Programming with C/C++ 4

Computer Games in 2D (5SD814)

Goals

Lecture goals:

- understand alpha blending
- how digital images are structured
- describe common image formats

Image

- array of picture elements (pixels)
- can be 1-, 2- or 3-dimensional
  - width only
  - width and height
  - width, height and depth
- image data can be compressed
  - on disk, i.e. decompressed before use (png, jpeg, ...)
  - in memory, i.e. used as is (textures)

Image Formats

- raster graphic format
  - bitmap images
  - an array of pixels
  - good memory storage mapping
- vector graphic format
  - a collection of points/vectors connected by lines and curves
  - advantages of being resolution independent

Raster Graphics

- a rectangular grid of pixels
- a 2D array with width and height
- each pixel color specified by a number of bits
  - accessed using x and y
  - $(0, 0)$ top left corner

Pixel Formats

- common pixel formats in bits per pixel (bpp)
  - monochrome (1 bpp)
  - gray scale (8 bpp)
  - palettized (8 bpp)
  - full color (8-32 bpp)

Pixel

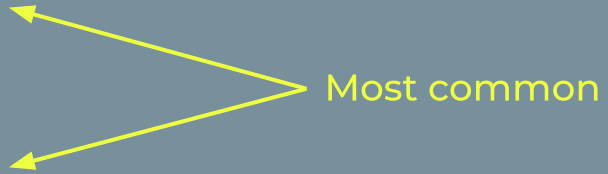
- a pixel represents a *color*
- a pixel can have 1-4 color *components/channels*
  - red, green, blue and alpha
- a pixel can be in integer or floating point format
- *color depth* is the number of bits needed to represent a pixel
  - another name for it is *bit depth*

Color Depth (examples**)**

- examples of different non-compressed color depths (integer)
  - `R5 G6 B5` (16-bits)
  - `R8 G8 B8` (24-bits)
  - `R5 G5 B5 A1` (16-bits)
  - `R4 G4 B4 A4` (16-bits)
  - `R8 G8 B8 A8` (32-bits)
  - `R10 G10 B10 A2` (32-bits)

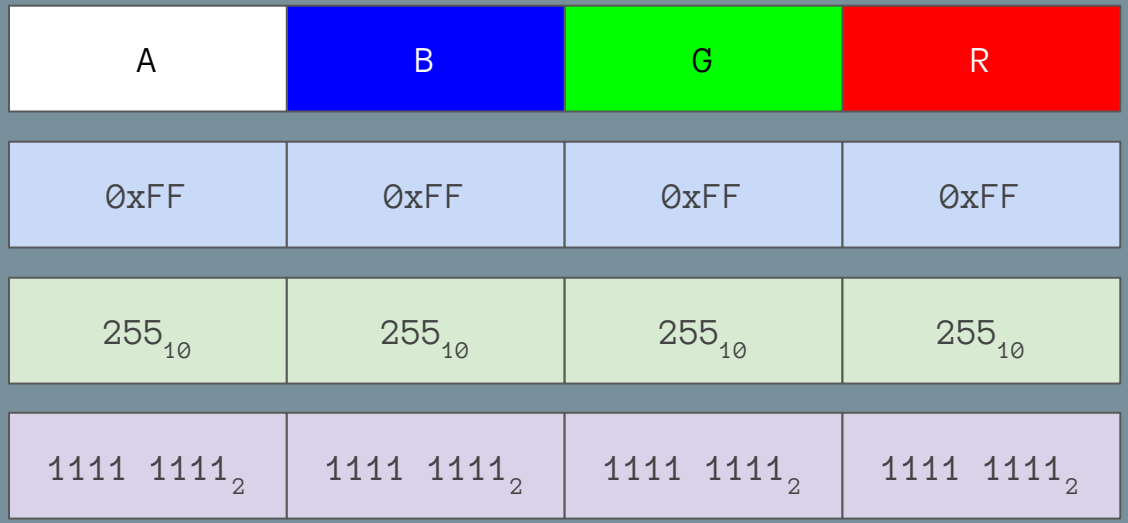
Color Depth (examples)

- examples of different non-compressed color depths (integer)
  - `R5 G6 B5` (16-bits)
  - `R8 G8 B8` (24-bits)
  - `R5 G5 B5 A1` (16-bits)
  - `R4 G4 B4 A4` (16-bits)
  - `R8 G8 B8 A8` (32-bits)
  - `R10 G10 B10 A2` (32-bits)



Color Depth (examples)

Example of a 32-bit R8G8B8A pixel stored in memory.



