Embedded Vision Design

EVD1 - Week 2

Image Fundamentals
Graphics Algorithms

By Hugo Arends

HAN_UNIVERSITY OF APPLIED SCIENCES

Image Fundamentals

- Functions for creating and deleting images
- Functions for converting images
- Functions for reading and writing pixels
- Basic image processing operators

- Scaling fast

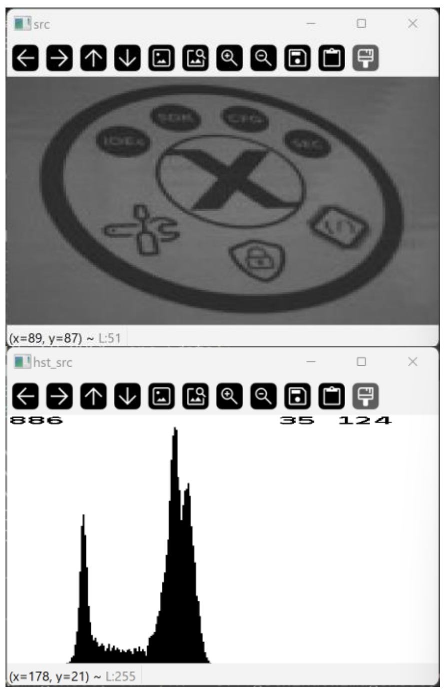
Scaling

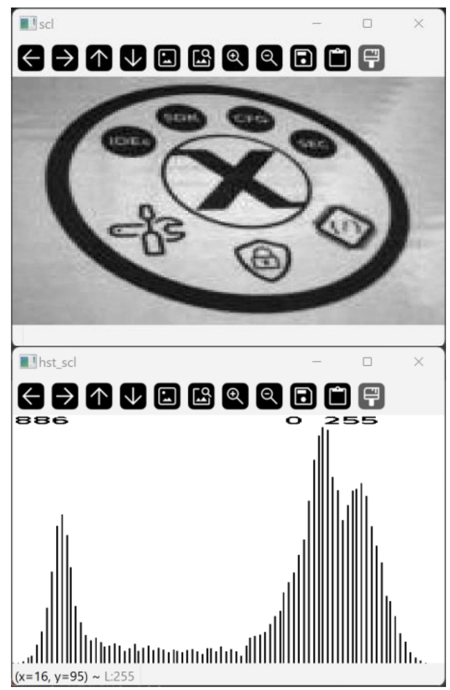
- Used to enhance contrast
- Used to scale larger pixel datatypes to smaller pixel datatypes e.g. float to basic
- Min-max scaling is defined as

$$p_{dst}(x, y) = \frac{dst_{max} - dst_{min}}{src_{max} - src_{min}} \cdot (p_{src}(x, y) - src_{min}) + dst_{min}$$

Scaling - example

$$p_{dst}(x,y) = \frac{255 - 0}{src_{max} - src_{min}} \cdot (p_{src}(x,y) - src_{min}) + 0$$





Scaling operation

$$p_{dst}(x,y) = \frac{dst_{max} - dst_{min}}{src_{max} - src_{min}} \cdot (p_{src}(x,y) - src_{min}) + dst_{min}$$

Requires a lot of operations for every destination pixel:

- Read all source pixels to determine src_{max} and src_{min}
- Calculate stretch factor $\frac{dst_{max}-dst_{min}}{src_{max}-src_{min}}$
- Calculate new pixel value $p_{dst}(x,y)$ (with a point number) and store the result

HAN_UNIVERSITY OF APPLIED SCIENCES

Performance optimization

- Optimize the implemented code for execution speed
- Several techniques discussed in random order

Performance optimization

Use the compiler optimization levels

- None (-O0)
- Optimize (-O1)
- Optimize more (-O2)
- Optimize most (-O3)
- Optimize for size (-Os)
- Optimize for Debug (-Og)

Performance optimization

Use the compiler optimization levels

For the given project there are two build configurations, making it easy to switch between optimization levels

- Optimize for debug (-Og)

- Optimize most (-O3)

TIP. Change the target Project > Build Configurations > Set Active

Performance optimization

Use the compiler optimization levels

For the given project there are two build configurations, making it easy to switch between optimization levels

- Optimize for debug (-Og)

- Optimize most (-O3)

TIP. Other optimization levels are configured in Project > Properties > C/C++ Build > Settings > MCU GCC Compiler

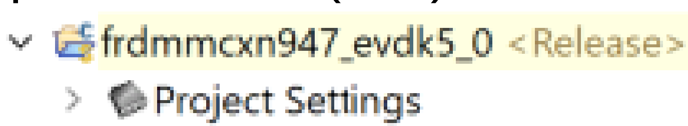
> Optimization > Optimization level

Performance optimization

- Optimize for debug (-Og)

~850 µs

- Optimize most (-O3)



~230 µs

```c
// Copy image
uint8_pixel_t *s = (uint8_pixel_t *)src->data;
uint8_pixel_t *d = (uint8_pixel_t *)dst->data;

dst->rows = src->rows;
dst->cols = src->cols;
dst->type = src->type;

for(int32_t r = src->rows-1; r >= 0; r--)
{
    for(int32_t c = src->cols-1; c >= 0; c--)
    {
        *d++ = *s++;
    }
}
```

Performance optimization

Use the built in FPU!

- Is already enabled in the given project, because it is required by the video driver
- Will you be using ‘doubles’ or ‘floats’?

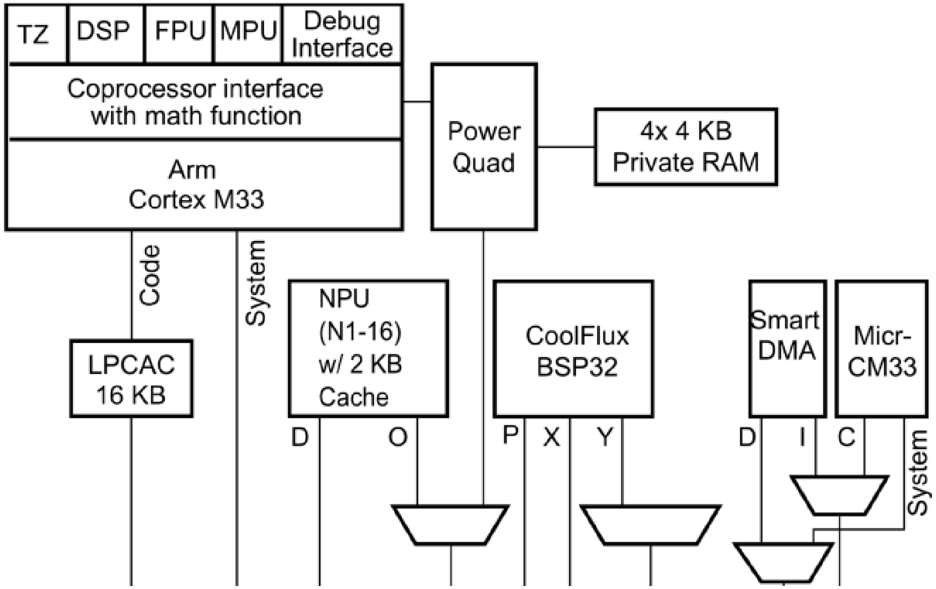
```
src->data[0] += 0.5;   // Is 0.5 a float or a double?
src->data[0] += 0.5f;  // Is 0.5 a float or a double?
```

- Alternative (if your hardware doesn’t support a hardware FPU): implement Fixed Point calculations

Performance optimization

Use register variables

- *Register variables require less memory access*
- *Especially useful for loop-counters, because these are accessed often*
- *Although taken care of by the compiler, you can explicitly use the register keyword*
- https://en.wikipedia.org/wiki/Register_(keyword)



Source: MCXNx4x datasheet

HAN_UNIVERSITY OF APPLIED SCIENCES

Performance optimization

Use pointers to the source and destination data instead of getter and setter functions

- Optimize most (-O3)

$$\sim 230\ \mu s$$

```
// Copy image
uint8_pixel_t *s = (uint8_pixel_t *)src->data;
uint8_pixel_t *d = (uint8_pixel_t *)dst->data;

dst->rows = src->rows;
dst->cols = src->cols;
dst->type = src->type;

for(int32_t r = src->rows-1; r >= 0; r--)
{
    for(int32_t c = src->cols-1; c >= 0; c--)
    {
        *d++ = *s++;
    }
}
```

Performance optimization

Use pointers to the source and destination data instead of getter and setter functions

- Optimize most (-O3)

$$\sim 4300\ \mu s$$

```c
// Copy image
uint8_pixel_t *s = (uint8_pixel_t *)src->data;
uint8_pixel_t *d = (uint8_pixel_t *)dst->data;

dst->rows = src->rows;
dst->cols = src->cols;
dst->type = src->type;

for(int32_t r = src->rows-1; r >= 0; r--)
{
    for(int32_t c = src->cols-1; c >= 0; c--)
    {
        setUint8Pixel(dst, c, r, getUint8Pixel(src, c, r));
    }
}
```

Performance optimization

Loop unrolling

- *Testing conditions of a loop takes instructions and hence execution time*

$$\mathbf{160 \times 120 = 19200}$$

```
uint8_pixel_t min=UINT8_PIXEL_MAX, max=UINT8_PIXEL_MIN;
uint32_t imsize = src->rows * src->cols;
uint8_pixel_t *s = (uint8_pixel_t *)src->data;

// Scan input image for min/max values
for(uint32_t i=0; i<imsize; ++i)
{
    if(*s < min){ min = *s; }
    if(*s > max){ max = *s; }
    ++s;
}
```

Performance optimization

Loop unrolling

- *Testing conditions of a loop takes instructions and hence execution time*

$$\frac{160 \times 120}{2} = 9600$$

```
uint8_pixel_t min=UINT8_PIXEL_MAX, max=UINT8_PIXEL_MIN;
uint32_t imsize = (src->rows * src->cols) / 2;
uint8_pixel_t *s = (uint8_pixel_t *)src->data;

// Scan input image for min/max values
for(uint32_t i=0; i<imsize; ++i)
{
    if(*s < min){ min = *s; }
    if(*s > max){ max = *s; }
    ++s;

    if(*s < min){ min = *s; }
    if(*s > max){ max = *s; }
    ++s;
}
```

Performance optimization

Loop unrolling

- *Testing conditions of a loop takes instructions and hence execution time*

0

```
uint8_pixel_t min=UINT8_PIXEL_MAX, max=UINT8_PIXEL_MIN;

uint8_pixel_t *s = (uint8_pixel_t *)src->data;

// Scan input image for min/max values
if(*s < min){ min = *s; }
if(*s > max){ max = *s; }
++s;

if(*s < min){ min = *s; }
if(*s > max){ max = *s; }
++s;

…

if(*s < min){ min = *s; }
if(*s > max){ max = *s; }
++s;
```

Performance optimization

Perform a calculation only a single time, however…

```
// Scale the output to basic image type
for(uint32_t i=0; i<imsize; ++i)
{
    *d++ = (uint8_pixel_t)((255.0f/(max-min)) * (*s++ - min) + 0.5f);
}
```

Optimize most (-O3)

$\sim \mathbf{3520}\ \boldsymbol{\mu s}$

```
// Scale the output to basic image type
float factor = 255.0f/(max-min);

for(uint32_t i=0; i<imsize; ++i)
{
    *d++ = (uint8_pixel_t)((factor) * (*s++ - min) + 0.5f);
}
```

Optimize most (-O3)

$\sim \mathbf{3520}\ \boldsymbol{\mu s}$

Performance optimization

Construct a lookup table (LUT)

- All src pixels with the same value, will get the same new value after a calculation

src

	0	1	2	3
0	4	4	4	4
1	5	5	5	5
2	6	6	6	6
3	7	7	7	7

dst

	0	1	2	3
0	0	0	0	0
1	85	85	85	85
2	170	170	170	170
3	255	255	255	255

$$p_{dst}(x,y) = factor \cdot (p_{src}(x,y) - src_{min}) + 0.5$$

- Instead of writing the result to the dst the image…

Performance optimization

Construct a lookup table (LUT)

- All src pixels with the same value, will get the same new value after a calculation

src

	0	1	2	3
0	4	4	4	4
1	5	5	5	5
2	6	6	6	6
3	7	7	7	7

dst

	0	1	2	3
0	0	0	0	0
1	85	85	85	85
2	170	170	170	170
3	255	255	255	255

$$LUT[index] = factor \cdot (index - src_{min}) + 0.5$$

- Store it in a table (an array), where $index$ are all graylevels in src

Performance optimization

Construct a lookup table (LUT)

- All src pixels with the same value, will get the same new value after a calculation

src

	0	1	2	3
0	4	4	4	4
1	5	5	5	5
2	6	6	6	6
3	7	7	7	7

dst

	0	1	2	3
0	0	0	0	0
1	85	85	85	85
2	170	170	170	170
3	255	255	255	255

index	0	1	2	3	4	5	6	7	8	9									255	*Gray values in src*
LUT	0	0	0	0	0	85	170	255	0	0	...	0	0	0	0	0	0	0	0	*Gray values in dst*

- The time-consuming calculations is performed only 256 times

Performance optimization

Construct a lookup table (LUT)

- All src pixels with the same value, will get the same new value after a calculation

src

	0	1	2	3
0	4	4	4	4
1	5	5	5	5
2	6	6	6	6
3	7	7	7	7

dst

	0	1	2	3
0	0	0	0	0
1	85	85	85	85
2	170	170	170	170
3	255	255	255	255

$$p_{dst}(x,y) = LUT[\, p_{src}(x,y) \,]$$

- And use the table to assign values to dst

Performance optimization

Use for a (local) counter 32-bit variables !

	31			0
uint32_t	00000000	00000000	00000000	00000000

- Example: count from 0 to 100
- Using a uin8_t is not efficient in a 32-bit microcontroller
- All registers and RAM are 32-bit.
- When using an uint8_t, the compiler must make sure that $255 + 1 = 0$
- This is an additional instruction (which one?) for each addition!!

```
uint8_t i = 0;

while(i < 100)
{
    // Do work here

    i++;
}
```

Performance optimization

Read/write as less as possible from/to RAM!

