blue value
rgbReserved[0]	1	mostly zero, sometimes used for a Transparency value
.....	.	.....
Field	**Bytes**	**8 - Bit Pixel Data**
pixel[0]	1	Here begins an 1D array of unsigned characters, where each value is an index into the palette.
.....	.	.....

Probleme der DIBs

Es wäre naiv anzunehmen, dass Bitmaps einfache Datenstrukturen seien.

Verständnisproblem mit der Doppelbezeichnung BITMAPINFO und BITMAPINFOHEADER
BITMAPINFO ist eine Datenstruktur variabler Länge. Sie kann nie kürzer als 40 BYTE sein und nie länger als 40 + 4*256 = 1064 Byte.
BITMAPINFO ist nämlich der gemeinsame Oberbegriff für BITMAPINFOHEADER und Palette:
In BITMAPINFO ist immer ein BITMAPINFOHEADER der Länge 40 Byte enthalten, manchmal eine Palette, aber oft auch nicht.
Die Information, ob eine Palette da ist oder nicht und wenn ja, mit wieviel Farben steht im BITMAPINFOHEADER unter biClrUsed.
Die mächtigen Funktionen, mit denen man DIBs vom Hauptspeicher in die Graphikkarte laden kann (z.B. StretchDIBits()) verlangen unter vielen anderen Parametern einen Pointer auf ein BITMAPINFO, nicht auf einen BITMAPINFOHEADER. Obwohl beide Pointer immer auf die gleiche Adresse zeigen, funktionieren diese Funktionen nur, wenn man ihnen einen Pointer vom Typ (BITMAPINFO *) übergibt. Die Funktionen belohnen einem damit, dass sie die Palette dann auch automatisch benutzen, sofern eine da ist.
Probleme mit der y-Achse:
1. Rechteckige Rasterbilder adressiert der Mensch gerne zweidimensional mit einer Spaltenzahl x und einer Zeilenzahl y.
2. Ausgehend von der europäischen Schreibrichtung (technisiert in den Sync-Signalen der Zeilendisplays) sind wir gewöhnt, dass der Nullpunkt eines Rasters links oben liegt, die y-Achse also nach unten zeigt und das Raster zeilenweise von links nach rechts geschrieben wird.
3. Das widerspricht aber der Gewohnheit der analytischen Geometrie, wo die y-Achse nach oben zeigt. Die Erfinder des Bitmap-Formats haben das Bitmap-Format geometrisch definiert: Der Nullpunkt liegt links unten im Bild und die nullte Zeile beginnt links unten im Bild und die y-Achse zeigt nach oben.
4. Man kan folglich die Pixel eines Bitmapfiles nie in der Reihenfolge auf den Display schreiben, wie sie auf der Hardisk stehen, sondern man muß erst alle Zeilen lesen und dann die letzte (oberste) Zeile zuerst ausgeben.
Probleme der linearen Adressierung = Serialisierung:
1. Rechenmaschinen und Harddisks haben einen linearen Adressraum, d.h. die Zeilen eins Bildes müssen hintereinander aufgereiht werden.
2. Wenn man das erste Pixel einer Zeile direkt hinter das letzte Pixel der Vorgängerzeile schreibt, bekommt das Rasterbild im Hauptspeicher und auf der Harddisk die gute Form eines kontinuierlich laufenden Datenstroms ohne jede Zeilenstruktur adressierbar durch einen einzigen linearen Index i.
3. Wenn der Programmierer die Zeilenlänge xSize kennt, ist das kein großes Problem. Die Zeilenummer y errechnet sich aus i ganzzahlig dividiert durch xSize (Formel: y = i / xSize) und die Spaltennummer x ist der Rest der Division (Formel: x = i % xSize).
4. Umgekehrt, wenn der Programmierer die Spaltennummer x und die Zeilenummer y eines Pixels kennt, kann er leicht die lineare Adresse ausrechenen: i = y mal xSize plus x (Formel: i = y *xSize + x).
Probleme mit der Zeilenlänge:
1. Rechner konnten schon früher keine Bits adressieren, sondern nur Bytes. Moderne 32-Bit Rechner können weder Bits noch Bytes adressieren, sondern nur komplette Worte der Länge 32 Bit = 4 Bytes. Die Speicher sind also wie kariertes Papier eingeteilt in kleine minmal adressierbare Bereiche .
2. Um interne Microprogramme zu beschleunigen, haben die Erfinder des Bitmap-Formats festgelegt, dass jede Zeile genau auf so einer Adressgrenze beginnen muss. Das bedeutet, dass zwischen dem letzten Pixel einer Zeile und dem ersten Pixel der nächsten Zeile Leerpixel eingefüllt werden müssen und zwar immer dann, wenn die Zeilengrenze nicht exakt auf der 32-Bit Speichereinteilung liegt. Diese Festlegung hat schon viele Programmierer in Wahnsinn und Selbstmord getrieben.
Probleme mit der Farbe:
Es ist unmöglich ein Format zu erfinden, das auf jedes Display dieser Welt passt. Beispiel: Windows funktioniert einwandfrei auf binären Graphikkarten, die nur hell und dunkel kennen. Es ist klar, dass Windows (nicht WindowsCE) auf solchen Graphikkarten so gut wie alle DIBs katastrophal anzeigt.
Alte Probleme zwischen Microsoft und IBM:
Es gibt ein spezielles Color Table Paletten-Format für das IBM-Betriebssystem OS/2, es besitzt nur 3 Byte pro Farbe (RGBTRIPLE statt RGBQUAD). Man muss mühsam aus bfOffBits und biClrUsed berechnen, ob ein BMP seine Palette im Windows-Microsoft-Format oder (das ist zum Glück selten der Fall) im IBM-OS/2--Format kodiert hat. (Wir ignorieren die IBM-Bitmaps, was wahrscheinlich zum Crash führt, wenn Sie so ein File öffnen.)