Alpha Blending

- combining foreground images with a background image
- can also be called alpha compositing
- uses a alpha component for transparency or opacity
  - coverage or occlusion information
  - partial or full
  - smooth edge transitions
- many different operations, or blend modes, can be applied
  - we only care about additive blending

Alpha Blending (formula)

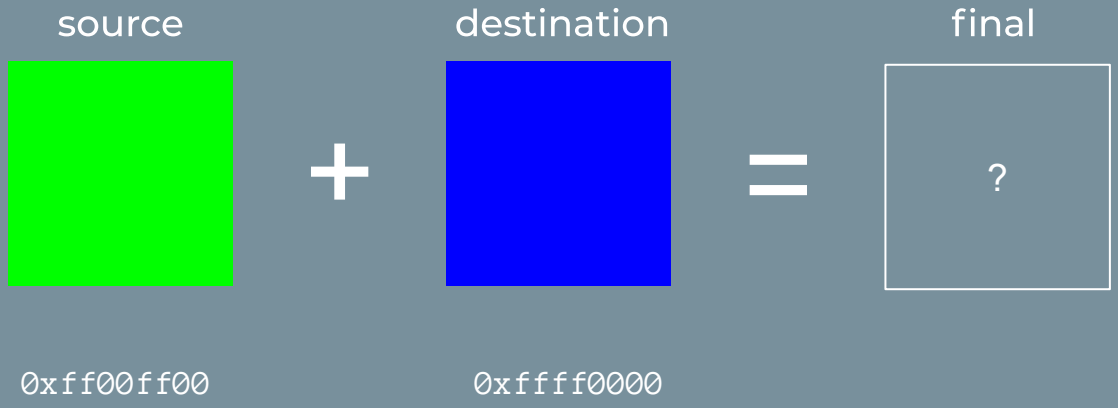
```
final = src * src.a + dst * (1 - src.a)
```

Alpha Blending (formula)

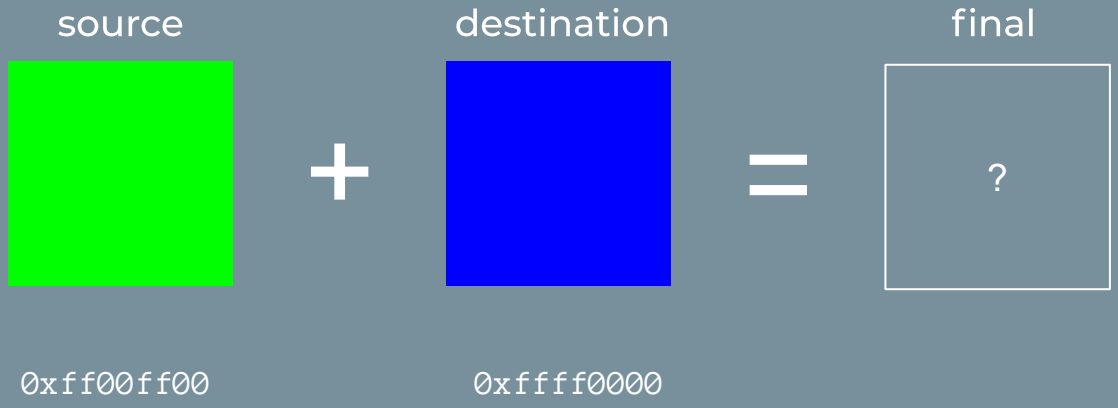
```
final = src * src.a + dst * (1 - src.a)
```

$$
\begin{aligned}
R_{final} &= R_{src} * A_{src} + R_{dst} * (1 - A_{src}) \\
G_{final} &= G_{src} * A_{src} + G_{dst} * (1 - A_{src}) \\
B_{final} &= B_{src} * A_{src} + B_{dst} * (1 - A_{src})
\end{aligned}
$$

