Blizzard's GRP Image Format

from Lambchops, (a boy from AUS)

You can often find me online at server.war2.ru ☺ the world's last active Warcraft 2 Server.

(has been for the last 10 years and still going strong – drop in any time for a free nostalgia hit)

This document describes the GRP graphics format

which was developed by ***Blizzard Entertainment*** and used to store and render 8bit graphics elements in various game titles first released in the mid to late 90's. ***Specifically it relates to the GRP format as it was used in Warcraft 2.***

***** Although I'm not aware of it being the case, it's quite possible that some time in the last 20 years Blizzard could have expanded the specification to include support for any number of later formats and/or instructions, if so then this information is not intended to describe any such new-fangled shenanigans.

What's here?

☺ A *description of the GRP format and some thoughts on why it's implemented that way and how it is used by the Warcraft 2 game engine*

Overview:

GRPs hold one or more (generally related) 8 bit images. Typically they hold multiple images of the same subject matter that are used for animations.

The format allows for transparency without declaring a particular palette entry to be the "transparent color". GRPs are designed for use within an 8 bit palette base graphics environment, however they contain no palettes and describe only pixel data. The visible pixels are pre-packed in a bytestream that is optimised for fast rendering over the top of a background image using non-graphics specific hardware (*i.e.* Just a 386 or 486 CPU) as it was developed before the widespread adoption of modern GPUs.

Contents:

The GRP Header:

The first structure in a GRP is very simple. It is 6 bytes in length and contains just the number of images in the set and the canvas size, like this:

```
WORD  NumberOfFrames
WORD  CanvasWidth
WORD  CanvasHeight
```

Number Of Frames:

This is the number of **Frame Header**s that follow the **GRP Header.**
This is not necessarily the number of images that are present in the GRP.

Canvas Size:

Each GRP set has what I call a "*canvas*" size, which is an arbitrary bounding box that all of the images in the set must fit within. The exact size is not terribly critical as it is a imaginary construct, the only real use of which is to define the centre of the X and Y axis for the purposes of centring the image on a specific location, or for mirroring the image if required. Although these things are not part of the GRP format itself, it is designed to facilitate them being achieved at runtime with a minimum of effort.

As the maximum lateral shift available from a single GRP 'instruction' is 127 pixels, the format is designed for and best suited to image sets <= 128 pixels in width, however there is nothing I am aware of to preclude stringing 2 or more 'SHIFT' instructions together to allow for larger images, although I have not seen this done in any of the *Blizzard* GRPs that I have examined. It should also be noted that the maximum non-transparent width and height for any individual frame is hard limited to 255 pixels by the storage of **`LineWidth`** and **`NumberOfLines`** as type BYTE in the **Frame Header**s.

The largest GRP canvas sizes in Warcraft 2 are for town halls etc., which are 4x4 game squares in size. With each square being 32x32 pixels, so these units fill their canvas size which is fixed at 128x128. All the other building canvas sizes are set at 64x64 for 2x2 buildings (farms, towers etc.), 96x96 for 3x3 (most others), the only exception being oil platforms which are 96x96 for what is essentially a 2x2 structure as they also hold the image of the underlying oil patch. In contrast many of the 'unit' canvas sizes are quite generous, for example the peon image sets use a 72x72 canvas despite hardly ever getting near the edge and certainly never touching it (***see figure 1*** *).*

