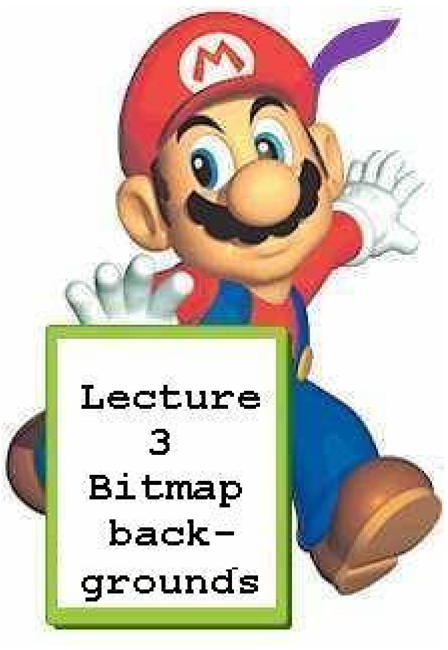
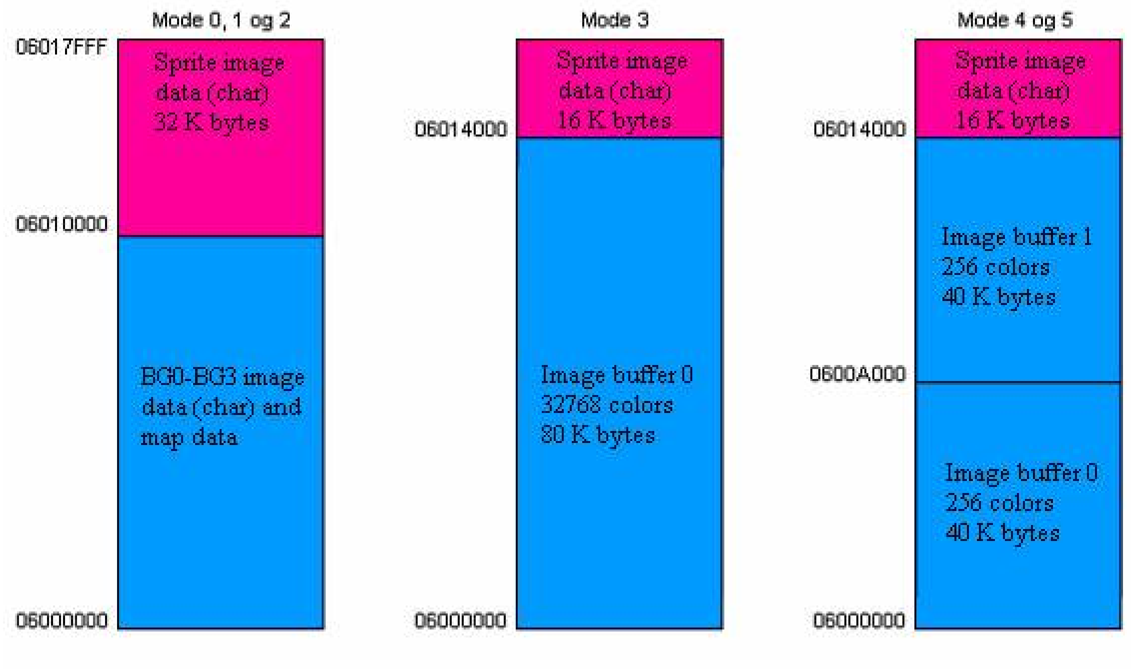


Overview of VRAM in Different Modes.