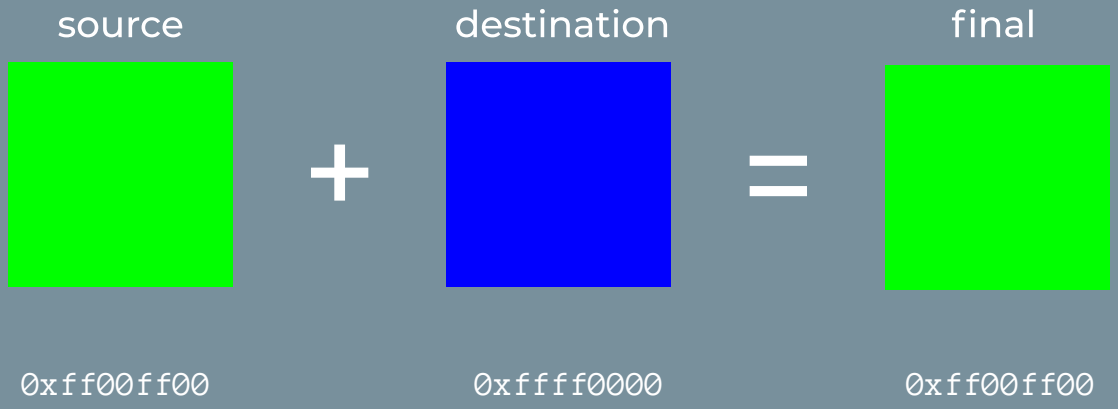
Alpha Blending (example)



Alpha Blending (example)



Alpha Blending (example)



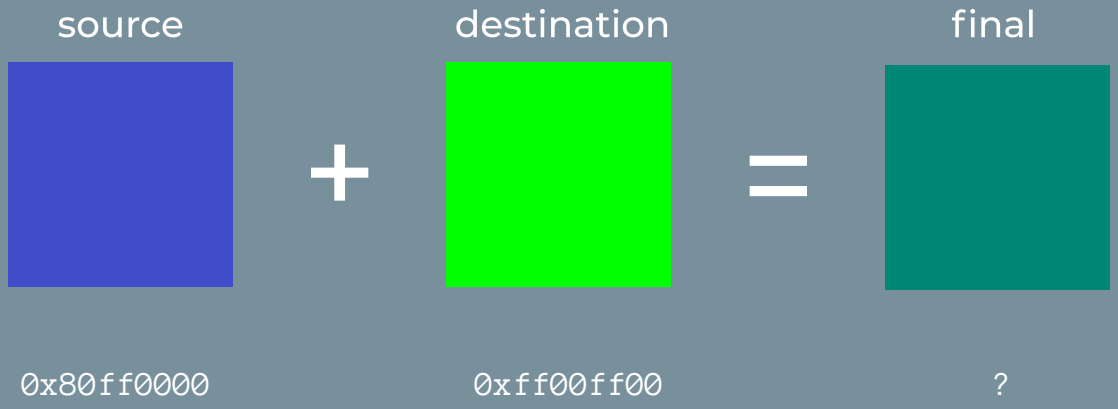
Alpha Blending (example)