- If possible, read/write 4 pixels in a single cycle

```c
// Copy uint8_t image in chunks of four pixels
long int i = src->rows * src->cols / 4;
uint32_t *s = (uint32_t *)src->data;
uint32_t *d = (uint32_t *)dst->data;

dst->rows = src->rows;
dst->cols = src->cols;
dst->type = src->type;

while(i-- > 0)
{
    *d++ = *s++;
}
```

Performance optimization

Read/write as less as possible from/to RAM!

- If possible, read/write 4 pixels in a single cycle

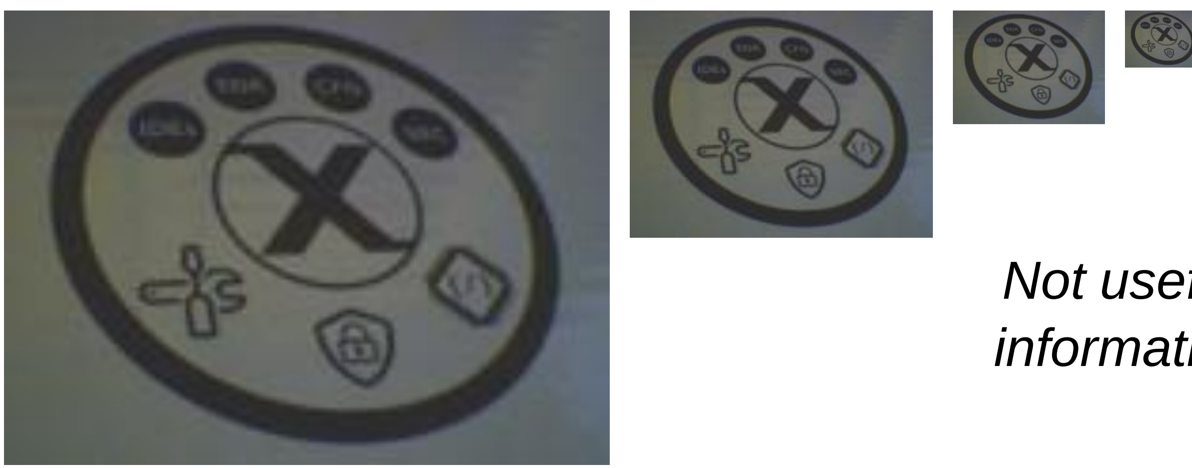
- Operations are executed on registers in the CPU.
- Memory access is a time consuming (and energy intensive) operation.
- Bit-shifting a 32-bit variable takes less execution time than reading 4 bytes from memory!

```
// src is a uint8_pixel_t image
uint32_t *s = (uint32_t *)src->data;

uint32_t four_pixels = *s++;

if((uint8_pixel_t)(four_pixels >> 0) > max){max = (uint8_pixel_t)(four_pixels >> 0);}
if((uint8_pixel_t)(four_pixels >> 8) > max){max = (uint8_pixel_t)(four_pixels >> 8);}
Etc.
```

Performance optimization

Use mipmaps (image pyramids)

- *Example: scale() (Optimize most (-O3))*
- **160x120: ~3520 µs**
- **80x60: ~880 µs**
- **40x30: ~220 µs**
- **20x15: ~60 µs**

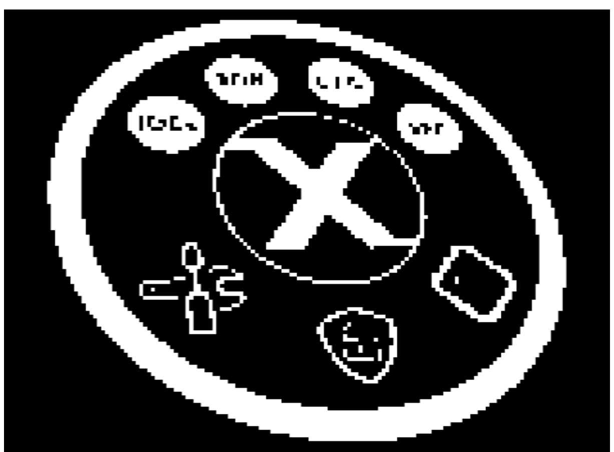
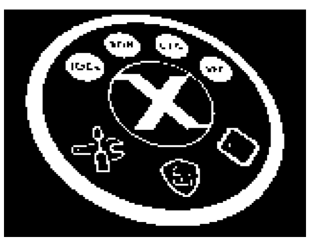
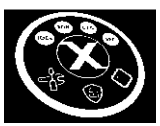


Not useful if no grayscale information should be lost

Performance optimization

Use mipmaps (image pyramids)

- *Example: threshold() (Optimize most (-O3))*
- **160x120: ~4570 µs**
- **80x60: ~1150 µs**
- **40x30: ~290 µs**
- **20x15: ~80 µs**

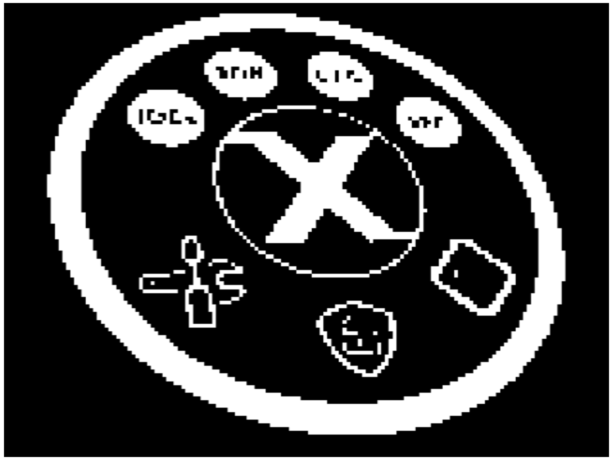
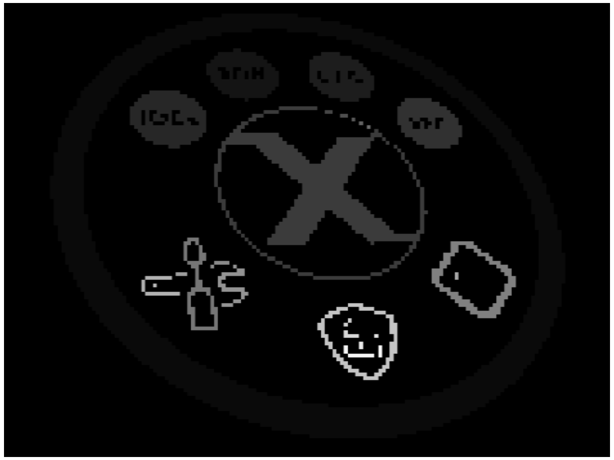

Very useful for finding (the location of) binary objects

Performance optimization

Use mipmaps (image pyramids)

- *Example:* **160x120:   ~16000 µs**

threshold()   *labelTwoPass()*

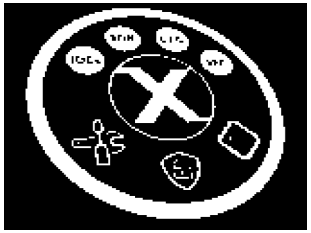




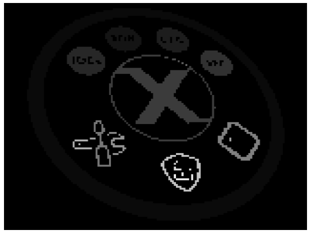
Performance optimization

Use mipmaps (image pyramids)

- *Example:* **80x60: ~4200 µs**

threshold() *labelTwoPass()*



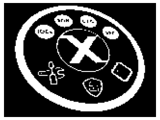


Performance optimization

Use mipmaps (image pyramids)

- *Example:* **40x30:   ~1010 µs**

threshold()   *labelTwoPass()*

Performance optimization

Use mipmaps (image pyramids)

- *Example:* **20x15: ~280 µs**

threshold() *labelTwoPass()*



An object located at (10,10)
is approximately at (80,80)
in the original format

Performance optimization

Use mipmaps (image pyramids)

- Use the *zoomFactor()* function for creating a pyramid

```
// ----------------------------------------------------------------------------
// Local image memory allocation
// ----------------------------------------------------------------------------
image_t *src = newUint8Image(EVDK5_WIDTH, EVDK5_HEIGHT);
image_t *dst = newUint8Image(EVDK5_WIDTH, EVDK5_HEIGHT);

const uint32_t zoom_factor = 2;
image_t *tmp = newUint8Image(EVDK5_WIDTH/zoom_factor, EVDK5_HEIGHT/zoom_factor);
```

Performance optimization

Use mipmaps (image pyramids)

- Use the *zoomFactor()* function for creating a pyramid

```
while(1U)
{
    // --------------------------------------------------------------------
    // Wait for camera image complete
    // --------------------------------------------------------------------
    while(smartdma_camera_image_complete == 0)
    {}

    smartdma_camera_image_complete = 0;

    // --------------------------------------------------------------------
    // Image processing pipeline
    // --------------------------------------------------------------------
    // Convert uyvy_pixel_t camera image to uint8_pixel_t image
    convertToUint8(cam, src);
```

Performance optimization

Use mipmaps (image pyramids)

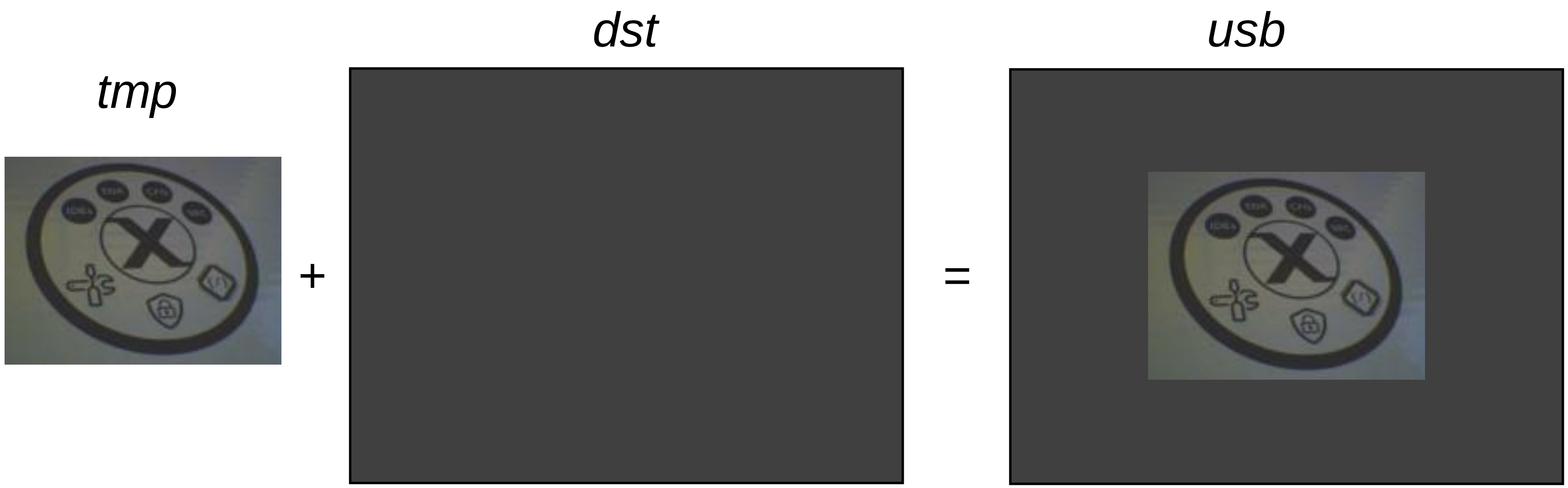
- Use the *zoomFactor()* function for creating a pyramid

```
// Zoom the image
zoomFactor(src, tmp, 0, 0, src->cols, src->rows, ZOOM_OUT, zoom_factor);

// Process the zoomed image
threshold(tmp, tmp, 0, 64);
labelTwoPass(tmp, tmp, CONNECTED_EIGHT, 64);
scale(tmp, tmp);
```

Performance optimization

Use mipmaps (image pyramids)

- If needed, show the zoomed image by copying it to a QQVGA image (which is the size of the usb image)
- Add a border for nice visual effect



Performance optimization

Use mipmaps (image pyramids)

- If needed, show the zoomed image by copying it to a QQVGA image (which is the size of the usb image)
- Add a border for nice visual effect

```
// Show zoomed image
// Only one resolution can be transferred via USB, so:
// 1. Set destination image to gray background (64) for nicer visual effect
// 2. Copy smaller image to the center of the larger image for
//    nicer visual effect
memset(dst->data, 64, dst->cols * dst->rows);
int offset_c = (dst->cols / 2) - (tmp->cols / 2);
int offset_r = (dst->rows / 2) - (tmp->rows / 2);
for(int r=0; r < tmp->rows; r++)
{
    memcpy((dst->data)+((r+offset_r) * dst->cols) + offset_c, (tmp->data) +
        (r * tmp->cols), tmp->cols);
}
```

Performance optimization

Use mipmaps (image pyramids)

- Use the *zoomFactor()* function for creating a pyramid

```
    // Convert uint8_pixel_t image to bgr888_pixel_t image for USB
    convertToBgr888(dst, usb);

    // ----------------------------------------------------------------------
    // Set flag for USB interface that a new frame is available
    // ----------------------------------------------------------------------
    image_available_for_usb = 1;
}
```

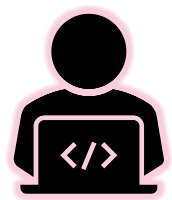
References

- Shore, C. (2010) Efficient C Code for ARM Devices, ARM, Downloaded November 2022 from https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-01-26-13/ATC_2D00_152_5F00_paper_5F00_Shore.pdf

- Mukherjee, S. () Efficient C Code for ARM Devices, ARM, Downloaded November 2022 from https://web.archive.org/web/20170829213827/https://www.arm.com/files/pdf/AT_-_Better_C_Code_for_ARM_Devices.pdf

- Wikipedia contributors. (2024, August 24). Mipmap. In Wikipedia, The Free Encyclopedia. Retrieved 18:58, October 5, 2024, from https://en.wikipedia.org/w/index.php?title=Mipmap&oldid=1242024792