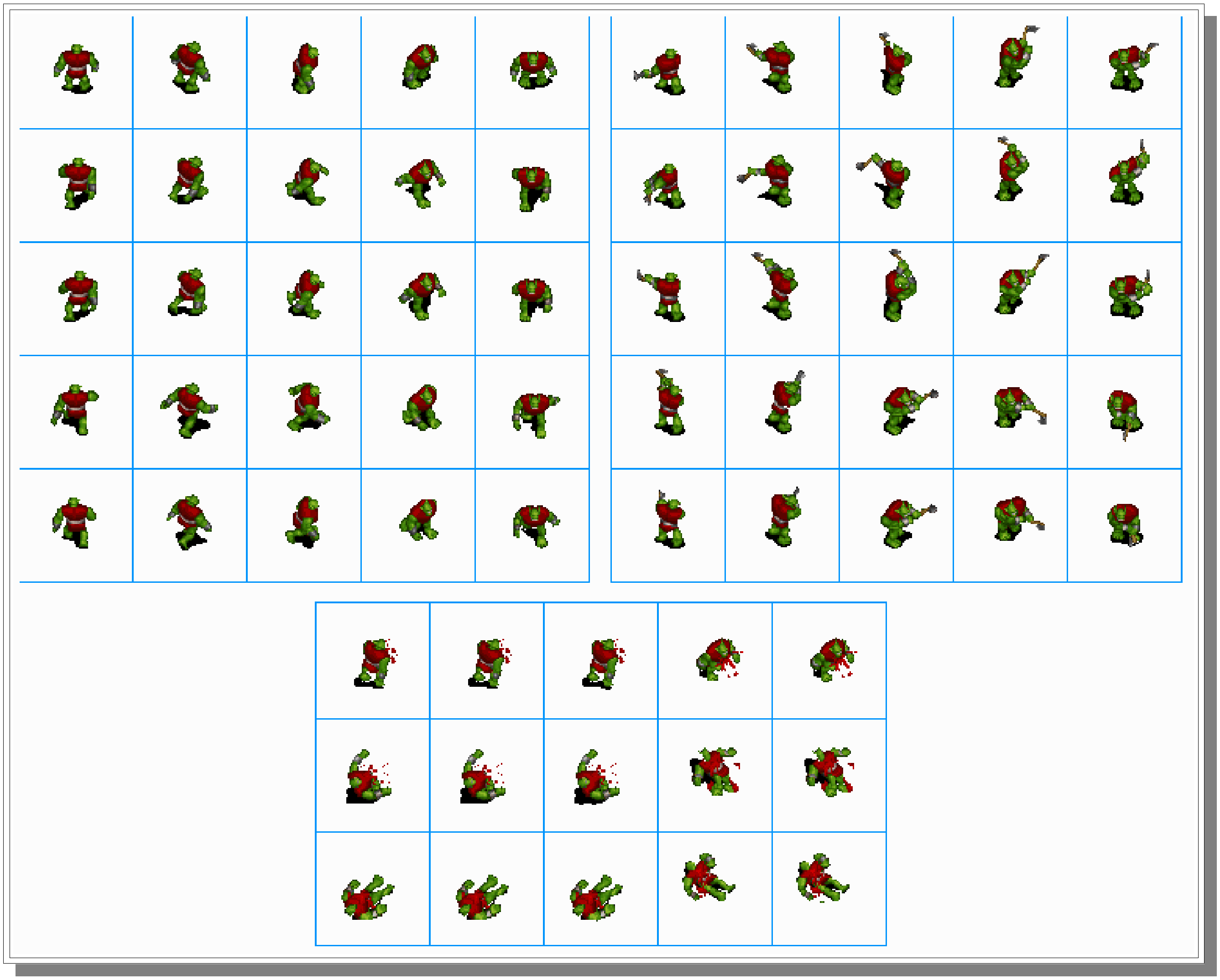


Figure 1 – The Standard Peon GRP

The 65 "standard peon" frames all sit quite comfortably inside the peon's 72x72 pixel canvas (the blue grid). Being well less than 128 pixels in width, its size makes no real difference as only the visible pixel data is encoded in the GRP. The 'SHIFT' instruction takes up one byte regardless, for any SHIFT < 128 and uses no more or less CPU time to calculate the offset for any value up to that limit. Each frame is referenced sequentially in the GRP file, they are arranged here so the different types of images can be easily identified. First are the 5 'stand' frames, then 20 'move' frames, followed by 25 'attack' frames and finally the 15 'dead' frames.

The Frame Headers:

Immediately following the ***GRP Header*** is one ***Frame Header*** for each frame in the set. Frame Headers are 8 Bytes long, so the frame header section of the GRP is 8xNumberOfFrames bytes in size. Each frame header is as follows:

```
BYTE     XOffset
BYTE     YOffset
BYTE     LineWidth
BYTE     NumberOfLines
DWORD    OffsetToData
```

The values are used as follows:
Starting from the top of the (imaginary) canvas, **`YOffset`** pixel lines are skipped on the destination bitmap before unpacking the first pixel line. Each line of pixel data then has an implied lateral shift of **`Xoffset`** pixels from the edge of the canvas before any further shifting of the pixel output location is applied by the bytestream data. There are exactly **`NumberOfLines`** lines of pixel data in the frame, and each pixel line is exactly **`LineWidth`** pixels long, ***including any 'SHIFT' instructions, but not the*** **`XOffset`** ***value*.** The **`OffsetToData`** member contains the offset from the base of the GRP to the start of the **Data Header** structure *(see below)*.

☼ *Keeping track of the number of pixels described in each line of pixel data is the only reliable way to decode a GRP frame.*

Whether it's an *SHIFT*, a *REPEAT* or a *PIXEL* instruction just add it all up, and when it equals **`LineWidth`** then that line is done.

The **Frame Header**s appear immediately after the **GRP Header**, with their order defining the number of each frame. Each **Frame Header** contains a **`DWORD`** offset to the pixel data that could be anywhere at all in the GRP object and does not necessarily have to appear in any specific order. This feature allows a single chunk of bytestream data to be defined as the target for any number of declared frames.

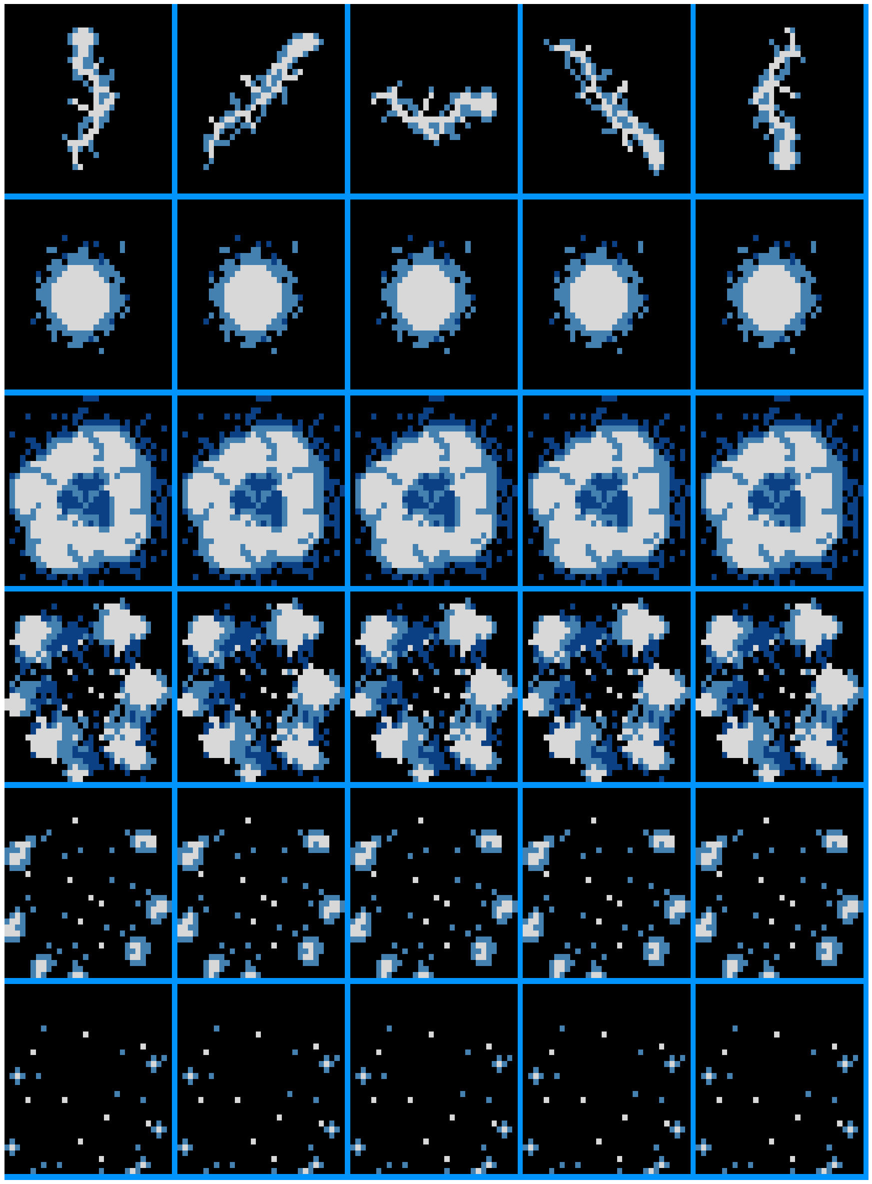


Figure 2 – Mage Lightning GRP

The GRP for mage's 'lightning' missile weapon contains 30 frames. The first 5 frames are unique, however the remaining 25 frames consist of 5 groups of 5 identical frames each.