0xff00ff00 0xffff0000 0xff00ff00

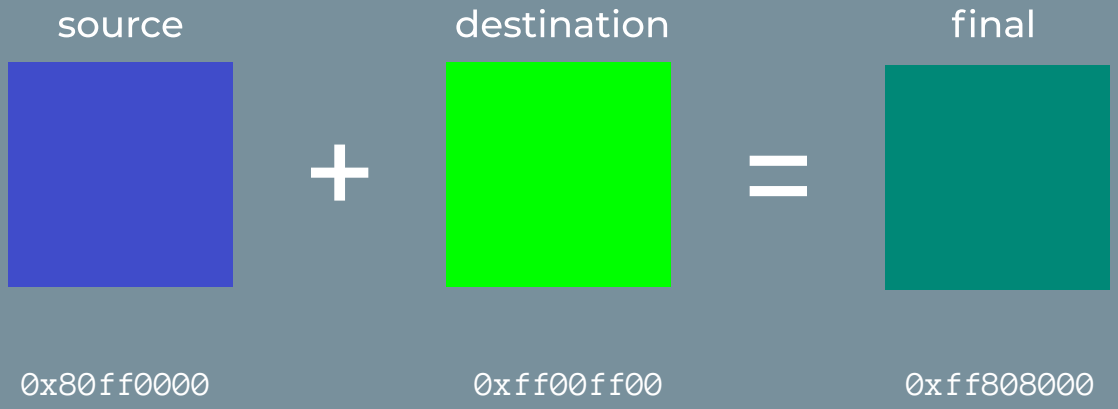
B =	0 * 1.0	+	255 * (1.0 - 1.0)	=	0	=	0x00
G =	255 * 1.0	+	0 * (1.0 - 1.0)	=	255	=	0xff
R =	0 * 1.0	+	0 * (1.0 - 1.0)	=	0	=	0x00

src **dst**

Alpha Blending (example)



Alpha Blending (example)



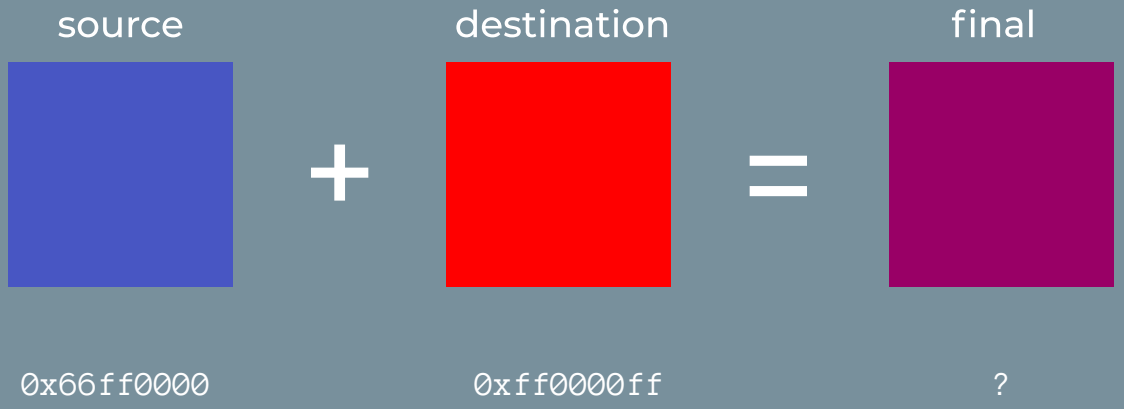
Alpha Blending (example)

0x80ff0000 0xff00ff00 0xff808000

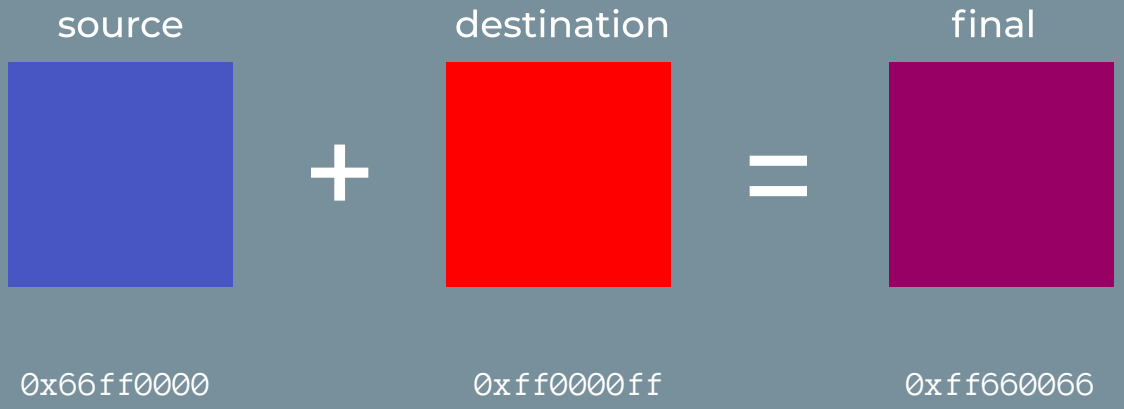
```
B = 255 * 0.5 +   0 * (1.0 - 0.5) =  128  =  0x80
G =   0 * 0.5 + 255 * (1.0 - 0.5) =  128  =  0x80
R =   0 * 0.5 +   0 * (1.0 - 0.5) =    0  =  0x00
```