EVD1 – Assignment



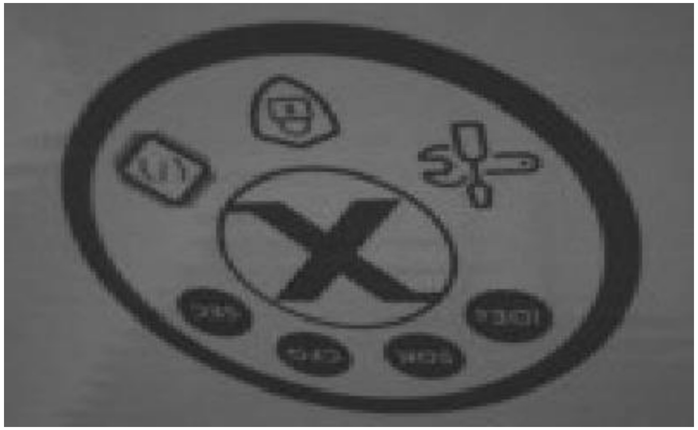
Study guide

Week 2

1 Image fundamentals – scaleFast()

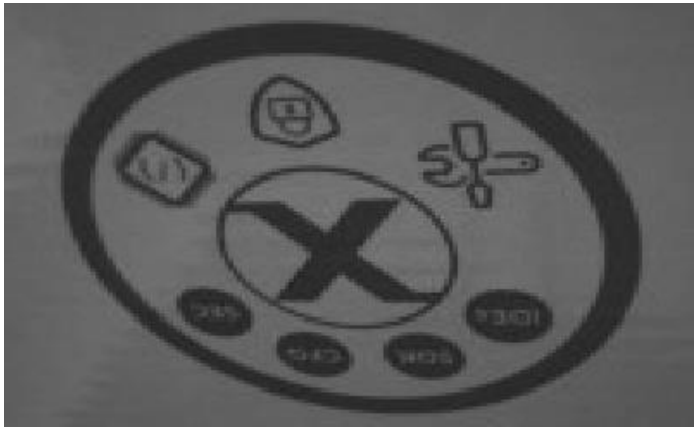
Rotate 180

- The camera is mounted upside down
- We need an ultra-fast implementation for rotating the image 180 degrees



Rotate 180

- The following methods will be compared for performance
  1. Using gonio functions
  2. Flipping the image in both horizontal and vertical direction
     a) C
     b) ARM inline assembly
     c) ARM with C idiom
     d) ARM 32-bit architecture optimized assembly



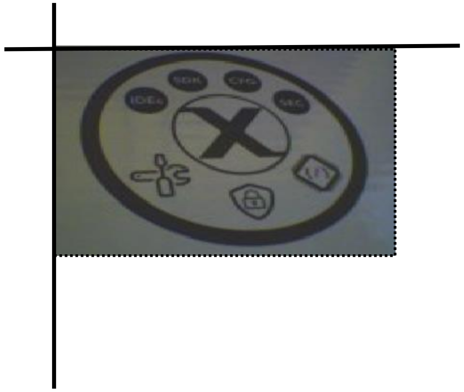
Rotate 180 - gonio

- Rotation is defined as:

$$x' = x_c + (x - x_c) \cdot cos\theta - (y - y_c) \cdot sin\theta$$
$$y' = y_c + (x - x_c) \cdot sin\theta + (y - y_c) \cdot cos\theta$$

where

(x_c, y_c): the rotation origin

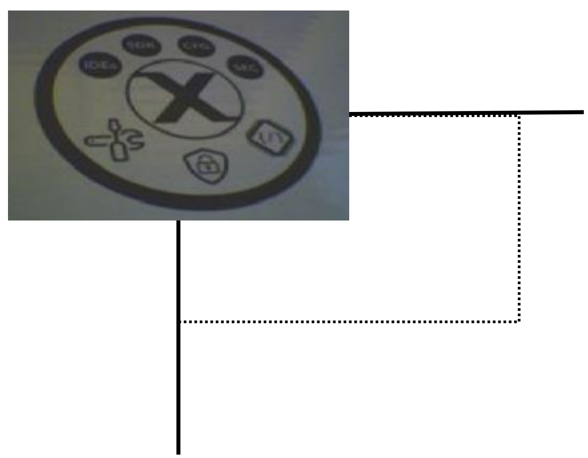


Rotate 180 - gonio

- Rotation is defined as:

$$x' = x_c + (x - x_c) \cdot cos\theta - (y - y_c) \cdot sin\theta$$

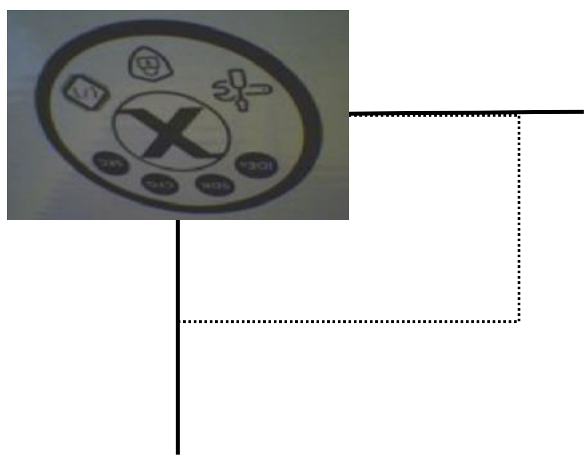
$$y' = y_c + \underbrace{(x - x_c)}_{\text{Translate}} \cdot sin\theta + \underbrace{(y - y_c)}_{\text{Translate}} \cdot cos\theta$$



Rotate 180 - gonio

- Rotation is defined as:

$$x' = x_c + (x - x_c) \cdot cos\theta - (y - y_c) \cdot sin\theta$$
$$y' = y_c + \underbrace{(x - x_c) \cdot sin\theta}_{\text{Rotate}} + \underbrace{(y - y_c) \cdot cos\theta}_{\text{Rotate}}$$

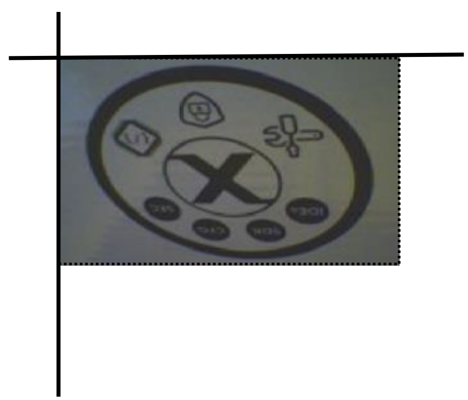


Rotate 180 - gonio

- Rotation is defined as:

$$x' = x_c + (x - x_c) \cdot cos\theta - (y - y_c) \cdot sin\theta$$

$$y' = \underbrace{y_c}_{\text{Translate}} + (x - x_c) \cdot sin\theta + (y - y_c) \cdot cos\theta$$



Rotate 180 - gonio

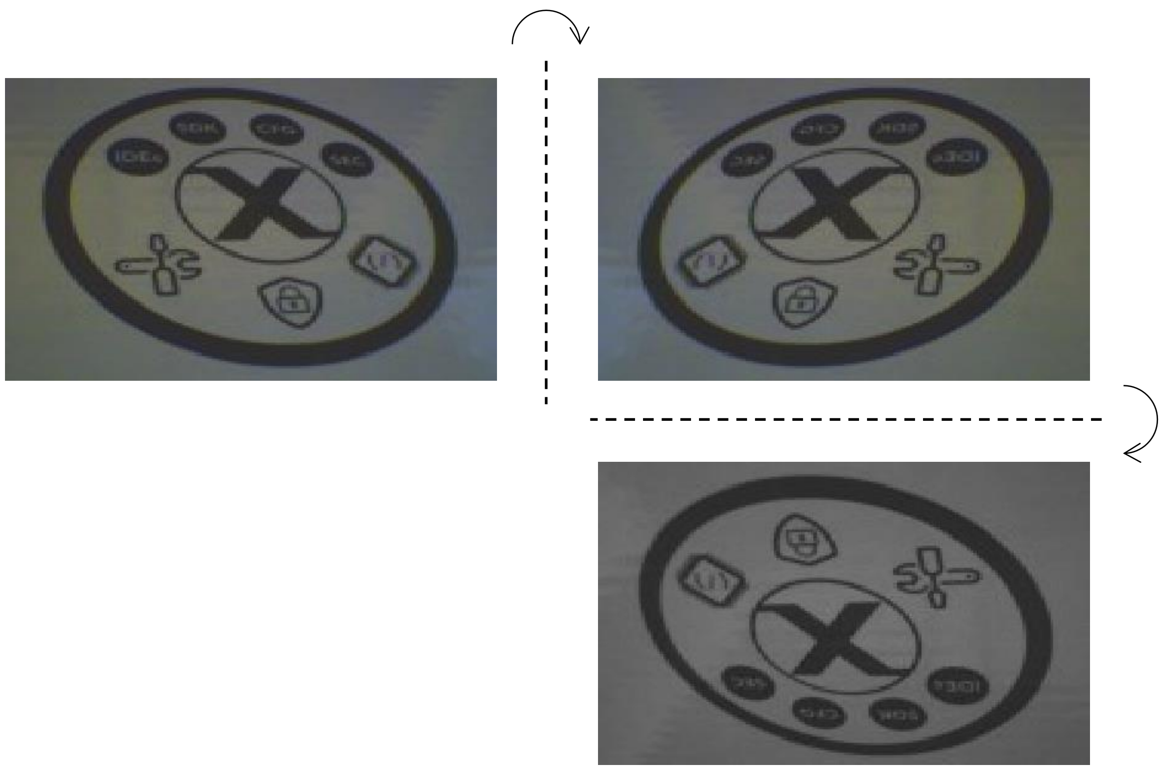
void **rotate**(const image_t ***src**, image_t ***dst**, const float **radians**, const point_t **center**);

See file **EVDK_Operators\graphics_algorithms.c**

Implementation	Execution time (QQVGA and optimize most (-O3)):
Gonio	11330 us
Flip in C	
Flip in ARM inline assembly	
Flip in ARM with C idiom	
Flip in ARM architecture optimized assembly	

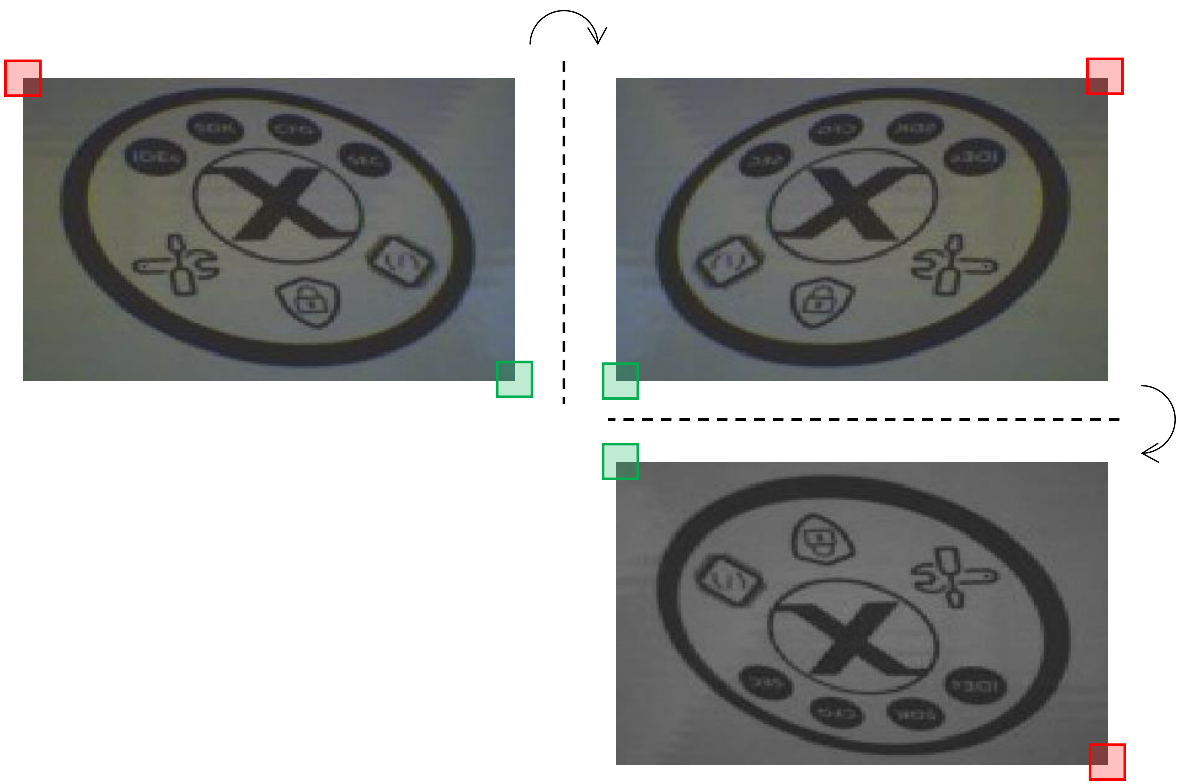
Rotate 180 – Flip in C

- Flip the image in both directions



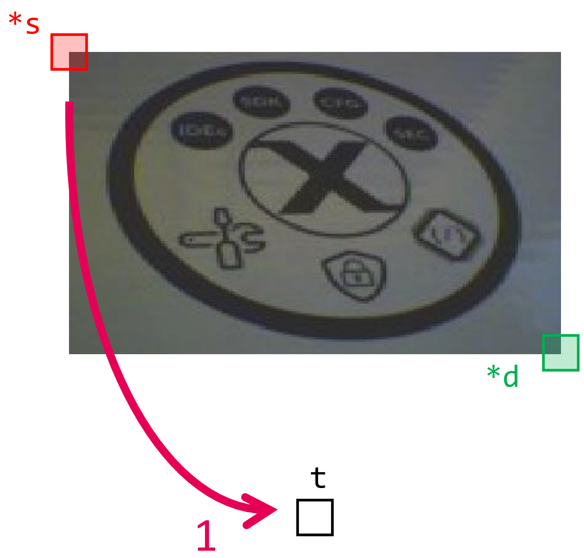
Rotate 180 – Flip in C

- Flip the image in both directions



Rotate 180 – Flip in C

- Flip the image in both directions

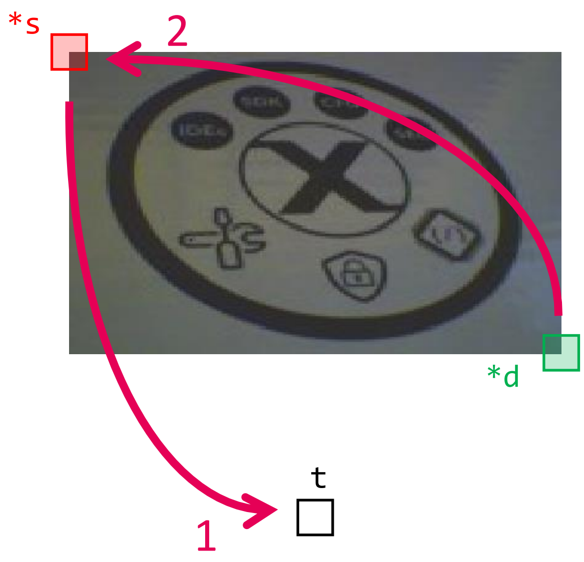



```c
uint32_t i;
uint8_pixel_t *s = (uint8_pixel_t *)img->data;
uint8_pixel_t *d = (uint8_pixel_t *)(img->data +
    (img->rows * img->cols * sizeof(uint8_pixel_t)) -
    (1 * sizeof(uint8_pixel_t)));

uint8_pixel_t t;

for(i = (img->rows * img->cols) / 2; i > 0; i--)
{
    t    = *s; // 1


}
```