In fact, this GRP contains 30 frame headers, but only 10 frames of pixel data. The 'repeat' frames are just repeats of the same 8 byte **Frame Header** which, of course has the same value for `OffsetToData` so points to the same block of pixel data for each frame.

Figure 3 shows the actual header values for the mage lightning GRP.

GRP Header 0000: NumberOfFrames : 30 0002: CanvasWidth : 32 0004: CanvasHeight : 32		
Frame Header: Frames 1 - 10	**Frame Header: Frames 11 - 20**	**Frame Header: Frames 21 - 30**
0006: F01 XOffset : 11	0056: F11 XOffset : 1	00A6: F21 XOffset : 0
0007: F01 YOffset : 11	0057: F11 YOffset : 1	00A7: F21 YOffset : 0
0008: F01 LineWidth 11	0058: F11 LineWidth 31	00A8: F21 LineWidth 32
0009: F01 NumberOfLines 24	0059: F11 NumberOfLines 32	00A9: F21 NumberOfLines 27
000A: F01 OffsetToData: 0x000000F6	005A: F11 OffsetToData: 0x00000684	00AA: F21 OffsetToData: 0x00000BCD
000E: F02 XOffset : 5	005E: F12 XOffset : 1	00AE: F22 XOffset : 0
000F: F02 YOffset : 5	005F: F12 YOffset : 1	00AF: F22 YOffset : 0
0010: F02 LineWidth 23	0060: F12 LineWidth 31	00B0: F22 LineWidth 32
0011: F02 NumberOfLines 23	0061: F12 NumberOfLines 32	00B1: F22 NumberOfLines 27
0012: F02 OffsetToData: 0x000001E9	0062: F12 OffsetToData: 0x00000684	00B2: F22 OffsetToData: 0x00000BCD
0016: F03 XOffset : 4	0066: F13 XOffset : 1	00B6: F23 XOffset : 0
0017: F03 YOffset : 4	0067: F13 YOffset : 1	00B7: F23 YOffset : 0
0018: F03 LineWidth 24	0068: F13 LineWidth 31	00B8: F23 LineWidth 32
0019: F03 NumberOfLines 11	0069: F13 NumberOfLines 32	00B9: F23 NumberOfLines 27
001A: F03 OffsetToData: 0x000002EB	006A: F13 OffsetToData: 0x00000684	00BA: F23 OffsetToData: 0x00000BCD
001E: F04 XOffset : 4	006E: F14 XOffset : 1	00BE: F24 XOffset : 0
001F: F04 YOffset : 4	006F: F14 YOffset : 1	00BF: F24 YOffset : 0
0020: F04 LineWidth 23	0070: F14 LineWidth 31	00C0: F24 LineWidth 32
0021: F04 NumberOfLines 23	0071: F14 NumberOfLines 32	00C1: F24 NumberOfLines 27
0022: F04 OffsetToData: 0x00000399	0072: F14 OffsetToData: 0x00000684	00C2: F24 OffsetToData: 0x00000BCD
0026: F05 XOffset : 10	0076: F15 XOffset : 1	00C6: F25 XOffset : 0
0027: F05 YOffset : 10	0077: F15 YOffset : 1	00C7: F25 YOffset : 0
0028: F05 LineWidth 11	0078: F15 LineWidth 31	00C8: F25 LineWidth 32
0029: F05 NumberOfLines 24	0079: F15 NumberOfLines 32	00C9: F25 NumberOfLines 27
002A: F05 OffsetToData: 0x0000049C	007A: F15 OffsetToData: 0x00000684	00CA: F25 OffsetToData: 0x00000BCD
002E: F06 XOffset : 5	007E: F16 XOffset : 0	00CE: F26 XOffset : 0
002F: F06 YOffset : 5	007F: F16 YOffset : 0	00CF: F26 YOffset : 0
0030: F06 LineWidth 19	0080: F16 LineWidth 32	00D0: F26 LineWidth 31
0031: F06 NumberOfLines 20	0081: F16 NumberOfLines 31	00D1: F26 NumberOfLines 25
0032: F06 OffsetToData: 0x0000058F	0082: F16 OffsetToData: 0x0000091E	00D2: F26 OffsetToData: 0x00000D1D
0036: F07 XOffset : 5	0086: F17 XOffset : 0	00D6: F27 XOffset : 0
0037: F07 YOffset : 5	0087: F17 YOffset : 0	00D7: F27 YOffset : 0
0038: F07 LineWidth 19	0088: F17 LineWidth 32	00D8: F27 LineWidth 31
0039: F07 NumberOfLines 20	0089: F17 NumberOfLines 31	00D9: F27 NumberOfLines 25
003A: F07 OffsetToData: 0x0000058F	008A: F17 OffsetToData: 0x0000091E	00DA: F27 OffsetToData: 0x00000D1D
003E: F08 XOffset : 5	008E: F18 XOffset : 0	00DE: F28 XOffset : 0
003F: F08 YOffset : 5	008F: F18 YOffset : 0	00DF: F28 YOffset : 0
0040: F08 LineWidth 19	0090: F18 LineWidth 32	00E0: F28 LineWidth 31
0041: F08 NumberOfLines 20	0091: F18 NumberOfLines 31	00E1: F28 NumberOfLines 25
0042: F08 OffsetToData: 0x0000058F	0092: F18 OffsetToData: 0x0000091E	00E2: F28 OffsetToData: 0x00000D1D
0046: F09 XOffset : 5	0096: F19 XOffset : 0	00E6: F29 XOffset : 0
0047: F09 YOffset : 5	0097: F19 YOffset : 0	00E7: F29 YOffset : 0
0048: F09 LineWidth 19	0098: F19 LineWidth 32	00E8: F29 LineWidth 31
0049: F09 NumberOfLines 20	0099: F19 NumberOfLines 31	00E9: F29 NumberOfLines 25
004A: F09 OffsetToData: 0x0000058F	009A: F19 OffsetToData: 0x0000091E	00EA: F29 OffsetToData: 0x00000D1D
004E: F10 XOffset : 5	009E: F20 XOffset : 0	00EE: F30 XOffset : 0
004F: F10 YOffset : 5	009F: F20 YOffset : 0	00EF: F30 YOffset : 0
0050: F10 LineWidth 19	00A0: F20 LineWidth 32	00F0: F30 LineWidth 31
0051: F10 NumberOfLines 20	00A1: F20 NumberOfLines 31	00F1: F30 NumberOfLines 25
0052: F10 OffsetToData: 0x0000058F	00A2: F20 OffsetToData: 0x0000091E	00F2: F30 OffsetToData: 0x00000D1D

Figure 3 – Mage lightning GRP and Frame Headers.