Was ist ein Device Dependent Bitmap DDB ?

DDPs besitzen weder einen BITMAPFILEHEADER noch einen BITMAPINFOHEADER noch eine Palette.
Sie haben eine variable Datenstruktur, die sich nach dem aktuellen lokalen Device Context richtet. Wenn so ein DDB vom Hauptspeicher in die Graphikkarte kopiert wird, sind keinerlei Anpassungen von Format und Farbe notwendig. Ein DDB zeichnet sich dadurch aus, dass alle Formatangaben so gespeichert sind, dass sie 1:1 in den Device Context und von dort über den Bus in die Graphikkarte kopiert werden können.
Vorteil: DDBs sind viel schneller als DIBs.
Nachteil: DDBs kann man nicht zwischen Rechnern austauschen, es sei denn, man beschränkt sich auf einen untersten Standard (800x600, 16 Farben), den so gut wie alle Graphikkarten haben.
In Wahrheit wandelt Windows alle DIBs zuerst in DDBs um, bevor sie in die Graphikkarte kopiert werden. Dieser Umwandlungsvorgang ist so zeitaufwendig, dass man damit keine Games schreiben könnte.
Ausweg:
Bei der Installation einer Windows-Games werden alle Bilder in Form von DIBs von der CD gelesen und einmal und für eine feste Graphikkarteneinstellung in DDBs umgewandelt und als DDBs auf die Festplatte geschrieben.
Vorteil: Zur Laufzeit wird auf die schnellen DDBs zugegriffen und nicht auf die langsamen DIBs.
Nachteil 1: Doppelter Speicherplatzbedarf
Nachteil 2: Komplette Neuinstallation notwendig, wenn eine neue Graphikkarte eingebaut wurde, wenn der Graphiktreiber erneuert wurde oder wenn auch nur die Auflösung des Windows-Desktops geändert wurde (letztere wird von den Games meistens automatisch auf 800x600 reduziert).
Diesen Weg gehen alle Bilder, die oft benötigt werden (z.B. Bildschrimschoner). Man speichert sie in der Regel doppelt, sowohl als tranportable DIBs wie auch als an die lokale Maschine (exakter: an den letzten Device Context) gebundene DDBs.

CBitmap ist die komfortable MFC-Klasse für DDBs. Mit `CreateCompatibleBitmap` erzeugt das Betriebssystem im Hauptspeicher ein getreues Abbild des Videomemory der Graphikkarte. Der Programmierer lenkt alle Graphikbefehles in dieses Speicherabbild und wenn er damit fertig ist kopiert er das Speicherabbild komplett 1:1 über den Bus ins Videomemory mit einem sogenannten Bit-Block-Transfer `BitBlt`. Wenn man mit `CBitmap` ein DDB auf einer Maschine anlegt, speichert und an einer anderen Maschine abspielt, dann kann das gut gehen, in den meisten Fällen entsteht aber Farb- und Pixelmüll.

Sind DIBs immer langsam ?