- Uses two pointers to iterate (half) the image

Rotate 180 – Flip in C

- Flip the image in both directions



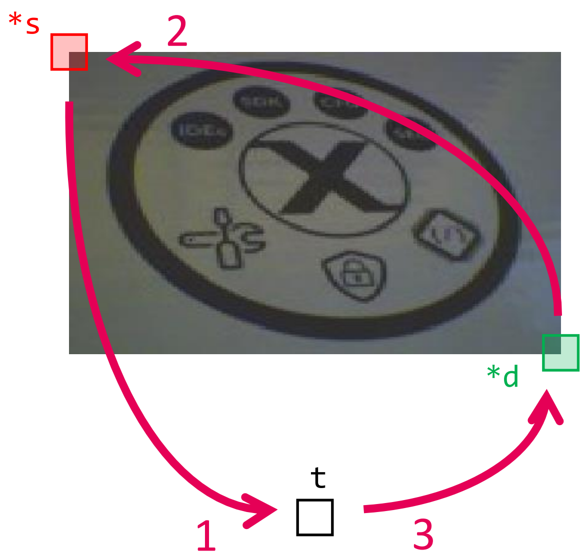

```c
uint32_t i;
uint8_pixel_t *s = (uint8_pixel_t *)img->data;
uint8_pixel_t *d = (uint8_pixel_t *)(img->data +
    (img->rows * img->cols * sizeof(uint8_pixel_t)) -
    (1 * sizeof(uint8_pixel_t)));

uint8_pixel_t t;

for(i = (img->rows * img->cols) / 2; i > 0; i--)
{
    t    = *s; // 1
    *s++ = *d; // 2

}
```

- Uses two pointers to iterate (half) the image

HAN_UNIVERSITY OF APPLIED SCIENCES

Rotate 180 – Flip in C

- Flip the image in both directions




```c
uint32_t i;
uint8_pixel_t *s = (uint8_pixel_t *)img->data;
uint8_pixel_t *d = (uint8_pixel_t *)(img->data +
    (img->rows * img->cols * sizeof(uint8_pixel_t)) -
    (1 * sizeof(uint8_pixel_t)));

uint8_pixel_t t;

for(i = (img->rows * img->cols) / 2; i > 0; i--)
{
    t    = *s; // 1
    *s++ = *d; // 2
    *d-- = t;  // 3
}
```

- Uses two pointers to iterate (half) the image

HAN_UNIVERSITY OF APPLIED SCIENCES

Rotate 180 – Flip in C

void **rotate180_c**(const image_t ***img**);

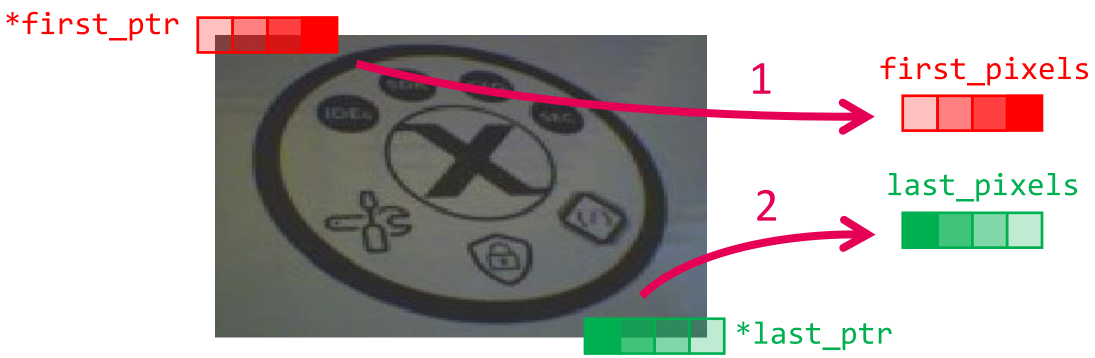
See file **EVDK_Operators\graphics_algorithms.c**

Implementation	Execution time (QQVGA and optimize most (-O3)):
Gonio	11330 us
Flip in C	490 us
Flip in ARM inline assembly	
Flip in ARM with C idiom	
Flip in ARM architecture optimized assembly	

Rotate 180 – Flip in ARM inline assembly


```c
// Pointer to the first four pixels
register uint32_t *first_ptr = (uint32_t *)img->data;

// Pointer to the end of the data
register uint32_t *last_ptr = (uint32_t *)(img->data + (img->rows * img->cols * sizeof(uint8_pixel_t)));

// Temporary variables
register uint32_t  first_pixels, last_pixels;
```

Rotate 180 – Flip in ARM inline assembly



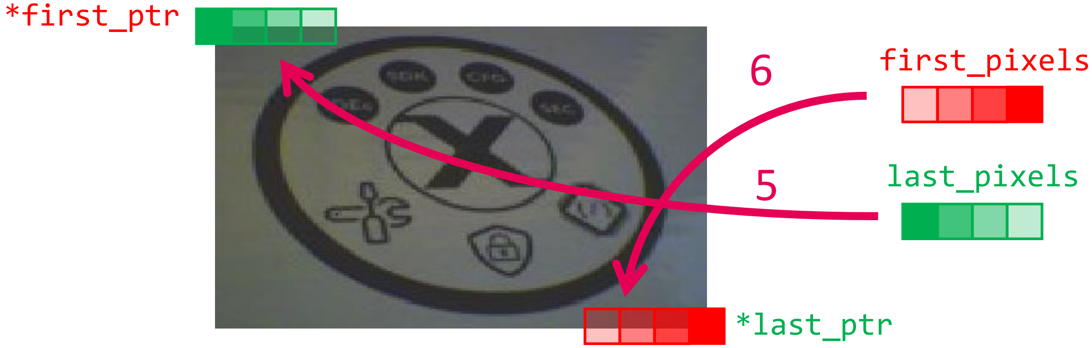

```
while(first_ptr != last_ptr)
{
    // Read pixels
    first_pixels = *first_ptr; // 1
    last_pixels = *(--last_ptr); // 2
```

Rotate 180 – Flip in ARM inline assembly


```
// Reverse 32-bit byte order : b3 b2 b1 b0 -> b0 b1 b2 b3
__asm__ ("REV %[result], %[value]" : [result] "=r" (first_pixels) : [value] "r" (first_pixels));// 3
__asm__ ("REV %[result], %[value]" : [result] "=r" (last_pixels)  : [value] "r" (last_pixels)); // 4
```

https://developer.arm.com/documentation/100235/0100/The-Cortex-M33-Instruction-Set/Cortex-M33-instructions?lang=en

Rotate 180 – Flip in ARM inline assembly




```
    *(first_ptr++) = last_pixels;  // 5
    *last_ptr      = first_pixels; // 6
}
```

Rotate 180 – Flip in ARM inline assembly

void **rotate180_arm**(const image_t ***img**);

See file **EVDK_Operators\graphics_algorithms.c**

Implementation	Execution time (QQVGA and optimize most (-O3)):
Gonio	11330 us
Flip in C	490 us
Flip in ARM inline assembly	160 us
Flip in ARM with C idiom	
Flip in ARM architecture optimized assembly	

Rotate 180 – Flip in ARM with C idiom

Some compilers at a specific optimization level recognise this pattern in C! This is called **Idiom recognition.**

```
// Reverse 32-bit byte order : b3 b2 b1 b0 -> b0 b1 b2 b3
first_pixels = ((first_pixels >> 24) & 0x000000FF) |
               ((first_pixels >>  8) & 0x0000FF00) |
               ((first_pixels <<  8) & 0x00FF0000) |
               ((first_pixels << 24) & 0xFF000000);

last_pixels = ((last_pixels >> 24) & 0x000000FF) |
              ((last_pixels >>  8) & 0x0000FF00) |
              ((last_pixels <<  8) & 0x00FF0000) |
              ((last_pixels << 24) & 0xFF000000);
```

```
// Reverse 32-bit byte order : b3 b2 b1 b0 -> b0 b1 b2 b3
__asm__ ("REV %[result], %[value]" : [result] "=r" (first_pixels) : [value] "r" (first_pixels));
__asm__ ("REV %[result], %[value]" : [result] "=r" (last_pixels)  : [value] "r" (last_pixels));
```

Rotate 180 – Flip in ARM with C idiom

void **rotate180_arm**(const image_t ***img**);

See file **EVDK_Operators\graphics_algorithms.c**

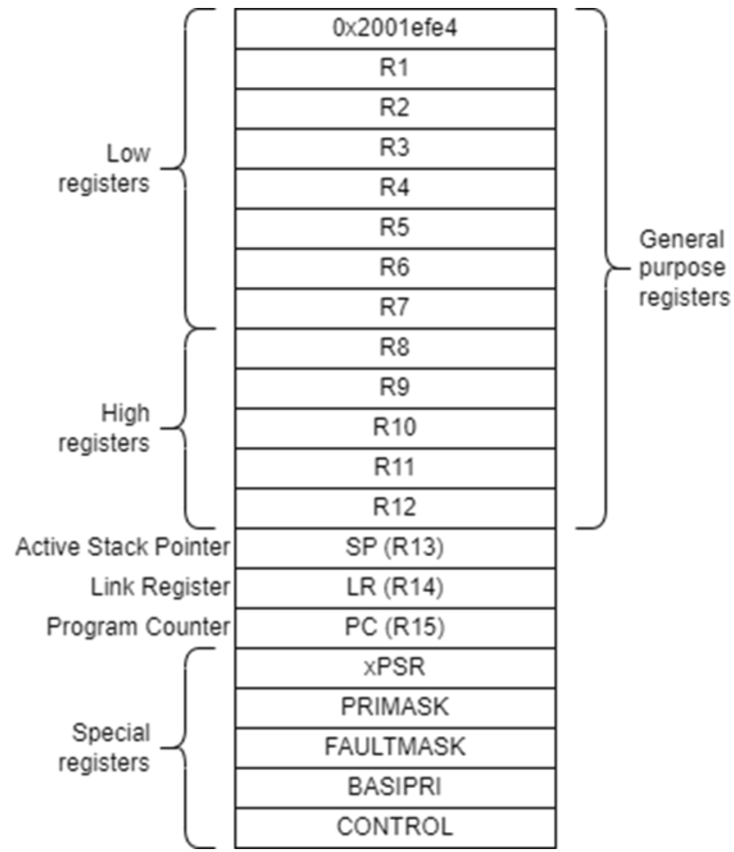
Implementation	Execution time (QQVGA and optimize most (-O3)):
Gonio	11330 us
Flip in C	490 us
Flip in ARM inline assembly	160 us
Flip in ARM with C idiom	160 us
Flip in ARM architecture optimized assembly	

Rotate 180 – Flip in ARM architecture optimized assembly

- Minimize the number of load and store operations
- Minimize the number of transfers between core registers

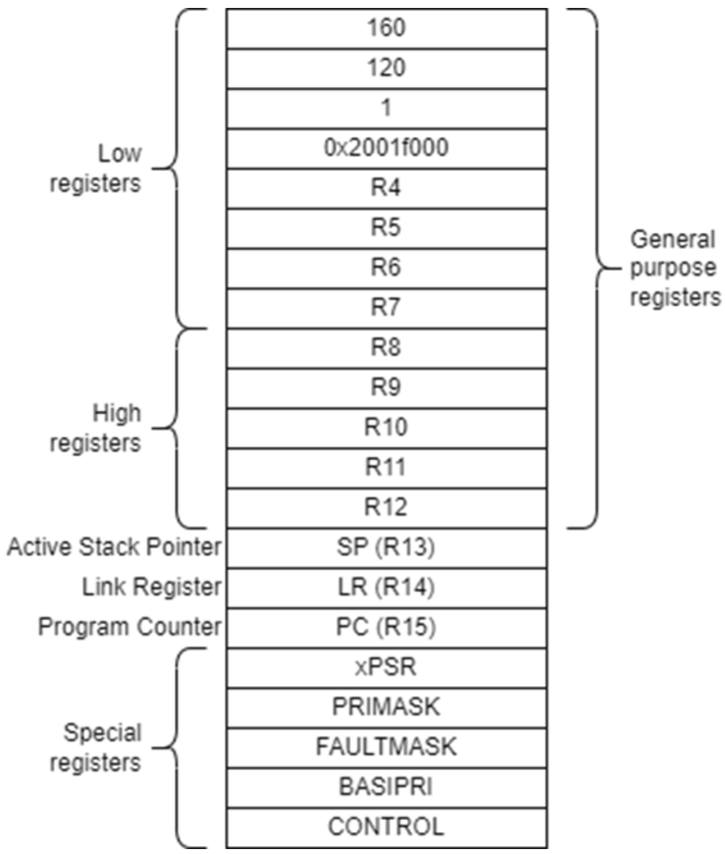
```
// Note: For using this function declare the
// following function prototype external in the
// project:
//
// extern void rotate180_cm33(const image_t *img);
//
// Example usage:
//
// rotate180_cm33(img);
```