Here we can see that after the first 5 Frame Headers, there are 5 lots of 5 repeats of the same frame header listed.

If you note that the **`OffsetToData`** for the first line of the first frame is **`0x000000F6`** and the <u>location</u> of the **`OffsetToData`** member for the last frame (30) is **`0x00F2,`** then as it is a **`DWORD`** (4 byte) value; 0xF2 + 4 = 0xF6 demonstrates that the bytestream for frame 1 appears directly after the last Frame Header... of course, as we know it doesn't necessarily *have* to appear here, but in reality as this is the very first byte of pixel data written to the GRP when it is encoded, there is no real point in putting it anywhere else. Regardless, this need not be relied upon, it is presented here only as proof of theory and a mnemonic device.

Using Frame headers to do the thinking.

GRP Frame Headers help the WC2 game engine to reduce the amount of work the graphics engine has to do when rendering each screen update.

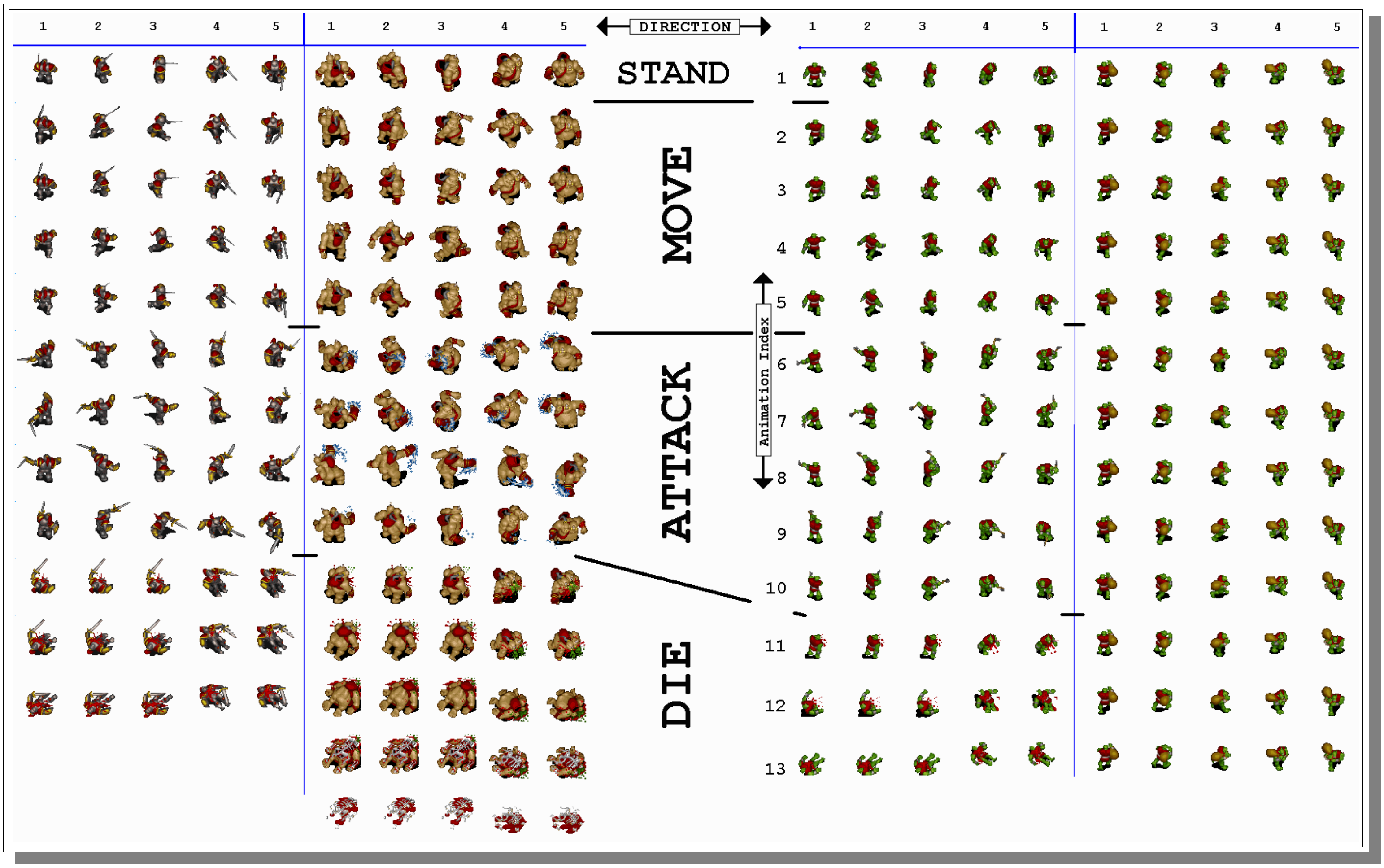


Figure 4 – Unit GRP Comparison

The footman and ogre attack animations have 4 frames each, while the peon attack/chop animation has 5 frames. Both the peon and footman have 3 frames in their death animation, while the ogre has 5. You can note that there are only 2 sets of death animation images for each unit to cover the 5 direction categories (meh... it's only a corpse anyway).

Also of interest is the gold carry GRP set for the peon which really only covers the 'stand' and 'move' actions, but has duplicate frame headers that repeat the images to also cover the attack and death animations. Presumably this is a safeguard in case a screen update is triggered after the peon's current action has been updated to 'attack' or 'dead' but before its GRP set has been switched back to the "standard peon" set.

The Game Engine vs. The Display Engine.**

The game engine has plenty to do. It spends a lot of its time fussing around its units – keeping all their various stats, timers, bits and bobs up to date, but there are 4 items of information that it needs to have ready on a sliver platter for the display engine:

1) The GRP set the unit is using.
2) The current animation index of the unit.
3) The direction the unit is facing.
4) and, of course, the screen co-ords where the unit is to be displayed

It needs to cross check the animation frame update time with the game timer to keep the animation index correct and up to date, however this value is entirely independent of the direction the unit is facing. The game engine keeps track of the unit's facing direction, however it doesn't actually make use of this value.