src **dst**

Alpha Blending (example)



Alpha Blending (example)



Alpha Blending (example)

0x66ff0000 0xff0000ff 0xff660099

```
B = 255 * 0.4 +   0 * (1.0 - 0.4) =  102 =  0x66
G =   0 * 0.4 +   0 * (1.0 - 0.4) =    0 =  0x00
R =   0 * 0.4 + 255 * (1.0 - 0.4) =  153 =  0x99
```

src **dst**

Image Subrectangle

How do we find the subrectangle that we want to draw?

Image Subrectangle

Source image.

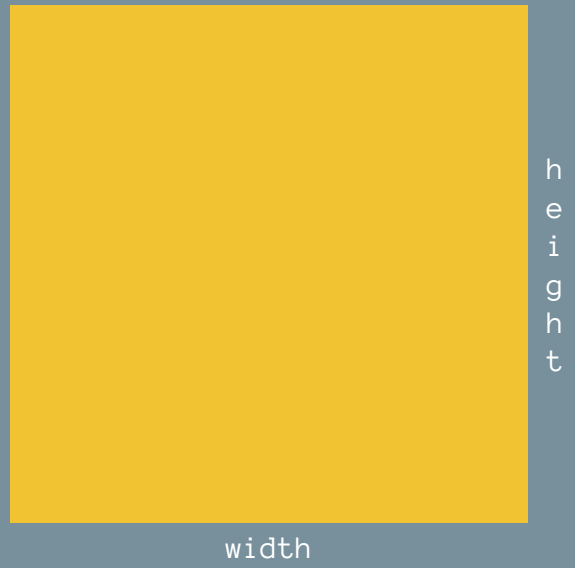


Image Subrectangle

Source image and subrect.

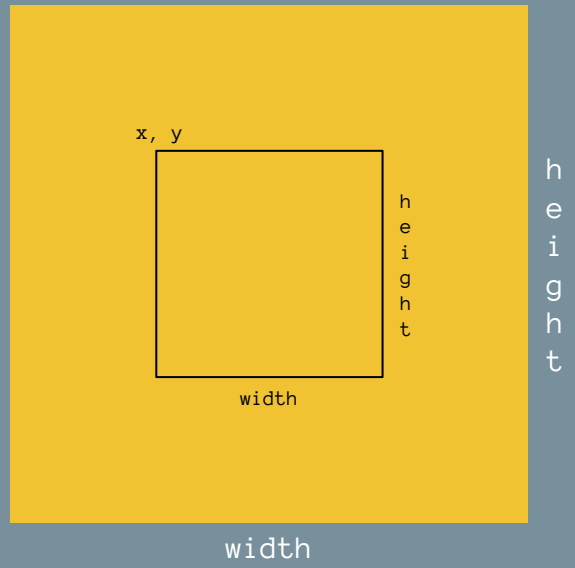
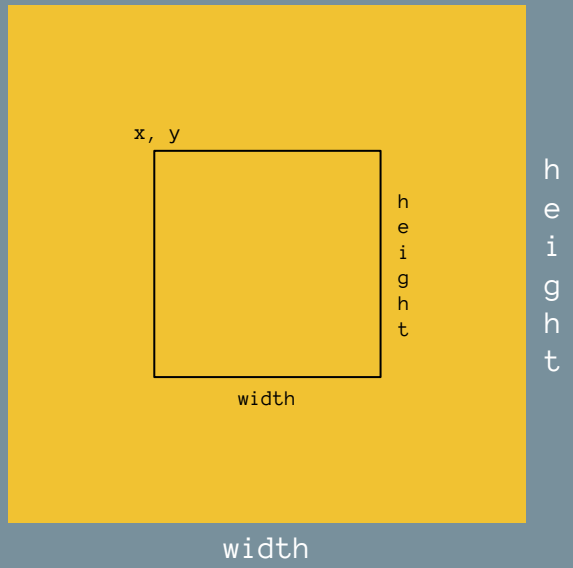