- Core register r0 holds the pointer to the image, because there is one argument



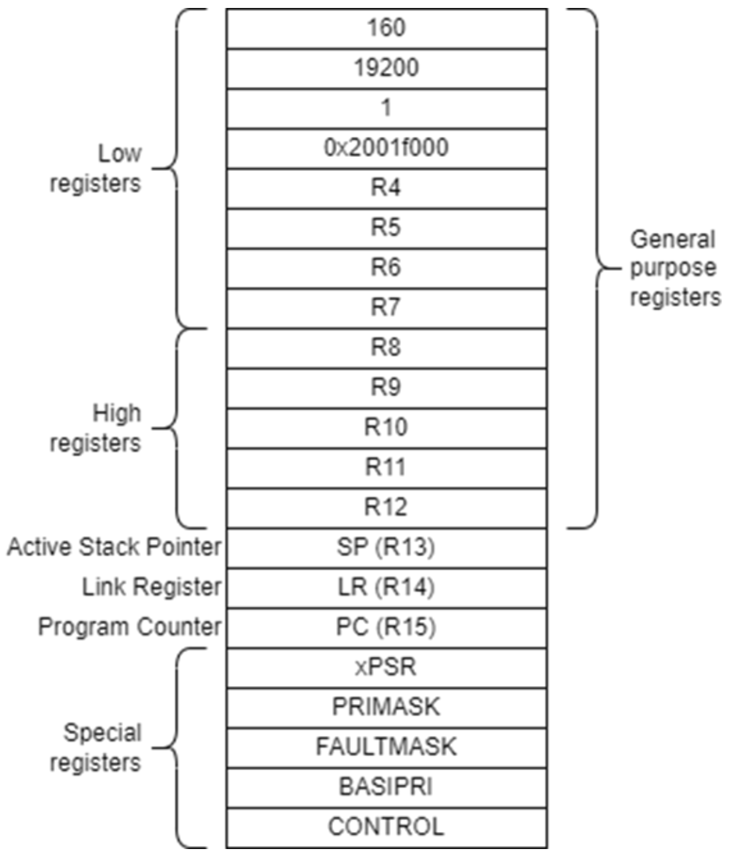
Rotate 180 – Flip in ARM architecture optimized assembly

```
rotate180_cm33:

        PUSH {r4-r10}

        // Load four words from image pointer
        // r0 = image_t.cols
        // r1 = image_t.rows
        // r2 = image_t.type
        // r3 = image_t.data
        LDMIA r0, {r0-r3}
```



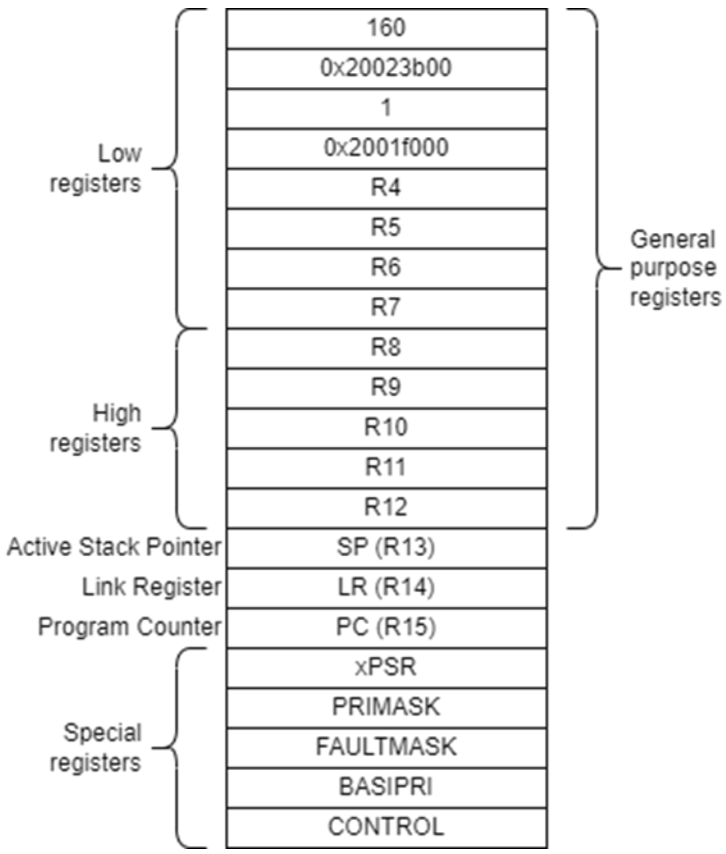
Rotate 180 – Flip in ARM architecture optimized assembly

```
rotate180_cm33:

        PUSH {r4-r10}

        // Load four words from image pointer
        // r0 = image_t.cols
        // r1 = image_t.rows
        // r2 = image_t.type
        // r3 = image_t.data
        LDMIA r0, {r0-r3}

        // Image size = cols x rows
        MUL r1, r0, r1
```



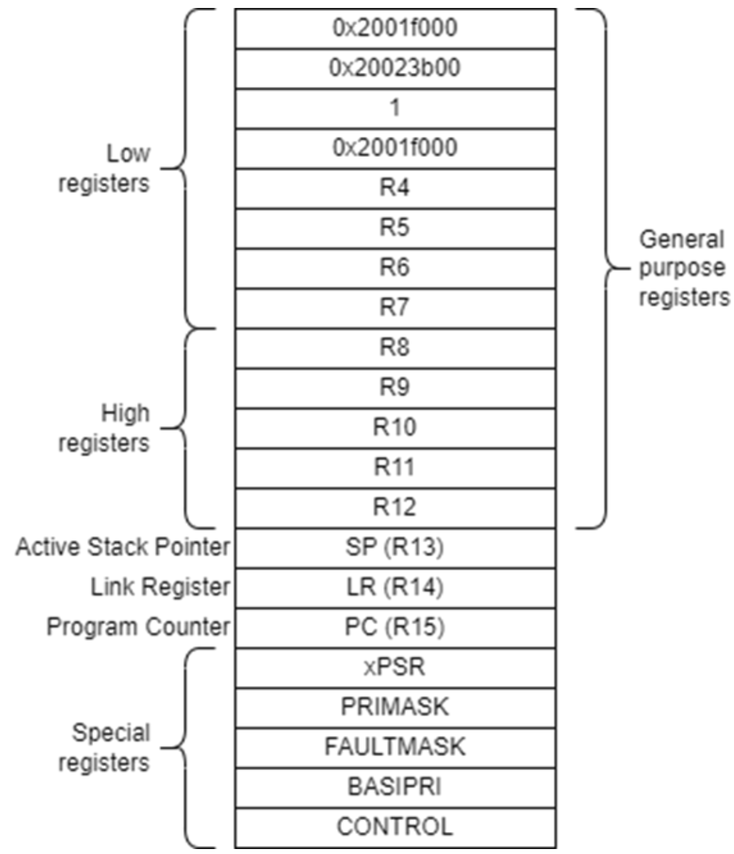
Rotate 180 – Flip in ARM architecture optimized assembly

```
rotate180_cm33:

        PUSH {r4-r10}

        // Load four words from image pointer
        // r0 = image_t.cols
        // r1 = image_t.rows
        // r2 = image_t.type
        // r3 = image_t.data
        LDMIA r0, {r0-r3}

        // Image size = cols x rows
        MUL r1, r0, r1

        // Backward pointer = size + data pointer
        ADD r1, r1, r3
```



Rotate 180 – Flip in ARM architecture optimized assembly

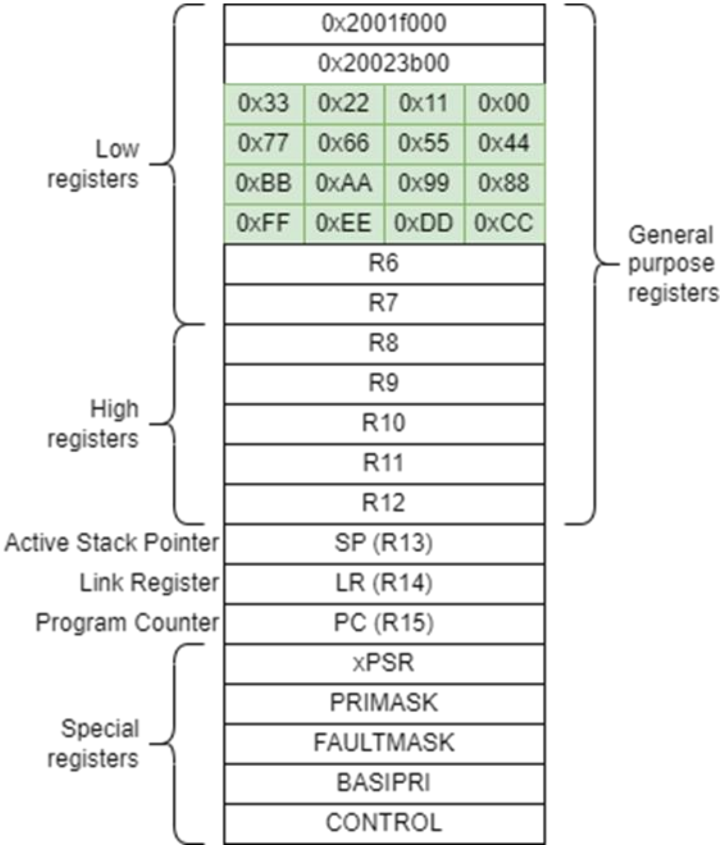
```
rotate180_cm33:

        PUSH {r4-r10}

        // Load four words from image pointer
        // r0 = image_t.cols
        // r1 = image_t.rows
        // r2 = image_t.type
        // r3 = image_t.data
        LDMIA r0, {r0-r3}

        // Image size = cols x rows
        MUL r1, r0, r1

        // Backward pointer = size + data pointer
        ADD r1, r1, r3

        // Forward pointer = data pointer
        MOV r0, r3
```



HAN_UNIVERSITY OF APPLIED SCIENCES

Rotate 180 – Flip in ARM architecture optimized assembly

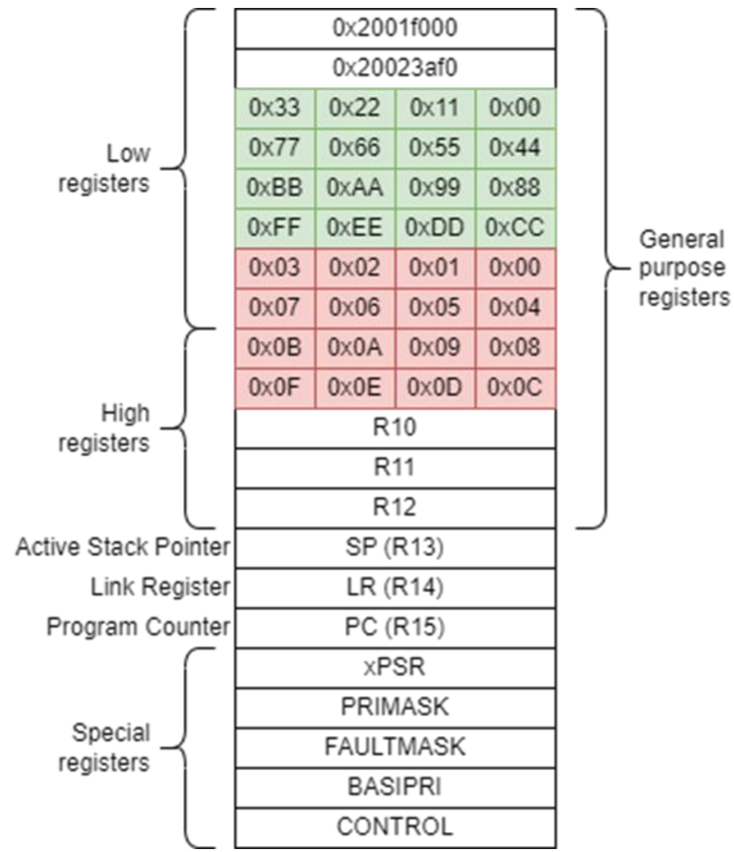
```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}
```



HAN_UNIVERSITY OF APPLIED SCIENCES

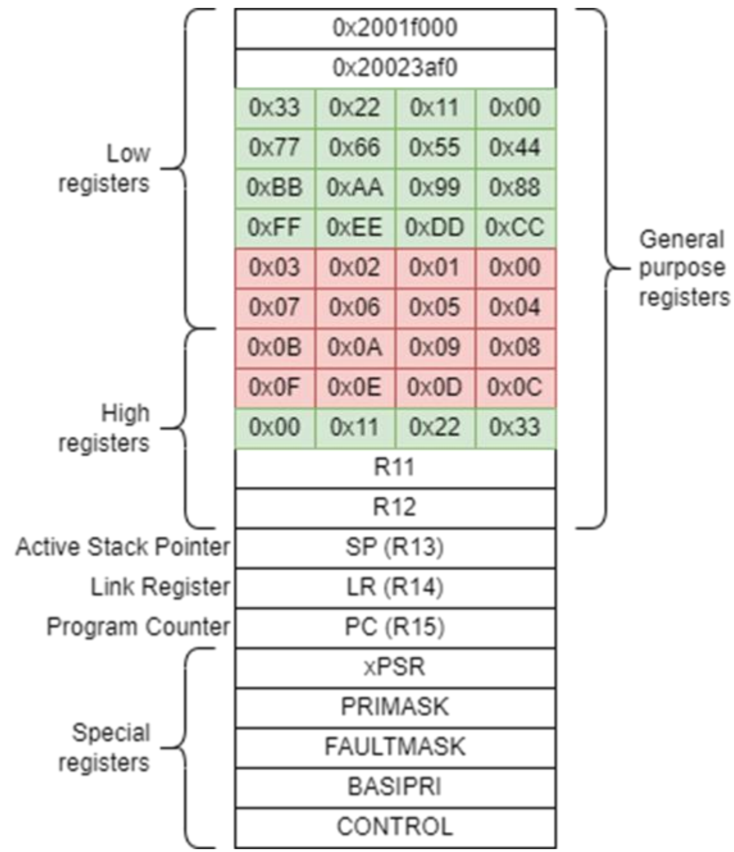
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}
```



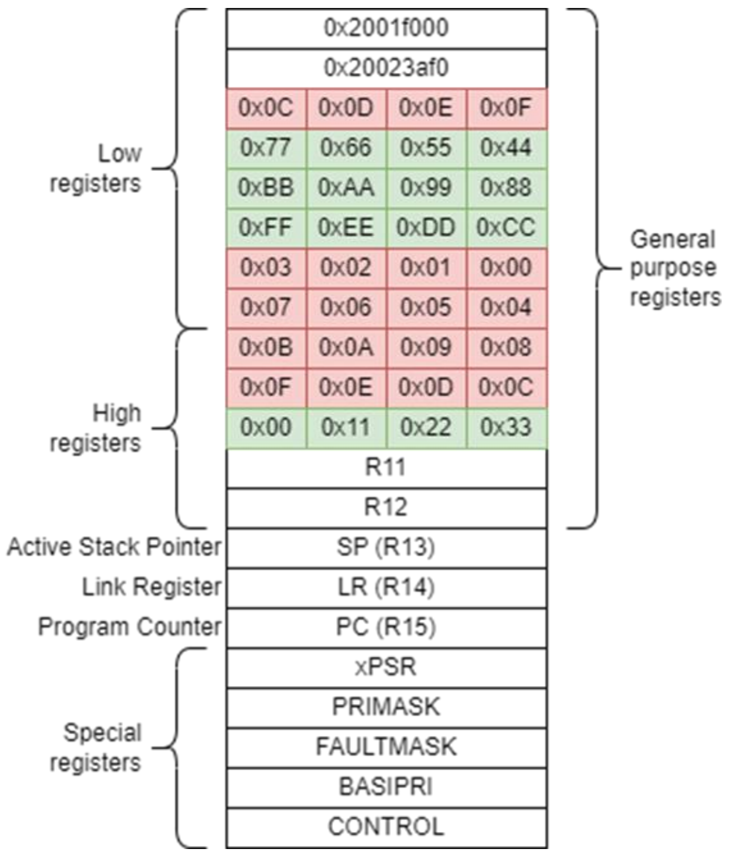
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}

        // Reverse the bytes in each word, as well
        // as the words themselves
        REV r10, r2
```