☺ *The facing direction is, in fact, a nonsense value which is kept only for visual aesthetics and plays no part in the game mechanics for any unit. It is generated according to the direction a unit is moving or attacking, or randomly updated for a unit that is standing, but it's only purpose is to be supplied to the display engine so that it can make pretty pictures that appeal to our funny, squishy, grey CPUs (never really thought about that did you?* **;)**

The game engine does most of the number crunching for the display engine, because the state of each unit is updated far less frequently than the screen is rendered. Also, while it doesn't do this type of 'fancy' (lol) calculations, when the ka-ka hits the whirly-gig, it is the display engine that has the real mother-load of grunt work to do, unpacking tens of thousands of GRP pixels into the screen buffer as may times a second as it can manage. So the game engine keeps the display data updated, and when the display engine has to suddenly render 150 units on the screen without getting bogged down, it has all the information it needs ready to go.

So each unit only has to be updated a couple of times a second if it is active, less if it is standing, but the screen needs to be updated according to the display frame-rate, Which is... um... well... I quite honestly have no idea what sort of frame rate WC2 needs to run at to look normal now I think about it, lol, but I'd take a stab at somewhere around a minimum of 12fps or so. Anything below that is going to look and feel pretty coarse.... regardless, when the display code decides it needs to render a given unit, it only needs its 4 pieces of information.

Actually the game engine updating the animation index is only an assumption, as in most cases the only a small subset of the total units in the game is actually being displayed, so it is possible that that the animation index is only updated for units being displayed. However when you consider that, for instance, a grunt standing next to an enemy farm doesn't stop attacking when its not on screen, it seems unlikely.

***note to self:** *must inject some code some time to check out the frame-rate...*

Deciding which frame to render where

...or "4 pieces of information and how to use them"

In most cases, the GRP set ***(1)*** is static for the life of the unit and solely dependent on the unit type. ***i.e.*** Ogres use the ogre GRP, as do ogre-mages and ogre-mage heroes. This never changes from when the unit is first trained to when its corpse lies rotting on the battlefield.

The obvious exceptions to this rule are peasants/peons which switch to a different GRP set when they are carrying gold or wood then switch back to the standard one when they either attack, die or return their goods. Similarly oil tankers have 2 GRP sets for when they are/aren't carrying oil, but as tankers can't attack the situation is a lot simpler, have only 5 directional images and death (sinking) animations in both sets. ***Also most buildings share common GRPs for when they are first placed and immediately after they are destroyed (building corpses).***

☼ *The animation index* ***(2)*** *and the facing direction* ***(3)*** *are combined to calculate which frame in the GRP is displayed and how it is displayed, but as the facing direction never interacts with the animation index (or anything else) the animation sequences always remain un-compromised provided the unit's action remains constant. For instance, the 'move' animation remains fluid even as a unit changes directions multiple times to negotiate obstacles.*

(nice ☺).

So basically, if the display engine was using the arbitrary (and no-doubt inaccurate) values I have printed on ***Figure 4,*** then ***assuming direction values are used that start with '1' for 'North' or facing straight up then proceed clockwise at 45° increments, so 'NE' = 2, 'East'= 3, 'SE'= 4 etc., finishing with NW = 8,*** it would simply calculate something similar to this:

☼ First find the final direction index from the unit's facing direction.

```
IF (facing_direction <= 5) THEN
   direction_index = facing_direction
   Mirror = False
ELSE
   direction_index = 10 - facing_direction
   Mirror = True
END
```

☼ Once the direction index and mirroring state are found, it just displays:

frame number `( animation_index – 1)*5 +direction_index`

..... at the appropriate screen co-ordinates***(4)***, either *mirrored* or *not,* as was dictated when we found the direction index.

Just one comparison, possibly a simple subtraction, a multiplication by 5 and a simple addition is all the work the display engine has to do before it does a couple of simple skips down the GRP structure and starts rendering pixels.

Using this system, all sorts of decisions about what frame to display in what parts of which animations, and which frames to copy for another purpose etc. can actually be made by the graphic artists while they drink kale smoothies and compare hipster beards in another building, and neither the programmer nor the nor the display engine even need think about it, let alone write *or execute* code to handle it. The Frame Headers are, in effect, operating mostly as a pointer table and are saving the game engine from having to make all sorts decisions about animation sequences every iteration.

Player Colors

There is, of course one more factor in Warcraft 2 unit display: the units are different colors for each player. This is achieved by using a defined range of palette entries - shades of red on the source GRP - for the relevant colored parts of each image. These entries then have a “color shift” value added to them when they are encountered by the display engine which effectively substitutes shades of the appropriate player's color. This 5^{th} piece of information has been omitted here for the sake of simplicity.

End of the Line: The Pixel Data

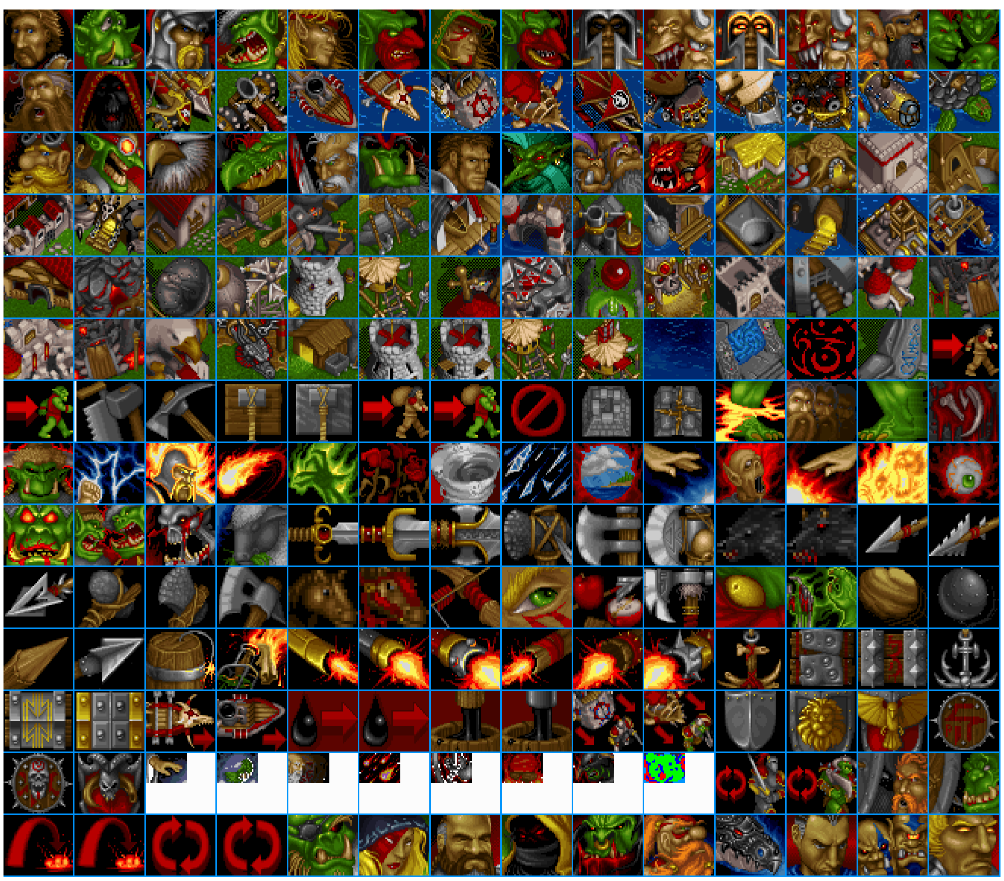
So, each frame header has a DWORD field called OffsetToData, it's an offset from the start of the GRP Header to ... well ... ***"HERE"*** =D ....the pixel data. Keep track of the rest of the frame header members too, we'll need them here.