MFC stellt für DIBs leider keine Klasse wie CBitmap für DDBs zu Verfügung, was ein schmerzlicher Mangel ist. Jedoch bietet das Windows-API (nicht aber MFC) einige sehr gute Funktionen für DIBs an: `CreateDIBitmap, GetDIBits, SetDIBits, StretchDIBits`. Microsoft hat es geschafft, diese früher grausam langsamen Funktionen wirksam zu beschleunigen. Trotzdem bleiben sie deutlich langsamer als ihre DDB-Entsprechungen.
Schnelle DIBs sind nur möglich mit Graphikkarten, die OpenGL- und/oder DirectX- Treiber anbieten.
Diese Treiber umgehen die gesamte Client-Server-Architektur von Windows, sie greifen direkt auf die Hardware zu. Sie sind zwar ein Sicherheitsrisko (unter NT gehen sie nicht, unter Windows2000 rätselhafterweise aber schon), aber sie machen DIBs schnell und spieletauglich und sind damit heute die unverzichtbare Basis für Multimedia unter Windows.

Extrabytes am Zeilenende beim Abspeichern von DIBs

Falls Sie DIBs auf die Harddisk speichern wollen, werden Sie auf das Problem stoßen, dass alle Zeilen am Anfang eines 32-Bit-Speicherwortes beginnen müssen, d.h. man darf nicht fugenlos eine Zeile nach der anderen schreiben, sonst wird der, der das File öffnen wird, keine Freude daran haben.
Greg Slabaugh macht folgenden intelligenten Programmiervorschlag:

The .bmp file format requires that the data along a scanline (horizontal line) in the image be aligned on a 4-byte boundary.

For 8-bit images, this means that the width of the image must be a multiple of 4. To address this issue, we will need to pad the image data before writing out to disk. The amount of padding is given by **extrabytes = (4 - width % 4) % 4**, where **width** is the width of the image. Let's see what this line of code does. Computing width % 4 determines by how many pixels we are over a 4-byte boundary. Subtracting this from 4 reveals how many pixels we need to pad the image so that its width is a multiple of 4. We finally take another % 4 to handle the case when width is already a multiple of 4. In this case, the padding should be zero bytes.

A 24 - bit bitmap writes out a (b, g, r) for each pixel in the image. Thus, we store 3 bytes for each pixel. Note that the order for the colors must be (b, g, r), not (r, g, b). As for 8-bit bitmaps, the data along a scanline (horizontal line) in the image must be aligned on a 4-byte boundary. For 24-bit images, this means that the quantity (3 * width) must be a multiple of 4. To address this issue, we will need to pad the image data before writing out to disk. The amount of padding is given by **extrabytes = (4 - (3 * width) % 4) % 4**, where **width** is the width of the image. Let's see what this line of code does. Computing (3 * width) % 4 determines by how many bytes we are over a 4-byte boundary. Subtracting this from 4 reveals how many bytes we need to pad the image so that its width is a multiple of 4. We finally take another % 4 to handle the case when (3 * width) is already a multiple of 4. In this case, the padding should be zero bytes.

Do not forget that we need to write out the .bmp data *upside-down*. So we will need to flip the image upside-down before writing out to disk.

Wie zoomt StretchDIBits ?

Das Zoomen von Matrizen ist grundsätzlich unmöglich (Ausnahme ganzzahlige Vergrößerung), weil eine Spalte/Zeile nicht verbreitert, sondern allenfalls verdoppelt werden kann. `StretchDIBits` scheint das unmögliche möglich zu machen.
Erklärung des Widerspruchs: `StretchDIBits` verzerrt das Bild weil die Funktion willkürliche Spalten /Zeilen verdoppelt (beim Vergrößern) oder löscht (beim Verkleinern). Die Positionen der Verdoppelungen und Löschungen werden möglichst äquidistant über alle Zeilen/Spalten so verteilt, dass die unvermeidlichen Verzerrungen möglichst wenig auffallen. Die enorme Redundanz der Bilder und die erstaunliche Störunempfindlichkeit der menschlichen visuellen Wahrnehmung macht den Betrachter genügend resistent, vorausgesetzt, die Verzerrungen treten nicht gehäuft an einer Stelle auf, sondern statistisch gleichverteilt überall im Bildfeld.
Man darf aber niemals Bilder, die Meßwerte liefern sollen (z.B. Koronarangiographien) mit `StretchDIBits` zoomen, es entstehen unkontrollierbare Messfehler durch das willkürliche Einfügen/Weglassen von Spalten/Zeilen.