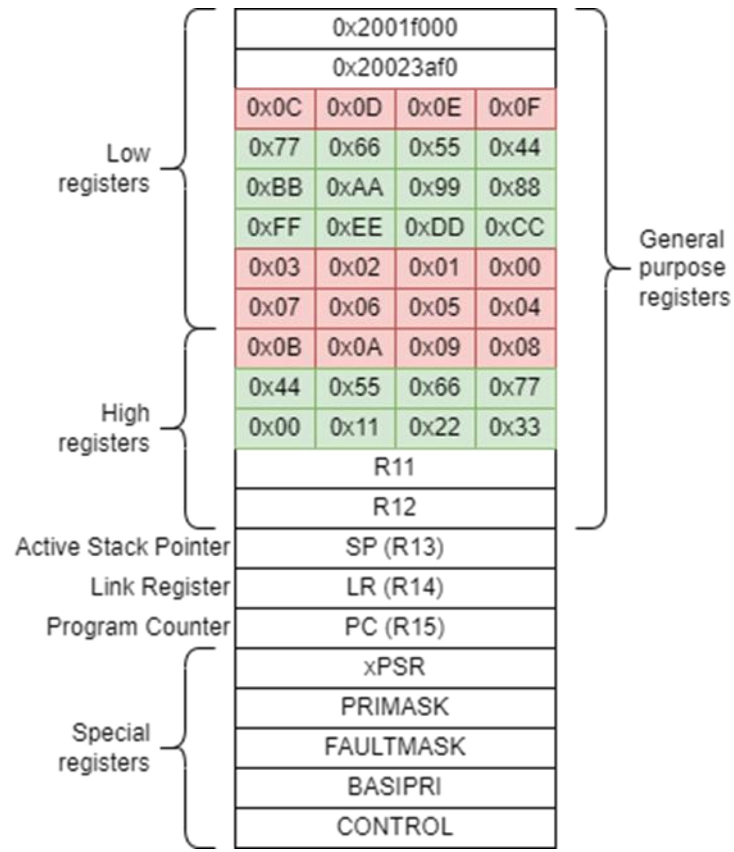
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}

        // Reverse the bytes in each word, as well
        // as the words themselves
        REV r10, r2
        REV r2,  r9
```



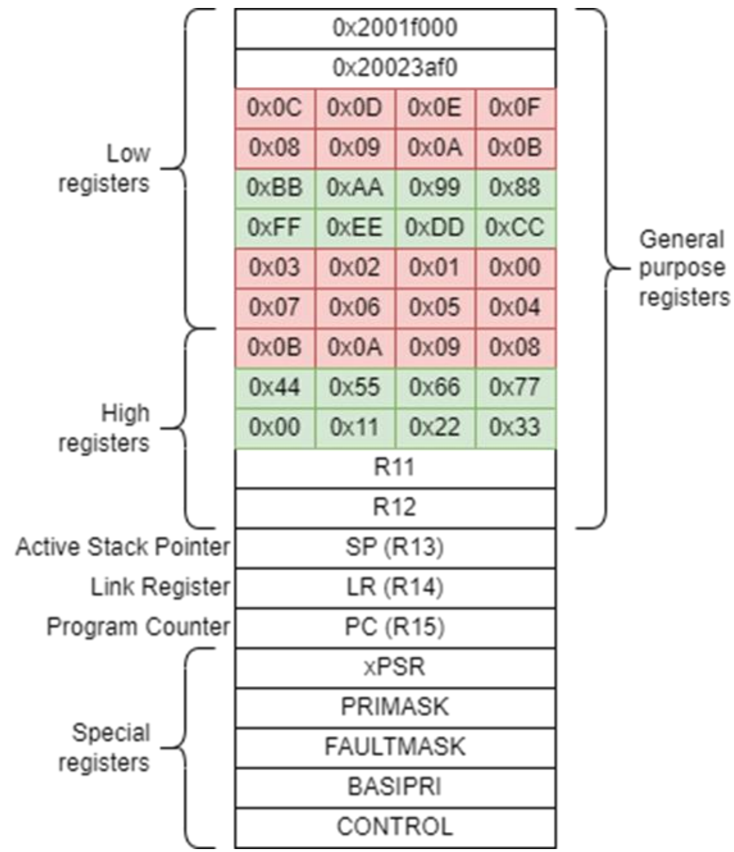
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}

        // Reverse the bytes in each word, as well
        // as the words themselves
        REV r10, r2
        REV r2,  r9
        REV r9,  r3
```



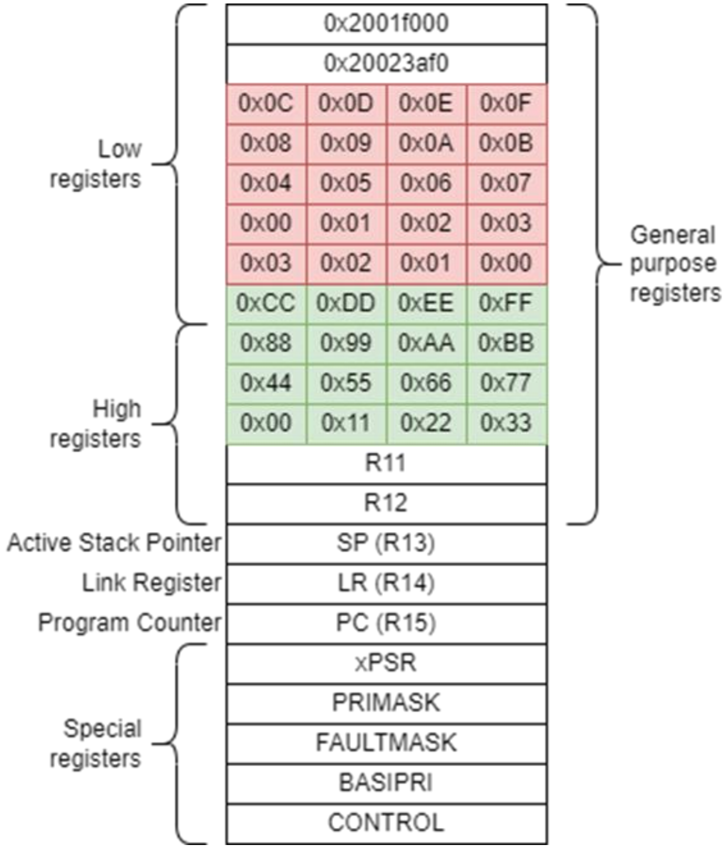
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}

        // Reverse the bytes in each word, as well
        // as the words themselves
        REV r10, r2
        REV r2,  r9
        REV r9,  r3
        REV r3,  r8
```



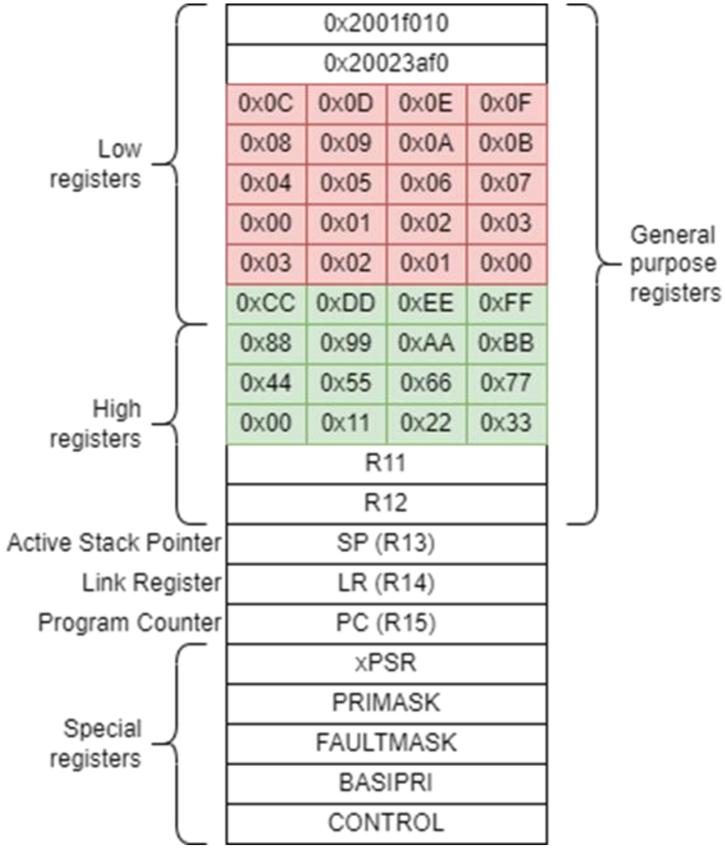
Rotate 180 – Flip in ARM architecture optimized assembly

```
rev_loop:
        // Load four words from forward pointer
        // and increment address afterwards
        LDMIA r0, {r2-r5}

        // Load four words from backward pointer,
        // decrement address before and update
        LDMDB r1!, {r6-r9}

        // Reverse the bytes in each word, as well
        // as the words themselves
        REV r10, r2
        REV r2,  r9
        REV r9,  r3
        REV r3,  r8
        REV r8,  r4
        REV r4,  r7
        REV r7,  r5
        REV r5,  r6
```



Rotate 180 – Flip in ARM architecture optimized assembly

```
// Write transformed words to forward
// pointer, increment address afterwards
// and update
STMIA r0!, {r2-r5}

// Write transformed words back to
// backward pointer and increment address
STMIA r1, {r7-r10}