The Data Header:

```
WORD   LineDataOffset x NumberOfLines
```

The **Data Header** simply consists of one **`WORD`** offset value for each pixel line in the frame. **`LineDataOffset`** is the offset from the start of the **Data Header** to the start of the data for each pixel line. Adding this value to **`OffsetToData`** from the relevant **Frame Header** will yield an offset from the GRP base. Typically the data for the first pixel line appears directly after the **Data Header** so the first **`LineDataOffset`** is usually equal to **(`NumberOfLines` x 2)**

☼ There are explanations of the offsets and how to detect the end of a pixel line in the **Frame Header** section.



The unit portraits and interface button images are stored in a single GRP that contains 198 Frames measuring 46 x 38 pixels

Figure 5 – Portraits and Buttons

Decoding the Pixel Bytestream

- The bytestream is list of instructions.
- Each instruction consists of a ***Code Byte*** followed by 0 or more ***Data Bytes***.

So when you have found the start of the data for a pixel line that you wish to render, read the first BYTE, this will be a ***Code Byte*** which will determine the nature of its instruction as follows:

If the ***Code Byte*** > **0x80** then it's a **SHIFT** instruction

The SHIFT instruction has no ***Data Bytes***
Subtract the **0x80** from the ***Code Byte*** to find the shift distance.
Add this value to the destination write pointer.

Elsif the ***Code Byte*** > **0x40** then it's a **REPEAT** instruction

The **REPEAT** instruction has **1** ***Data Byte***
Subtract the **0x40** from the ***Code Byte*** to find the repeat count.
Write the data byte at the destination write pointer target that many times
(incrementing the destination write pointer each time)

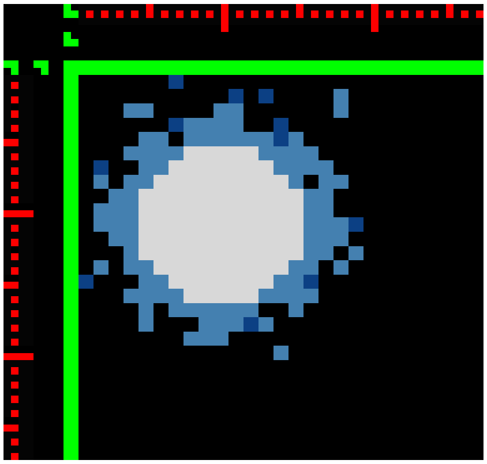
Else it's a **PIXEL** instruction

The **PIXEL** instruction has a variable number of ***Data Bytes***
The ***Code Byte*** value is the pixel count
There are that many ***Data Bytes*** of raw pixel data following
(write them out)

Then read the next **Code Byte** and repeat, until the total of all the:

shift distance + **repeat count** + **pixel count**

...values for the current line equals **`LineWidth`** from the **Frame Header**



Pixel Header FRAME 6

```
058F: line 01 ofs : 0x0028
0591: line 02 ofs : 0x002C
0593: line 03 ofs : 0x0036
0595: line 04 ofs : 0x0042
0597: line 05 ofs : 0x004B
0599: line 06 ofs : 0x0056
059B: line 07 ofs : 0x005E
059D: line 08 ofs : 0x006A
059F: line 09 ofs : 0x007A
05A1: line 10 ofs : 0x0084
05A3: line 11 ofs : 0x008F
05A5: line 12 ofs : 0x009B
05A7: line 13 ofs : 0x00A6
05A9: line 14 ofs : 0x00B1
05AB: line 15 ofs : 0x00C1
05AD: line 16 ofs : 0x00CE
05AF: line 17 ofs : 0x00D6
05B1: line 18 ofs : 0x00E0
05B3: line 19 ofs : 0x00EB
05B5: line 20 ofs : 0x00F1
```