Image Subrectangle

Source image and **subrect**.

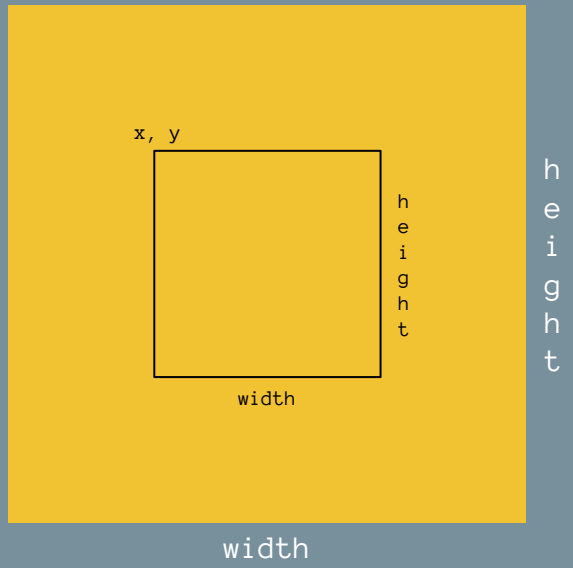



```
verify subrect

y_start = subrect.y
y_end   = subrect.y + subrect.height
x_start = subrect.x
x_end   = subrect.x + subrect.width
```

Image Subrectangle

Source image and **subrect**.




```
verify subrect

y_start = subrect.y
y_end   = subrect.y + subrect.height
x_start = subrect.x
x_end   = subrect.y + subrect.width

for y = y_start to y_end
  for x = x_start to x_end
    index = y * width + x
    pixel = source[index]
```

Common Image Formats

- Portable Network Graphics (.png)
  - Lossless compression using DEFLATE (Huffman + LZ77)
  - Complicated file format
  - Still raw image format when uncompressed
  - E.g. `1024x1024x4` = 4 MiBytes (RGBA 8-bit per component)
- Graphics Interchange Format (.gif)
  - Lossless compression (LZW)
  - Originally limited to 256 colors
  - Can have 8-bits per component (24-bits)
  - Supports animations

Common Image Formats

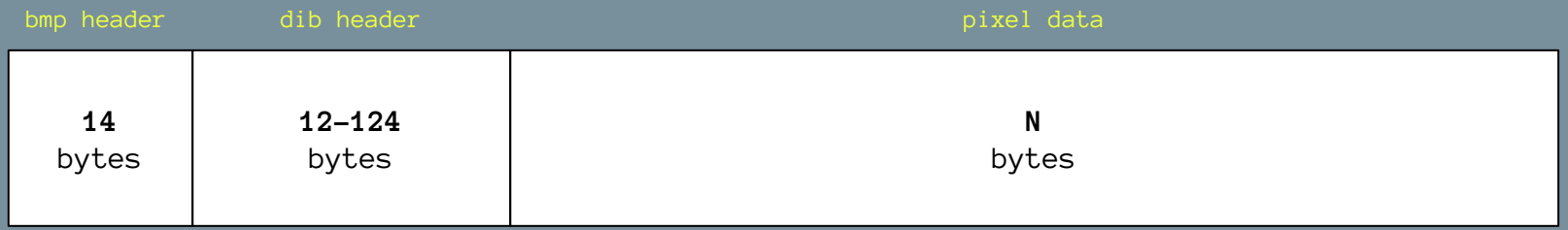
- JPEG (.jpg, .jpeg, ...)
  - Lossy compression using Discrete Cosine Transform
  - Mainly used for photographic images
  - Good compression rate
  - Blocky artifacts
- Truevision TGA (.tga)
  - 8-, 15-, 16-, 24-, or 32-bits per pixel
  - Lossless compression using Run-Length Encoding (RLE)
  - Simple file format!