// Repeat until forward and backward
// pointer meet
CMP r0, r1
BNE rev_loop

POP {r4-r10}
BX lr
```



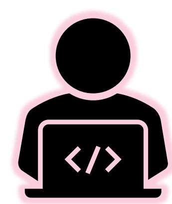
Rotate 180 – Flip in ARM architecture optimized assembly

void **rotate180_cm33**(const image_t ***img**);

See file **source\rotate180_cm33.s** (MCUXpresso-IDE only)

Implementation	**Execution time (QQVGA and optimize most (-O3)):**
Gonio	11330 us
Flip in C	490 us
Flip in ARM inline assembly	160 us
Flip in ARM with C idiom	160 us
Flip in ARM architecture optimized assembly	120 us

EVD1 – Assignment



Study guide

Week 2

2 Image fundamentals – clearuint8image_cm33()

3 EXTRA Image fundamentals – convertuyvytouint8_cm33()

Graphics Algorithms

- Manipulate the position of pixels in an image
- Used to create images by using algorithms

- Affine transformation
- Warp

Affine transformation

- What is an affine transformation?

> *"An affine transformation is a function between affine spaces which preserves points, straight lines and planes"*

- An affine transformation changes a pixel location. It does not affect its value.
- Characteristics
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved

Affine transformation

- Translation example:

$$x' = x + c \text{ (where } c = 50)$$
$$y' = y + f \text{ (where } f = 50)$$

Affine transformation

- Scaling example:

$$x' = ax \text{ (where a = 2)}$$
$$y' = ey \text{ (where } e = 2\text{)}$$

Affine transformation

- Conclusion: there is a function T that transforms source coordinates to destination coordinates:

$$(x', y') = T \cdot (x, y)$$

- In a matrix notation:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = T \begin{vmatrix} x \\ y \end{vmatrix}$$

- Scaling example:

$$\begin{vmatrix} ax \\ ey \end{vmatrix} = T \begin{vmatrix} x \\ y \end{vmatrix}$$

- Solving T :

$$\begin{vmatrix} ax \\ ey \end{vmatrix} = \begin{vmatrix} \boldsymbol{a} & 0 \\ 0 & \boldsymbol{e} \end{vmatrix} \begin{vmatrix} x \\ y \end{vmatrix}$$

Affine transformation

- Translation example:

$$\begin{vmatrix} x + c \\ y + f \end{vmatrix} = T \begin{vmatrix} x \\ y \end{vmatrix}$$

- However, there is no solution for T…

- To solve T, we can use homogeneous coordinates instead of Cartesian coordinates!

Affine transformation

- The relation between Cartesian coordinates (x, y) and homogeneous coordinates (X, Y, Z) is defined as:

$$x = \frac{X}{Z} \; and \; y = \frac{Y}{Z} \; with \; Z \neq 0$$

- When $Z = 1$, we get:

$$x = X \; and \; y = Y$$

- In a matrix notation when $Z = 1$:

$$\begin{vmatrix} X' \\ Y' \\ Z' \end{vmatrix} = T \begin{vmatrix} X \\ Y \\ Z \end{vmatrix} \equiv \begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = T \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Affine transformation

- For translation this yields:

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} x+c \\ y+f \\ 1 \end{vmatrix} = T \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Now, solving T is possible:

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} x+c \\ y+f \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & \boldsymbol{c} \\ 0 & 1 & \boldsymbol{f} \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Rewritten:

$$x' = 1x + 0y + 1c = x + c \quad \rightarrow c: \text{"}translation\ over\ x\text{"}$$
$$y' = 0x + 1y + 1f = y + f \quad \rightarrow f: \text{"}translation\ over\ y\text{"}$$

Affine transformation

- Now back to scaling.
  How can we also define scaling with homogeneous coordinates?

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} \boldsymbol{a} & 0 & 0 \\ 0 & \boldsymbol{e} & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Rewritten:

$$x' = ax + 0y + 0 = ax$$
$$y' = 0x + ey + 0 = ey$$

Affine transformation

- What does the following transformation matrix accomplish?

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & \boldsymbol{b} & 0 \\ \boldsymbol{d} & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Rewritten:

$$x' = x + by$$
$$y' = dx + y$$

- dst x is based on src x plus a factor b times y
- dst y is based on src y plus a factor d times x

- This operation is known as shearing

Affine transformation

- Conclusion: a transformation matrix is used to calculate the pixel location (x', y') in the destination image from the original pixel location (x, y) in the source image
- The general form of an affine transformation matrix is defined as:

$$T = \begin{vmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{vmatrix}$$

Affine transformation - examples

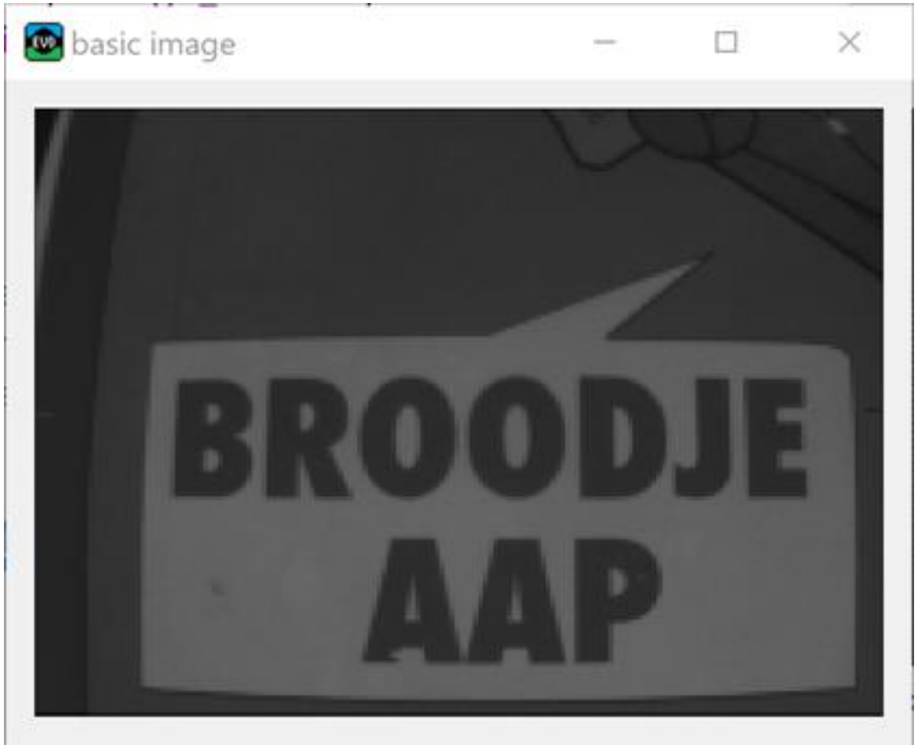
- Examples: identity

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Rewritten:

$x' = x + 0y + 0 = x$

$y' = 0x + y + 0 = y$



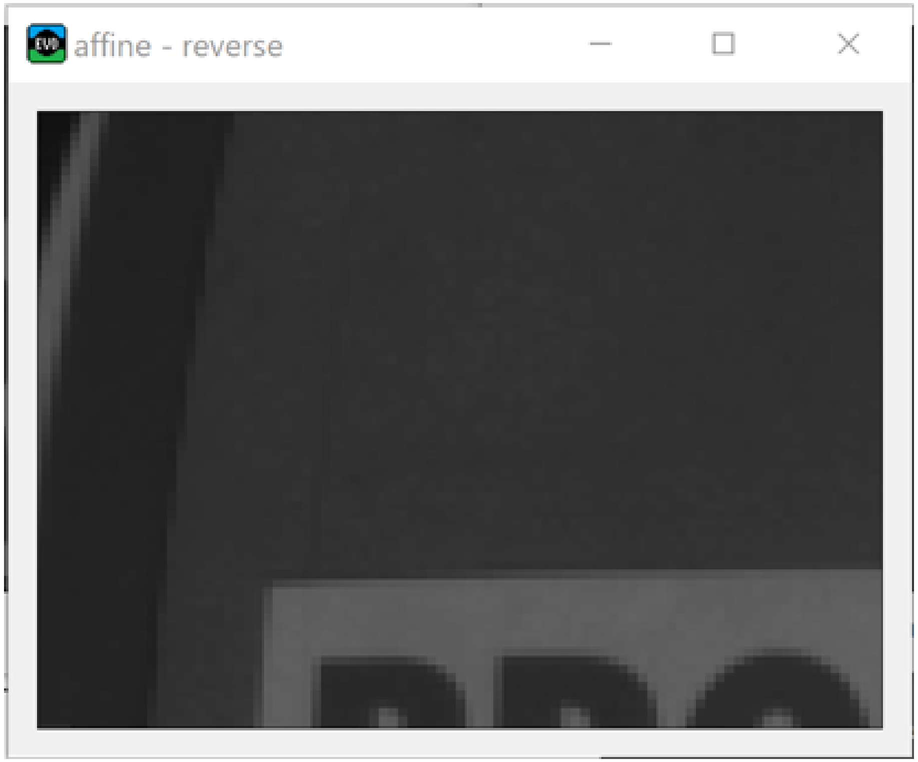
Affine transformation - examples

- Examples: scale

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

where
$a = 2$
$e = 2$

Rewritten:
$x' = 2x + 0y + 0 = 2x$
$y' = 0x + 2y + 0 = 2y$



HAN_UNIVERSITY OF APPLIED SCIENCES

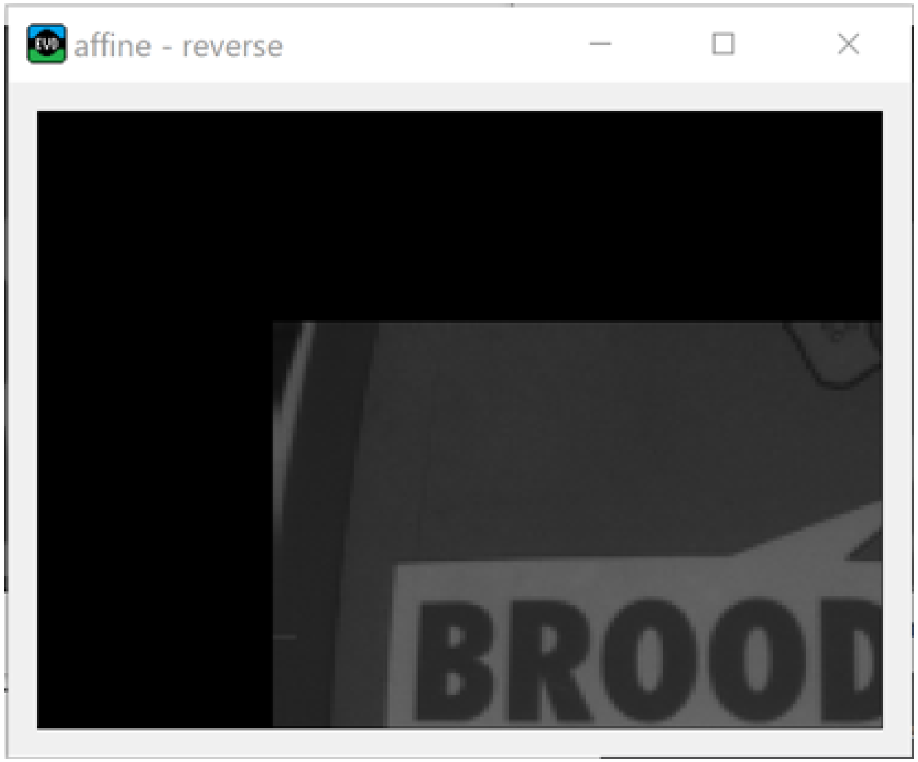
Affine transformation - examples

- Examples: Translation

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & c \\ 0 & 1 & f \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

where
$c = 50$
$f = 50$

Rewritten:
$x' = x + 0y + 50 = x + 50$
$y' = 0x + y + 50 = y + 50$



HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation - examples

- Examples: Shear

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & b & 0 \\ d & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

where
$b = 0.50$
$d = 0.25$

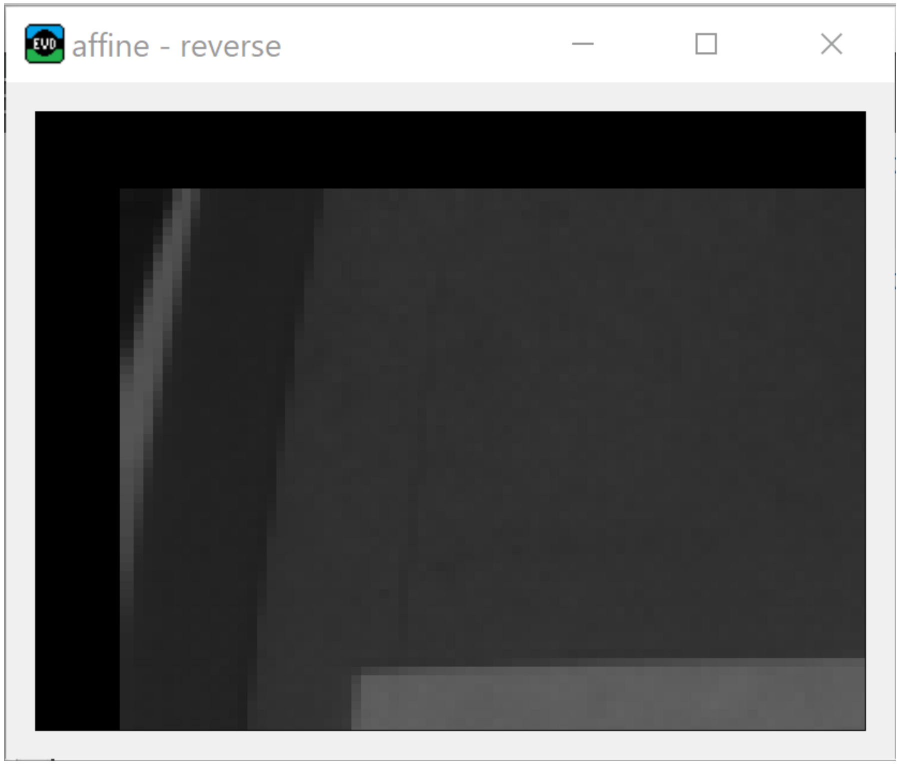
Rewritten:
$x' = x + 0.5y + 0 = x + 0.5y$
$y' = 0.25x + y + 0 = 0.25x + y$

HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation - examples

- Examples: Translate and scale combined

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 2 & 0 & 20 \\ 0 & 2 & 20 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$



Rewritten:

$x' = 2x + 0y + 20 = 2x + 20$

$y' = 0x + 2y + 20 = 2y + 20$

Affine transformation - examples

- Examples: Rotate CCW

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Where

$$\theta = \frac{\pi}{4} \text{ rad}$$

Rewritten:

$$x' = xcos\theta + ysin\theta + 0$$
$$y' = -xsin\theta + ycos\theta + 0$$

Affine transformation - examples

- Examples: Rotate CW

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

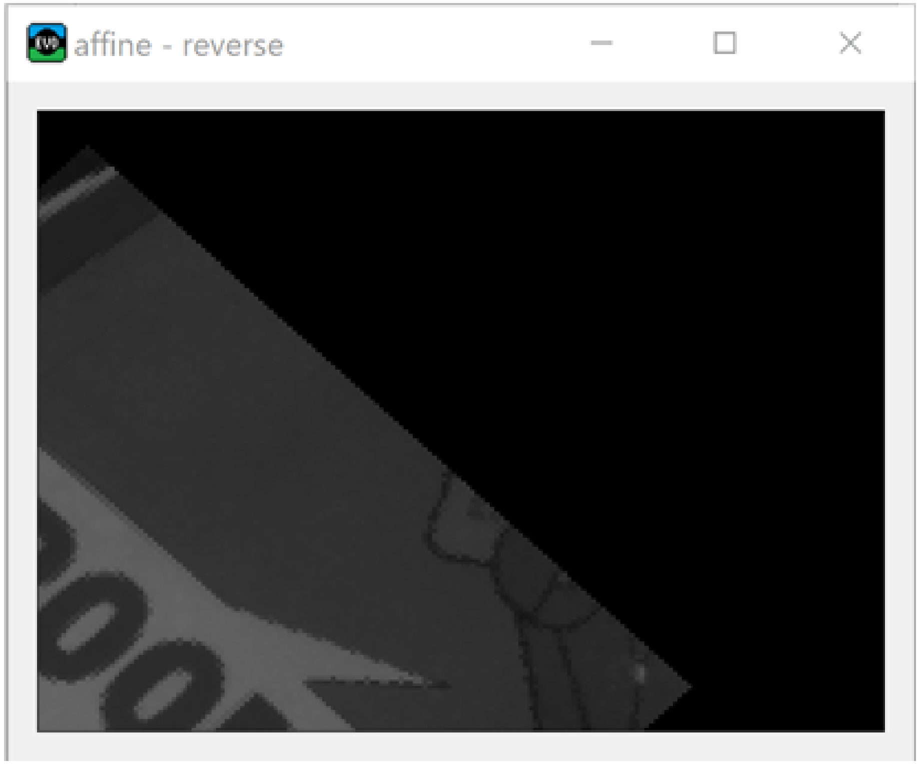
Where

$$\theta = \frac{\pi}{4} \text{ rad}$$

Rewritten:

$x' = xcos\theta - ysin\theta + 0$

$y' = xsin\theta + ycos\theta + 0$



HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation - examples

- CW or CCW rotation ??
- Notice that in images the y-axis points downward
- So, if Wikipedia (for example) says that this rotation matrix

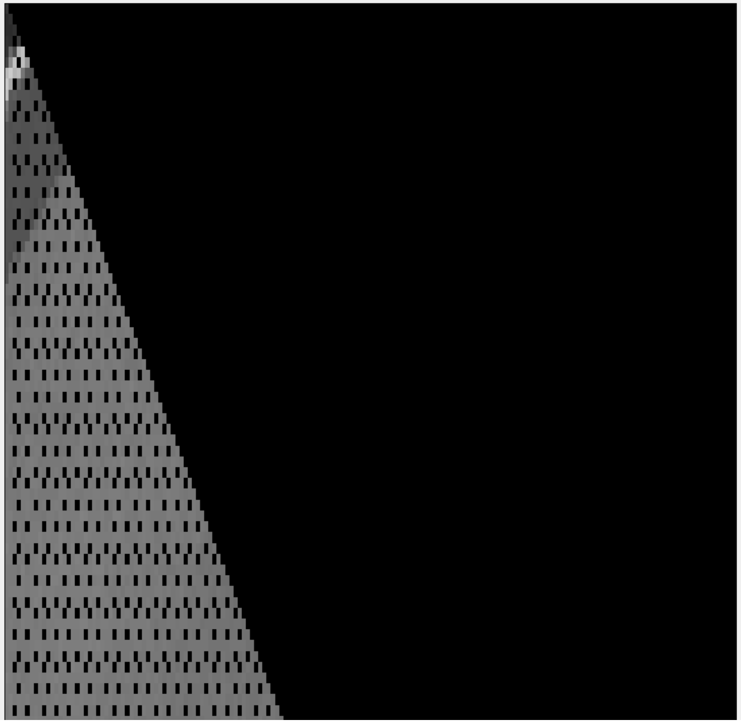
$$\begin{vmatrix} cos\theta & -sin\theta & c \\ sin\theta & cos\theta & f \\ 0 & 0 & 1 \end{vmatrix}$$

causes a CCW rotation, in image processing we will see a CW rotation!

Affine transformation

- What happened with the affine rotated the image?

affine - forward



	0	1	2	3	4	5	6	7	8	9	10	11
0	21	0	0	0	0	0	0	0	0	0	0	0
1	17	19	0	0	0	0	0	0	0	0	0	0
2	17	17	19	0	0	0	0	0	0	0	0	0
3	19	18	0	24	0	0	0	0	0	0	0	0
4	20	42	30	71	77	0	0	0	0	0	0	0
5	27	56	77	0	76	60	0	0	0	0	0	0
6	74	81	72	75	47	34	33	0	0	0	0	0
7	80	68	0	41	34	0	33	32	0	0	0	0
8	77	35	37	33	33	33	32	32	32	0	0	0
9	49	34	33	0	33	33	33	0	32	31	0	0
10	43	34	0	33	32	0	32	33	32	0	32	0
11	34	32	33	33	32	32	32	33	33	33	32	32
12	34	33	32	0	32	32	33	0	33	33	0	32

HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation

- Solution 1: bilinear interpolation
- Solution 2: average filter
- Solution 3: backward transformation instead of forward transformation

HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation

- Solution 1: bilinear interpolation
- Solution 2: average filter
- Solution 3: backward transformation

$$(x', y') = T \cdot (x, y) \quad \Rightarrow \quad (x, y) = \frac{(x', y')}{T}$$

HAN_UNIVERSITY OF APPLIED SCIENCES

Affine transformation

- In matrix calculation, division is performed by multiplication with the inverse matrix

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = T \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \quad \Rightarrow \quad \begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} T^{-1} = \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Affine transformation

- Identity

$$T^{-1} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

- Scaling

$$T^{-1} = \begin{vmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} 1/a & 0 & 0 \\ 0 & 1/e & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

wolframalfa.com

- Translation

$$T^{-1} = \begin{vmatrix} 1 & 0 & c \\ 0 & 1 & f \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} 1 & 0 & -c \\ 0 & 1 & -f \\ 0 & 0 & 1 \end{vmatrix}$$

Affine transformation

- Shear

$$T^{-1} = \begin{vmatrix} 1 & b & 0 \\ d & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} 1/(1-ab) & a/(ab-1) & 0 \\ b/(ab-1) & 1/(1-ab) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

- Rotate CW

$$T^{-1} = \begin{vmatrix} cos\theta & -sin\theta & c \\ sin\theta & cos\theta & f \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} cos\theta & sin\theta & (-c \cdot cos\theta - f \cdot sin\theta) \\ -sin\theta & cos\theta & (c \cdot sin\theta - f \cdot cos\theta) \\ 0 & 0 & 1 \end{vmatrix}$$

- Translation and scale combined

$$T^{-1} = \begin{vmatrix} a & 0 & c \\ 0 & e & f \\ 0 & 0 & 1 \end{vmatrix}^{-1} = \begin{vmatrix} 1/a & 0 & -c/a \\ 0 & 1/e & -f/e \\ 0 & 0 & 1 \end{vmatrix}$$

Affine transformation

- Backward affine transformation

affine - reverse

	0	1	2	3	4	5	6	7	8	9	10	11
0	21	19	19	0	0	0	0	0	0	0	0	0
1	17	19	19	19	0	0	0	0	0	0	0	0
2	17	17	19	24	24	0	0	0	0	0	0	0
3	17	18	30	24	61	77	0	0	0	0	0	0
4	22	42	30	71	77	77	60	0	0	0	0	0
5	66	56	77	76	76	60	35	35	0	0	0	0
6	66	81	72	75	54	34	35	33	32	0	0	0
7	80	68	68	41	34	34	33	32	32	32	0	0
8	55	61	37	33	34	33	32	32	32	32	31	0
9	55	34	33	33	33	33	33	32	32	31	31	32
10	34	34	34	33	32	32	32	33	32	32	32	32
11	33	33	33	33	32	32	32	33	32	33	32	32
12	33	33	32	32	32	32	33	33	33	33	32	32

Affine transformation - algorithm

void **affineTransformation**(const image_t ***src**, image_t ***dst**, eTransformDirection **d**, float **m[][3]**);

See file **EVDK_Operators\graphics_algorithms.c**