Bytestream

```
line01 CB   #  Instr  Data
05B7: [86]   6 SHIFT
05B8: [01]   1 PIXEL  [B7]
05BA: [8C]  12 SHIFT

line02 CB   #  Instr  Data
05BB: [8A]  10 SHIFT
05BC: [01]   1 PIXEL  [B7]
05BE: [81]   1 SHIFT
05BF: [01]   1 PIXEL  [B7]
05C1: [84]   4 SHIFT
05C2: [01]   1 PIXEL  [BB]
05C4: [81]   1 SHIFT

line03 CB   #  Instr  Data
05C5: [83]   3 SHIFT
05C6: [02]   2 PIXEL  [BB:BB]
05C9: [84]   4 SHIFT
05CA: [02]   2 PIXEL  [BB:BB]
05CD: [86]   6 SHIFT
05CE: [01]   1 PIXEL  [BB]
05D0: [81]   1 SHIFT

line04 CB   #  Instr  Data
05D1: [86]   6 SHIFT
05D2: [01]   1 PIXEL  [B7]
05D4: [44]   4 REPEAT [BB]
05D6: [82]   2 SHIFT
05D7: [01]   1 PIXEL  [B7]
05D9: [85]   5 SHIFT

line05 CB   #  Instr  Data
05DA: [84]   4 SHIFT
05DB: [02]   2 PIXEL  [BB:BB]
05DE: [81]   1 SHIFT
05DF: [46]   6 REPEAT [BB]
05E1: [02]   2 PIXEL  [B7:BB]
05E4: [84]   4 SHIFT

line06 CB   #  Instr  Data
05E5: [83]   3 SHIFT
05E6: [44]   4 REPEAT [BB]
05E8: [45]   5 REPEAT [71]
05EA: [44]   4 REPEAT [BB]
05EC: [83]   3 SHIFT

line07 CB   #  Instr  Data
05ED: [81]   1 SHIFT
05EE: [01]   1 PIXEL  [B7]
05F0: [82]   2 SHIFT
05F1: [02]   2 PIXEL  [BB:BB]
05F4: [47]   7 REPEAT [71]
05F6: [44]   4 REPEAT [BB]
05F8: [82]   2 SHIFT

line08 CB   #  Instr  Data
05F9: [81]   1 SHIFT
05FA: [01]   1 PIXEL  [BB]
05FC: [81]   1 SHIFT
05FD: [02]   2 PIXEL  [BB:BB]
0600: [49]   9 REPEAT [71]
0602: [01]   1 PIXEL  [BB]
0604: [81]   1 SHIFT
0605: [02]   2 PIXEL  [BB:BB]
0608: [81]   1 SHIFT

line09 CB   #  Instr  Data
0609: [82]   2 SHIFT
060A: [02]   2 PIXEL  [BB:BB]
060D: [4B]  11 REPEAT [71]
060F: [02]   2 PIXEL  [BB:BB]
0612: [82]   2 SHIFT

line10 CB   #  Instr  Data
0613: [81]   1 SHIFT
0614: [03]   3 PIXEL  [BB:BB:BB]
0618: [4B]  11 REPEAT [71]
061A: [02]   2 PIXEL  [BB:BB]
061D: [82]   2 SHIFT

line11 CB   #  Instr  Data
061E: [81]   1 SHIFT
061F: [03]   3 PIXEL  [BB:BB:BB]
0623: [4B]  11 REPEAT [71]
0625: [04]   4 PIXEL  [BB:BB:BB:B7]

line12 CB   #  Instr  Data
062A: [82]   2 SHIFT
062B: [02]   2 PIXEL  [BB:BB]
062E: [4B]  11 REPEAT [71]
0630: [03]   3 PIXEL  [BB:BB:BB]
0634: [81]   1 SHIFT

line13 CB   #  Instr  Data
0635: [83]   3 SHIFT
0636: [01]   1 PIXEL  [BB]
0638: [4B]  11 REPEAT [71]
063A: [02]   2 PIXEL  [BB:BB]
063D: [81]   1 SHIFT
063E: [01]   1 PIXEL  [BB]

line14 CB   #  Instr  Data
0640: [81]   1 SHIFT
0641: [01]   1 PIXEL  [BB]
0643: [81]   1 SHIFT
0644: [02]   2 PIXEL  [BB:BB]
0647: [49]   9 REPEAT [71]
0649: [02]   2 PIXEL  [BB:BB]
064C: [81]   1 SHIFT
064D: [01]   1 PIXEL  [BB]
064F: [81]   1 SHIFT

line15 CB   #  Instr  Data
0650: [01]   1 PIXEL  [B7]
0652: [83]   3 SHIFT
0653: [02]   2 PIXEL  [BB:BB]
0656: [47]   7 REPEAT [71]
0658: [03]   3 PIXEL  [BB:BB:B7]
065C: [83]   3 SHIFT

line16 CB   #  Instr  Data
065D: [83]   3 SHIFT
065E: [44]   4 REPEAT [BB]
0660: [45]   5 REPEAT [71]
0662: [44]   4 REPEAT [BB]
0664: [83]   3 SHIFT

line17 CB   #  Instr  Data
0665: [84]   4 SHIFT
0666: [01]   1 PIXEL  [BB]
0668: [81]   1 SHIFT
0669: [46]   6 REPEAT [BB]
066B: [82]   2 SHIFT
066C: [01]   1 PIXEL  [BB]
066E: [84]   4 SHIFT

line18 CB   #  Instr  Data
066F: [84]   4 SHIFT
0670: [01]   1 PIXEL  [BB]
0672: [83]   3 SHIFT
0673: [05]   5 PIXEL  [BB:BB:BB:B7:BB]
0679: [86]   6 SHIFT

line19 CB   #  Instr  Data
067A: [87]   7 SHIFT
067B: [03]   3 PIXEL  [BB:BB:BB]
067F: [89]   9 SHIFT

line20 CB   #  Instr  Data
0680: [8D]  13 SHIFT
0681: [01]   1 PIXEL  [BB]
0683: [85]   5 SHIFT
```

Figure 6 - Pixel Data for mage lightning – frame 6

GRP Format - Quick Reference

GRP Header

```
WORD   NumberOfFrames
WORD   CanvasWidth
WORD   CanvasHeight
```

Frame Header `[NumberOfFrames]`

```
BYTE   XOffset
BYTE   YOffset
BYTE   LineWidth
BYTE   NumberOfLines
DWORD  OffsetToData
```

(*offset from* **GRP Header**)

Data Header

```
WORD   LineDataOffset [NumberOfLines]
```

(*offset from* **Data Header**)

Code Bytes

0x01-0x3F	PIXEL	(+n Data Bytes)
0x41-0x7F	REPEAT	(+1 Data Byte)
0x81-0xFF	SHIFT	(No Data Bytes)

In Conclusion

My presumption of the intent of GRP's creation is that it was first and foremost about optimization... It was about getting the maximum possible graphics performance out of the hardware of the day, and was successful in that purpose.

With the benefit of the original GRP design, knowledge of its subsequent use and 20 years worth of hindsight, I might have implemented it *slightly* differently, however performance-wise there probably wouldn't be a noticeable improvement, possibly a tiny measurable one, but one thing that is to me, undeniable (because I remember it vividly) is that when Warcraft 2 first appeared in 1996 it ***looked*** better than any computer game I had ever seen.

GRP is more space efficient than traditional image containers, so it also had a positive effect by reducing RAM and file storage requirements. As the whole point was to get the pixels on the screen using the fewest possible number of CPU clocks, using any traditional compressed image format would have been totally counter-productive as it could even have resulted in more CPU time being used decompressing the images that displaying them. The GRP designers got around this by building in custom RLE compression that became part of the optimised bytestream in such a way that it was decompressed as it was being rendered by the CPU and actually made the process less work instead of more – which is a pretty neat trick, albeit simple and obvious in hindsight, as the best ideas tend to be.