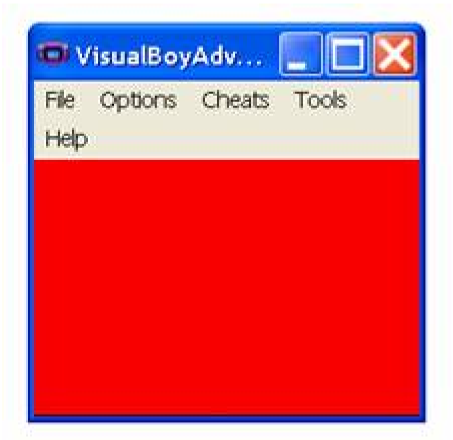

Mode 0, 1 og 2
06017FFF
Sprite image data (char) 32 K bytes
06010000
BG0-BG3 image data (char) and map data
06000000
Mode 3
06014000
Sprite image data (char) 16 K bytes
Image buffer 0 32768 colors 80 K bytes
06000000
Mode 4 og 5
06014000
Sprite image data (char) 16 K bytes
Image buffer 1 256 colors 40 K bytes
0600A000
Image buffer 0 256 colors 40 K bytes
06000000

Your first gba program

Here is the first program, copy it into notepad and store it as Test.cpp. Try to compile it by double clicking the Make.bat file, and run the Test.gba on the emulator. You will probably have to go back to the end of lesson 2 to get the details on how this should be done. Remember that you need to put all those header files into your folder (copy them in, and store them with the same name as in lesson 2).

```
#include "gba.h"

int main()
{
        u16 x;
        SetMode(MODE_3 | BG2_ENABLE);

        for(x=0; x<(240*160); x++)
        {
                FrontBuffer[x]=RGB(31, 0, 0);
        }
        return 0;
}
```

If you were able to compile and run this example, you should get a red screen on your gba emulator as shown below.



The first line in this program imports all those header files I told you about in lesson 2, and it looks like this.

#include "gba.h

The first line in the main() function looks like this.

SetMode(MODE_3 | BG2_ENABLE);

SetMode is a macro from the header file gba_video.h, and is used to set which video mode we are going to use in our code. In this example we choose to use mode 3 and that has a graphical resolution of 240x160 in 32768 colors. We also choose background 2, because this is the only background available in mode 3 (look in lesson 1 for overview on this). The way we set modes and backgrounds might be hard to understand if you have not seen this kind of thing before, but I will try to give an explanation here.

The macro SetMode is defined as follows.

#define SetMode(mode) REG_DISPCNT = (mode)

This means that it takes the value that we input after SetMode and assigns that value to the register defined as REG_DISPCNT.

REG_DISPCNT is a 16 bit register at address 0x4000000, it is maybe the most important register in the GBA.

On a bit level, REG_DISPCNT looks like this.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XX	XX	XX	OBJ	BG3	BG2	BG1	BG0	FB	OM	HB	DB	XX	MODE		

Bits 0-2 Decides which graphics mode the screen is going to have, a value between 0 and 5. **Bit 3** Not used **Bit 4** Decides which buffer we are going to use when using double buffering in mode 4 and 5.

Bit 5 Gives us the possibility to update OAM (sprite attribute data) during horizontal blank
Bit 6 Set the mapping mode for sprites 1D=0 and 2D=1.
Bit 7 Screen goes white (Forced Blank)
Bit 8-12 Set which background(s) we want to use, and set sprites on or off.
Bit 13-15 Not used.

If we look at MODE_0 and BG2_ENABLE in gba_video.h you might see that the value we have set for MODE_3 is 0x3, or as binary;

0000 0000 0000 0011

The value of BG2_ENABLE is 0x400 which is binary;

0000 0100 0000 0000

When we write SetMode (MODE_3 | BG2_ENABLE);

The following will happen;

```
       0000 0000 0000 0011          (MODE_3)
 OR    0000 0100 0000 0000          (BG2_ENABLE)
 =>    0000 0100 0000 0011
```

The operator | executes a logical OR between the two values (0x3 and 0x400). The result is then put into REG_DISPCNT, and then we are ready to write graphics to the screen.

The next line of the program.

```
u16 x;
```

This is only to define an unsigned short variable that is called x. u16 is defined as unsigned short, unsigned means that the variable can not contain negative numbers.

The gba screen (which is 240x160 pixels) is drawn the same way as a regular television screen, which means that it starts drawing in the upper left corner (pixel 0,0) and moves toward the right side of the screen. When the pixel is at pixel (239,0) it goes back to the left side of the screen (this is called HBLANK) and starts all over again at pixel (0,1). From here it moves right to pixel (239,1). In this way the whole gba screen is drawn line by line until it gets to point (239,159) then its a little break (called VBLANK) before it moves back to pixel (0,0) and starts drawing all over again. The screen is drawn about 60 times each second.