Common Image Formats

- Windows Bitmap (.bmp)
  - 8-, 16-, 24- or 32-bits per pixel
  - Can be compressed
  - Pixel data stored upside down (historical reasons CRT monitors)
  - Simple file format!

Bitmap File Format

- Bitmap header (`14` bytes)
- DIB header (`12-124` bytes)
- Pixel data (`N` bytes)
  - Stored bottom row first!



Bitmap File Format (bmp header)

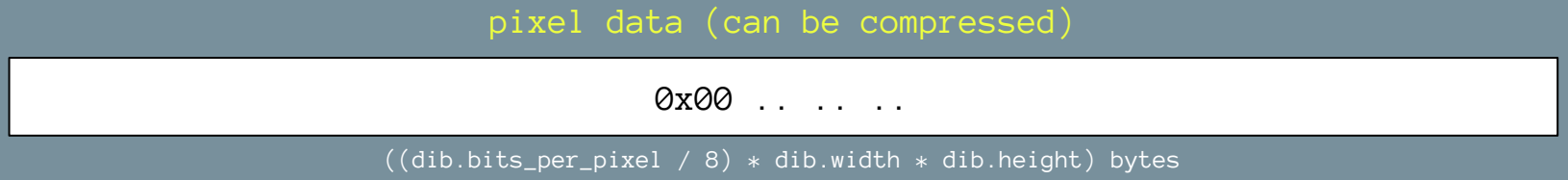
Bitmap File Format (dib header)

dib header size	image width	image height	color planes
0x28000000	0x00000000	0x00000000	0x0001
4 bytes	4 bytes	4 bytes	2 bytes

bits per pixel	compression method	image size	horizontal resolution
0x0000	0x00000000	0x00000000	0x00000000
2 bytes	4 bytes	4 bytes	4 bytes

vertical resolution	colors in palette	important colors used
0x00000000	0x00000000	0x00000000
4 bytes	4 bytes	4 bytes

Bitmap File Format (pixel data)



Questions?

Bonus #1 (bit manipulation & (and)**)**

UPPSALA UNIVERSITET

```
0xff & 0x07  =  0x07        1111 1111
                          & 0000 0111
                          -----------
                            0000 0111

0x67 & 0x11  =  0x01        0110 0111
                          & 0001 0001
                          -----------
                            0000 0001
```

Bonus #2 (bit manipulation | (or)**)**

```
0x83 | 0x30  =  0xb3      1000 0011
                         | 0011 0000
                         -----------
                           1011 0011

0x2f | 0x42  =  0x6f      0010 1111
                         | 0100 0010
                         -----------
                           0110 1111
```

Bonus #3 (bit manipulation << (shift left)**)**

```
0xff << 1  =  0x1fe          1111 1111 << 1
                            1 1111 1110

0x48 << 3  =  0x240          0100 1000 << 3
                           10 0100 0000
```

Bonus #4 (bit manipulation >> (shift right)**)**

```
0xff >> 1  =  0x7f        1111 1111 >> 1
                          0111 1111

0x48 >> 2  =  0x12        0100 1000 >> 2
                          0001 0010

0x74 >> 3  =  0x0e        0111 0100 >> 3
                          0000 1110
```

Bonus #5 (bit manipulation ^ (xor)**)**

```
0x93 ^ 0x31  =  0xa2      1001 0011
                        ^ 0011 0001
                        -----------
                          1010 0010

0x2f ^ 0x42  =  0x4d      0010 1111
                        ^ 0100 0010
                        -----------
                          0110 1101
```