The only part of the spec that perhaps should have been modified is the 4 *BYTE-sized* members present in the Frame Header structure. These would probably have been better incorporated into the Data Header, leaving the Frame Headers as just an array of DWORD pointers. To my mind this arrangement would probably have even been slightly faster, certainly no slower. It would have also resulted in smaller GRP sizes in some cases (and never larger). It just makes more sense as those 4 members describe the data, and repeating them for every 'dummy' frame header is just redundant.

Perhaps the X and Y offsets could be varied in duplicate frame headers as a method of scrolling animation, although I have not seen this being used. Certainly the 2 bytes that follow describe the data itself. I also might possibly have made them WORD sized instead of BYTE which would have given GRP a lot more flexibility, but in practice I don't think the 255x255 maximum frame size ever hindered *Blizzard's* efforts when making 8 bit games.

...however all this is just nit-picking after the fact, because 20 or so years ago, GRP was designed and implemented and did the exact job it was designed for, and did it well.

**** N.B.** My characterisation herein of two separate entities that I refer to as the “Game Engine” and the “Display Engine” are just my own projection of how I envisage 2 subsections of the WC2 executable could operate based on observations about the way the GRP graphics format is implemented and utilised. They are presented as explanatory tools to highlight the features of GRP graphics format and the way that those feature could be exploited to maximum effect by the executable and should not be mis-interpreted as any sort of literal description of how this executable actually functions or is constructed, nor be considered dissemination of any such proprietary information.

(Dear Lawyers, I made those bits up, if you don't believe me, go ask the programmers ;)

GRP Decoding Routine ASM Source

The following procedure will decode an arbitrary frame from GRP graphics resource, optionally mirrored about the Y axis (horizontally), to a destination pixmap of arbitrary width. It is assumed that the GRP object as a whole has already been placed in memory at the supplied address (i.e. Read from any disk file, should it be contained in such) and that the destination pixmap is in a 1 byte per pixel format compatible with 8bit palette based graphics hardware, which will be displayed using a palette appropriate to the GRP resource (the palette itself not forming any part of the GRP object). The {X,Y} location on the pixmap where the frame is to be displayed should be pre-calculated as per: **PixmapBaseAddress+X+(PixmapWidth*Y)** then supplied to the procedure as the destination address.

```
GRPdecodeFrame  proc  grpBase :DWORD,   \; base address of the GRP
                      frame   :DWORD,   \; frame number (0 based)
                      daddr   :DWORD,   \; destination address
                      dwidth  :DWORD,   \; width of destination pixmap
                      mirror  :DWORD     ; mirror the frame? (bool)
  pushad

  ; find Frame Header
  mov ebx,grpBase
  mov eax,frame
  lea ecx,[ebx+6+eax*8]

  ;edi = frame offset adjusted write address
  mov edi,daddr

  ;YOffset
  xor eax,eax
  mov al,BYTE ptr[ecx+1]
  mul dwidth
  add edi,eax

  ;XOffset
  xor edx,edx
  mov dl,BYTE ptr[ecx]
  add edi,edx

  ;ebx = address of Data Header
  mov esi,[ecx+4]
  add ebx,esi

  ;edx = line width
  mov dl,[ecx+2]

  ;ecx = number of lines
  movzx ecx,BYTE ptr[ecx+3]

  ;esi = address of current LineStartOffset
  mov esi,ebx

  ;get mirror arg before using ebp
  mov eax,mirror

  ;ebp = dwidth
  mov ebp,dwidth

  test eax,eax
  jnz do_mirror

  cld
  ;decode line
  next_line:

    push ecx
    push edx
    push esi
    push edi

    movzx esi,WORD ptr[esi]   ; get the LineDataOffset
    add esi,ebx               ; add the GRP base address
```

```
  ; decode instruction
  next_instruction:

    ; get code byte
    mov cl,[esi]
    inc esi

    ; SHIFT instruction
    cmp cl,80h
    jbe not_shift
    sub cl,80h
    add edi,ecx
    sub dl,cl
    jz done_line
    jmp next_instruction
    not_shift:

    ; REPEAT instruction
    cmp cl,40h
    jbe not_repeat
    sub cl,40h
    mov al,[esi]
    inc esi
    sub dl,cl
    jz @F
      rep stosb
      jmp next_instruction
    @@:
    rep stosb
    jmp done_line
    not_repeat:

    ; PIXEL instruction
    sub dl,cl
    jz @F
    rep movsb
  jmp next_instruction
  @@:
  rep movsb

  done_line:
  pop edi
  add edi,ebp

  pop esi
  add esi,2

  pop edx
  pop ecx

loop next_line
popad
ret



; ============================================
; = Decode the frame mirrored abut the Y-axis =
; ============================================
do_mirror:

; set the direction flag so stosb works backwards
std

; add LineWidth-1 to the output pointer
; ( start at the end of the line and write backwards )
add edi,edx
dec edi
```

```
  ;decode line
  next_linem:
    push ecx
    push edx
    push esi
    push edi

    movzx esi,WORD ptr[esi]     ; get the LineDataOffset
    add esi,ebx                 ; add the GRP base

    ; decode instruction
    next_instructionm:

    ; get code byte
    mov cl,[esi]
    inc esi

    ; SHIFT instruction
    cmp cl,80h
    jbe not_shiftm
      sub cl,80h
      sub edi,ecx
      sub dl,cl
      jz done_linem
      jmp next_instructionm
    not_shiftm:

    ; REPEAT instruction
    cmp cl,40h
    jbe not_repeatm
      sub cl,40h
      mov al,[esi]
      inc esi
      sub dl,cl
      jz @F
      rep stosb
      jmp next_instructionm
      @@:
      rep stosb
      jmp done_linem
    not_repeatm:

    ; PIXEL instruction
    sub dl,cl

    ; can't use movsb here  -  esi must inc / edi must dec
    @@:
      mov al,[esi]
      mov [edi],al
      inc esi
      dec edi
    loop @B
    test dl,dl
    jnz next_instructionm

    done_linem:
    pop edi
    add edi,ebp

    pop esi
    add esi,2

    pop edx
    pop ecx

  loop next_linem
  cld
  popad
  ret
GRPdecodeFrame  endp
```