The next new thing in this program is FrontBuffer, which is a pointer to the video memory (VRAM). A pointer can be used with an index to reference data in the same way as an array, and that is what we are doing in this line.

```
FrontBuffer[x]=RGB(31, 0, 0);
```

The pointer points to different pixels on the screen, and the value that comes from RGB(31, 0, 0) is used to set the color of this specific pixel, FrontBuffer[x] (remember its 32768 colors available in mode 3).

I am setting the color of every index with the RGB macro that was described in lesson 2. As you might see from the program I have set the red value to 31, that is the maximum, and the two other values green and blue are 0. This is why all the pixels on the screen are red when you run the program.

The For-loop is actually indexing all the indexes in FrontBuffer and setting them all to the RGB value for red.

```
for(x=0;x<(240*160);x++)
```

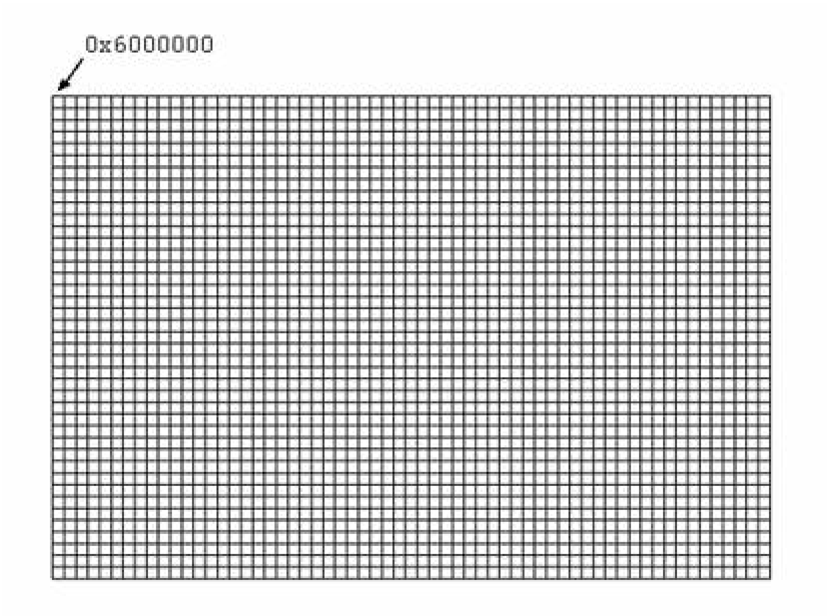
The last program line.

```
return 0;
```

This is because in c we use int main() in the start of the program. Lots of people use void main() in c++ programs, but here we have to use int main(), anything else wont work.

I will try to explain a little more exactly how the screen memory (VRAM) in mode 3 works. Each little square in the picture underneath is an address in the screen memory and is made up from an 8 bit value. The gba can only write 16 bit values to memory at a time. Because of this we had to define FrontBuffer as a u16, so that each time we write a 16 bit value to an index it will be split up into two parts, each of 8 bits, that will be written to two adjacent addresses.
Something you should keep in mind is that the gba only have 15 bit colors (2^15=32768), the most significant bit is ignored.

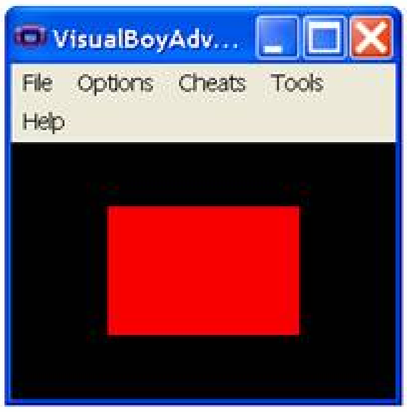
Here is an example: VRAM starts at 0x6000000, which means the pixel in the upper left corner (pixel 0,0) has address 0x6000000. If we now write a 16 bit value to FrontBuffer[0], that is the same as address 0x6000000, this value will be stored in the two addresses 0x6000000 and 0x6000001. In this way we see that each of the addresses get an 8 bit value. If we then write to the next pointer index that is FrontBuffer[1] and start at address 0x6000002 (because its a 16 bit pointer, and each address only has space for 8 bit). This value, will then be split in two and stored at address 0x6000002 and 0x6000003. It goes on like this until we have addressed the whole screen.



This might seem a little hard to address single points on the screen. I mean what if you want to place a pixel at (12,20)? Well if you use this little ‘trick’ you will be able to exactly tell the gba where you want your graphics.

```
#include "gba.h"

int main()
{
        u8 x,y;
        SetMode(MODE_3 | BG2_ENABLE);

        for(y=40;y<120;y++)
        {
                for(x=60;x<180;x++)
                {
                        FrontBuffer[(y*240)+x]=RGB(31, 0, 0);
                }
        }
        return 0;
}
```

This program is pretty much the same as the first, but we have included two loops and we have exchanged FrontBuffer[x] with FrontBuffer[(y*240)+x]. The extra loop is because we want to use two coordinates (x and y) to reference the pixel. We have used the loops to loop through the pixels we want to draw so that we get a rectangle on the screen. The reason we have exchanged FrontBuffer[x] with FrontBuffer[(y*240)+x] is that we know that gba has 240 (0-239) pixels horizontally on the screen, if we set FrontBuffer[240] we will get a pixel at (0, 1). From this we see that by multiplying the y value by 240 we will get on the desired y coordinate (0, y). If we then add the x value we want to place our pixel, we will end up with the right address in FrontBuffer. Because of this the formula for a chosen point within x = 0-239 and y = 0-159 will be (y*240)+x.



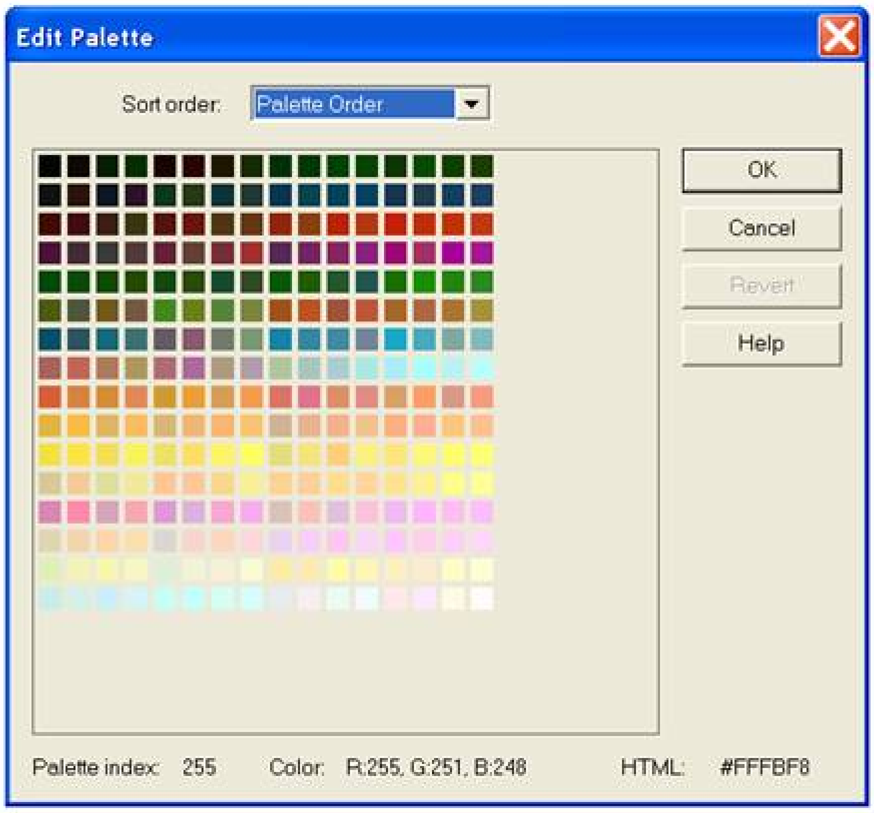
We have now looked at mode 3 and how you can use this mode to show graphics on the screen. We have set 15 bit colors into the address area from 0x6000000-0x6013FFF. As we can see this takes up a lot of space (75Kb), and uses almost all of the VRAM (96Kb). We have a total of 240x160 = 38400 individual pixels that can have whatever color we want from a palette of about 32768 colors. This is the main reason mode 3 uses this much memory (the computer has to store a 16 bit value for each pixel on the screen).

We are now going to have a look at mode 4 which at first seems pretty much like mode 3 but with a big difference. Mode 4 uses something called a palette, this palette contains a maximum of 256 colors chosen (from the 32768 available) by the programmer. Gba uses two palette sizes, the smallest is 16 colors, and the biggest is 256. The 16 bit colors we stored in VRAM in the last example is now stored in palette ram. This is a special memory area from 0x5000000-0x50001FF of 512 bytes that contains either one palette with 256 colors or 16 palettes with 16 colors each. We can set the colors we wish to use in our graphics using these palettes. This works the way that the color we input at position 0x5000000 in the palette gets nr 0 in the palette, the color at address 0x5000002, gets nr nr l, and so on.

When we choose how the picture is going to be when we input the values (the palette number of the color we wish to use) into VRAM, these automatically use the colors from the palette if mode 4 is set, it is done by the hardware. The main advantage of using a palette is that we save VRAM, and that we are able to write two pixels at a time (you haven’t forgotten that you always have to write 16 bit to VRAM), but this time we write two palette entry numbers (that is two colors) instead of only one 16 bit color. The disadvantage of doing this is that we can only use 256 different colors on the screen at once.

We use the same part of memory in mode 4 as in mode 3, but we only use half of it, because we only write one byte for each pixel on the screen. In mode 4 we only need to use VRAM from 0x6000000-0x6009FFF (37,5Kb), this is exactly half of what we used in mode 3. The memory area from 0x600A000-0x6013FFF (37,5Kb) is now free, and can actually be used as a buffer if we want to use double buffering/page flipping. This part of the memory works exactly like the first. We have the opportunity to decide which part of VRAM should be shown on the screen, by manipulating bit 4 in REG_DISPCNT.

This is a mode 4 picture and the palette for this picture.



Palette index: 255 Color: R:255, G:251, B:248 HTML: #FFFBF8

We will now have a look at a programming example that shows a picture on the gba screen.
You can choose whatever picture you want, but it has to be exactly 240x160 pixels. The picture has to be converted to a format the gba understands. From what I just have told you, we need palette and a image data to show our picture. There are alot of programs to convert from PC to gba. We are going to use a program called PCX2GBA, this converter program demands that the file you are going to convert has these properties.

- Maximum 256 colors.
- Image size exactly 240x160
- PCX image format

To be able to convert the picture you just drag the picture icon with your mouse pointer over the PCX2GBA icon and drop it. You will then see a small dos window telling you the size of the image. If you press a key this window will dissappear. A new file will then appear in your directory with the same name as your image file but with the extension .h. This file will contain all the information needed to show the picture on the gba.

Its very important that the name of the image file does not start with a number, because the name of the image file will also be the name of a table in the .h file you created, and as you know we cant use variables starting with a number. The converted image will look something like this.

```
/*********************************************\
*      bilde.h                         *
*         by dovotos pcx->gba program        *
/*********************************************/

#define  bilde_WIDTH   240
#define  bilde_HEIGHT  160

const u16 bildeData[] = {        0x4909, 0x1B1B, 0x2767, 0x1B1B, 0x1B1B, 0x1B49, 0x1F27, 0x1B49,
                                0x1B1F, 0x4932, 0x091B, 0x601F, 0x1B1B, 0x1B16, 0x1B1B, 0x1B27,
                                0x161B, 0x161B, 0x1F1B, 0x351B,           0x161F, 0x091B, 0x6235,
                                0x1B13, 0x6016, 0x4616, 0x1B1B, 0x6216, 0x1F35, 0x6760, 0x6D63,
                                0x6264, 0x671B, 0x1F62, 0x6269, 0x6962, 0x6869, 0x6C63, 0x6469, 0x6369,
                                0x6464, 0x6969, 0x696D, 0x696D, 0x6969, 0x6969, 0x8E77, 0x5C77, 0x705F,
                                0x8071,...};


const u16 bildePalette[] = {    0x0000, 0x0021, 0x0060, 0x00A0, 0x0003, 0x0005, 0x0044,
                0x00A3, 0x04C0, 0x00C0, 0x0100, 0x0100, 0x00C2, 0x0121,
                                0x00E2, 0x00E3, 0x0441, 0x0445, 0x1041, 0x1045, 0x0CC1, 0x08E4,
                                0x1CC1, 0x18C4, 0x28C0, 0x2901, 0x2D00, 0x3100, 0x28C2, 0x24E4,
                                0x2CE2, 0x30E4, 0x0028, 0x0828, 0x0867, 0x08C7, 0x044A, 0x044C,
                                0x08C9, 0x08CC, ...};
```

Here you see what I mean when I said that the file name is the same as the start of the array name. Both arrays bildePalett[] and bildeData[] are actually much longer than shown here, but its really not important to see all the numbers.

This program uses the image data to show the image on the gba.

```
#include "gba.h"
#include "bilde.h"

int main()
{
        u16 loop, x;
        SetMode( MODE_4 | BG2_ENABLE );

        for(loop=0;loop<256;loop++)
        {
                BG_PaletteMem[loop]=bildePalette[loop];
        }

        for (x=0; x<(120*160); x++)
        {
                FrontBuffer[y] = bildeData[y] ;
        }
        return 0;
}
```

The first new thing in this program is that we include the header file bilde.h, which contains the two arrays bildeData and bildePalette. This is the information we need to get the image on the screen.

```
#include "bilde.h"
```

We set graphics mode 4, and background 2 on, which is the only background available in mode 3, 4 and 5.

```
SetMode( MODE_4 | BG2_ENABLE );
```

The next lines load the content from the table bildePalette in memory, at address 0x5000000-0x50001FF. BG_PaletteMem is a 16 bit pointer to address 0x5000000. We use this as an array, the same way we did when we used FrontBuffer, and load the colors into palette memory. We can see that the loop goes all the way up to 256, because that is the amount of colors needed to show this image. If the palette only consisted of only 100 colors, our loop could stop at 100.

```
for(loop=0;loop<256;loop++)
{
        BG_PaletteMem[loop]=bildePalette[loop];
}
```

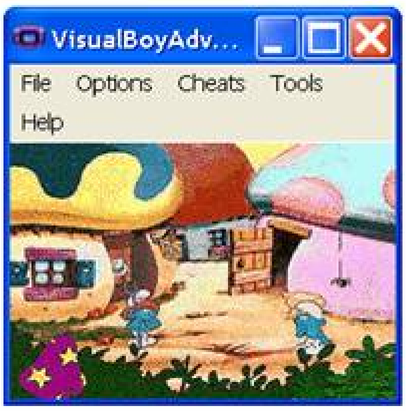
The next loop loads the content of the bildeData table into VRAM so that the gba will know where on the screen to put the different colors. We are only using the memory area from 0x6000000-0x6009FFF (37,5Kb). This loop looks very much like the one used in the first program example with mode 3, but in this example we use 120 instead of 240. The reason for this is that when we write only half as many bytes (remember each color is a 1 byte entry into the palette) to VRAM as we do in mode 3. Because we always write 16 bit at a time, and the image converter has taken this into consideration, and put the two adjacent bytes together to form a 16 bit number that we can directly load into VRAM to get the picture on the screen..

```
for (x=0; x<(120*160); x++)
{
        FrontBuffer[y] = bildeData[y] ;
}
```

We can also address the screen with x and y, coordinates (as we did in the second example with mode 3), the result is the same but because we use nested loops this way is probably slower then the first one.

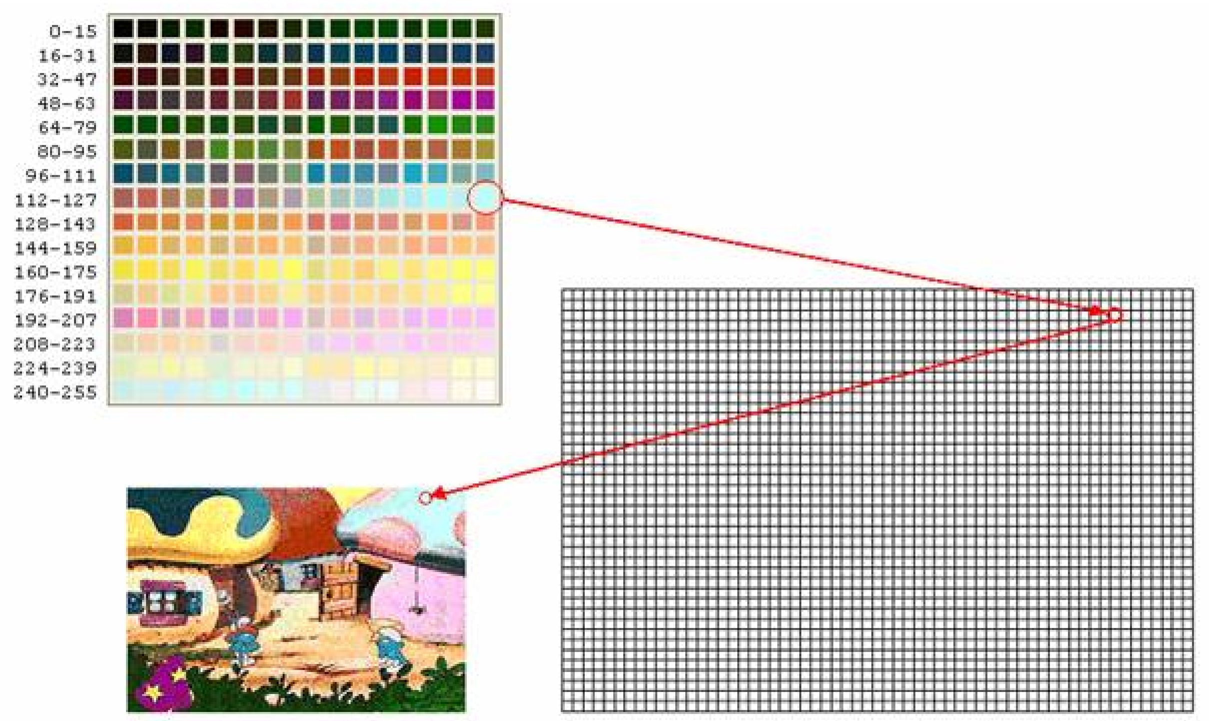
```
#include "gba.h"
#include "bilde.h"

int main()
{
        u8 x,y;
        u16 loop;
        SetMode( MODE_4 | BG2_ENABLE );

        for(loop=0;loop<256;loop++)
        {
                BG_PaletteMem[loop]=bildePalette[loop];
        }
        for (y=0; y<160;y++)
        {
                for (x=0; x<120;x++)
                {
                        FrontBuffer[(y*120)+x] = bildeData[(y*120)+x] ;
                }
        }
}
```

The result should be something like this.



This image is stored the way that the value that is at address 0x6000000 is one byte that refers to a color in the palette. If the value 2 had been stored at address 0x6000000, the gba would have figured out which the color that was nr 2 in the palette and put this in the pixel at point (0, 0). Address 0x6000001 contains a reference to the color in point (1, 0), address 0x6000002 contains a reference to the color in point (2,0) and so on.

Here we can see how color nr 127 in the palette is stored in VRAM, and where it is used in the picture (actually several places).


0-15
16-31
32-47
48-63
64-79
80-95
96-111
112-127
128-143
144-159
160-175
176-191
192-207
208-223
224-239
240-255