```
// Scale example
float a = 2.0f;
float b = 2.0f;

float m_forward[2][3] = {{  a ,   0,   0},
                         {  0 ,   b,   0}};
float m_backward[2][3] = {{ 1/a,  0 , 0},
                          {  0 , 1/b, 0}};

affineTransformation(src, dst, TRANSFORM_BACKWARD, m_backward);
```

Warp

- What does the following transformation matrix accomplish?

$$\begin{vmatrix} X \\ Y \\ Z \end{vmatrix} = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

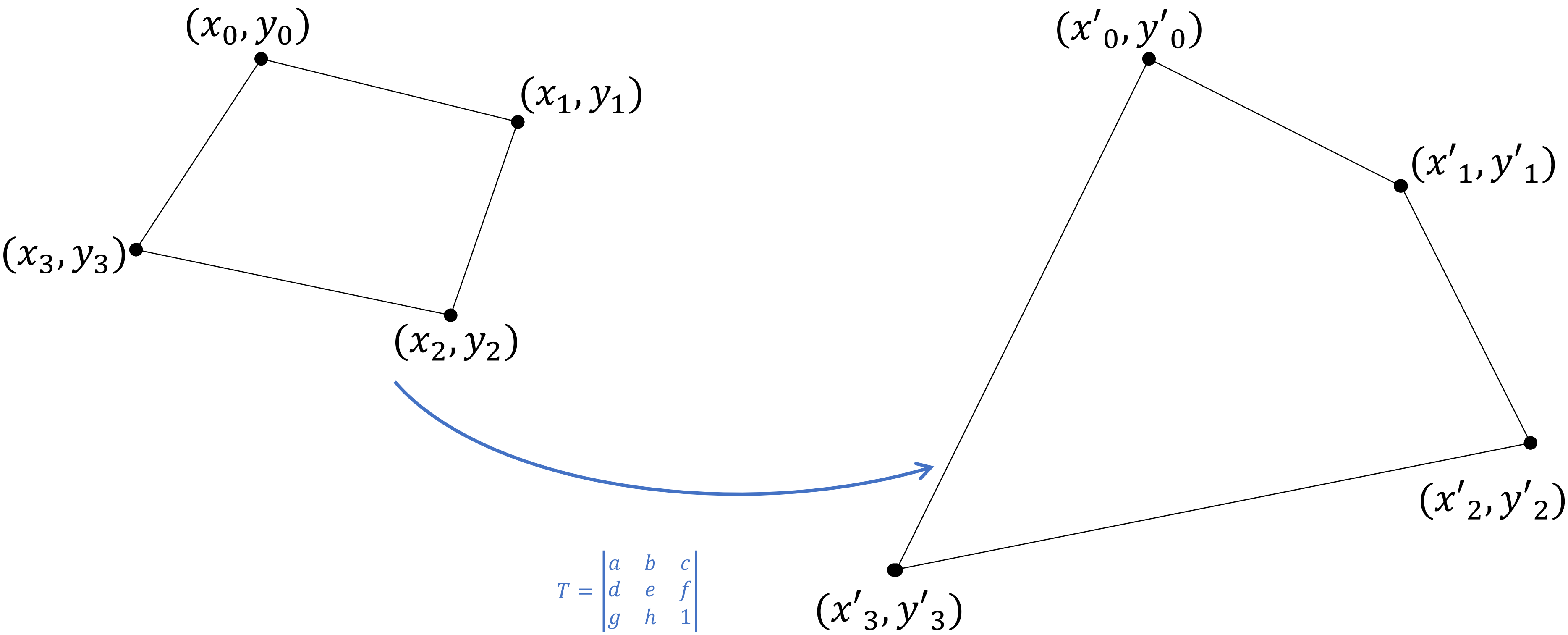
- Rewritten:

$$X = ax + by + c$$
$$Y = dx + ey + f$$
$$Z = gx + hy + 1$$

- This operation is known as geometric distortion
- Some characteristics are:
  - Lines map to lines
  - Parallel lines do not necessarily remain parallel
  - Ratios are not necessarily preserved

Warp

Example



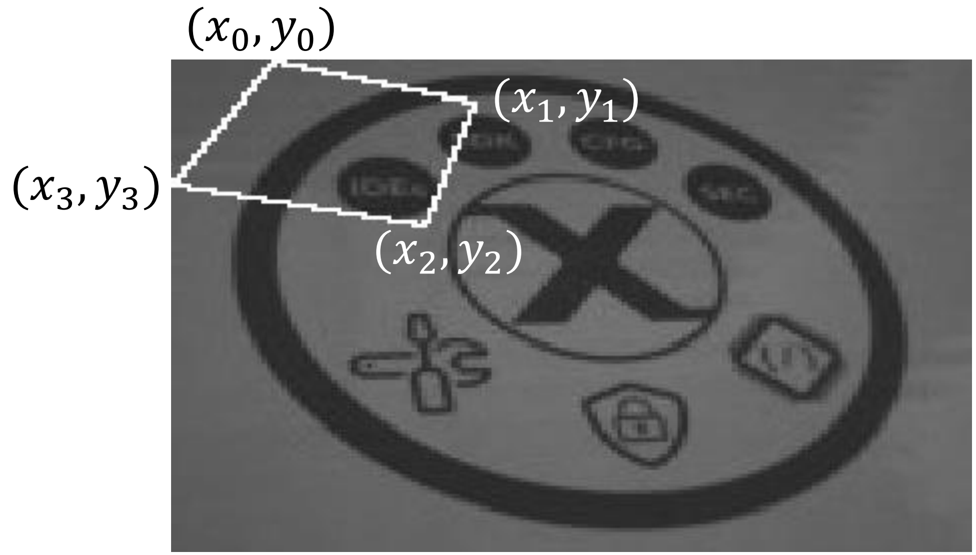
$$T = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix}$$

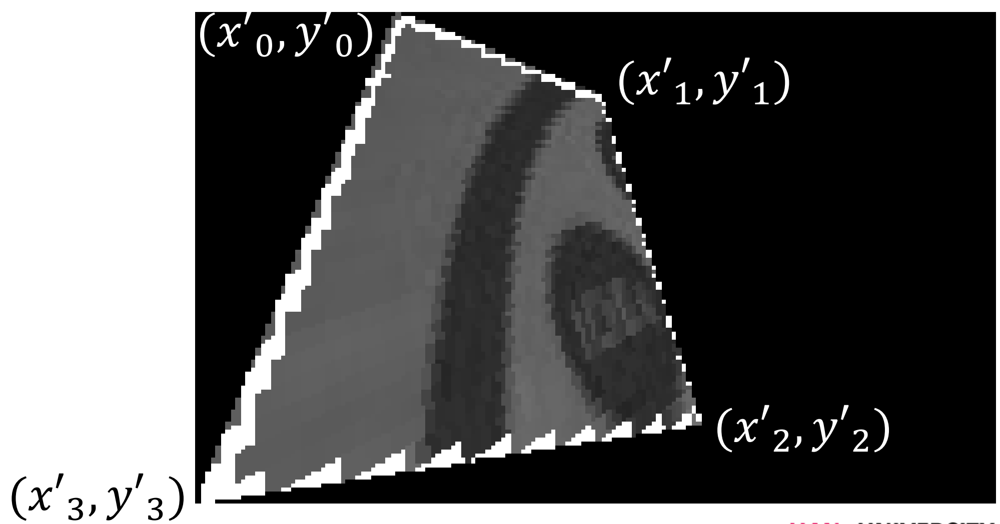
HAN_UNIVERSITY OF APPLIED SCIENCES

Warp - example

From:
$(x_0, y_0) = (20{,}0)$
$(x_1, y_1) = (60{,}10)$
$(x_2, y_2) = (50{,}40)$
$(x_3, y_3) = (0{,}30)$

To:
$(x'_0, y'_0) = (40, 0)$
$(x'_1, y'_1) = (80, 20)$
$(x'_2, y'_2) = (100{,}100)$
$(x'_3, y'_3) = (0{,}119)$





HAN_UNIVERSITY OF APPLIED SCIENCES

Warp

$$X = ax + by + c$$
$$Y = dx + ey + f$$
$$Z = gx + hy + 1$$

- Given a source coordinate (x, y), the destination coordinate (x', y') is then calculated by the functions for perspective equivalence

$$x' = \frac{X}{Z} = \frac{ax + by + c}{gx + hy + 1}$$

$$y' = \frac{Y}{Z} = \frac{dx + ey + f}{gx + hy + 1}$$

Warp

- Rewritten:

$$x' = \frac{X}{Z} = \frac{ax + by + c}{gx + hy + 1} \;\Rightarrow\; \boldsymbol{ax + by + c - gxx' - hyx' = x'}$$

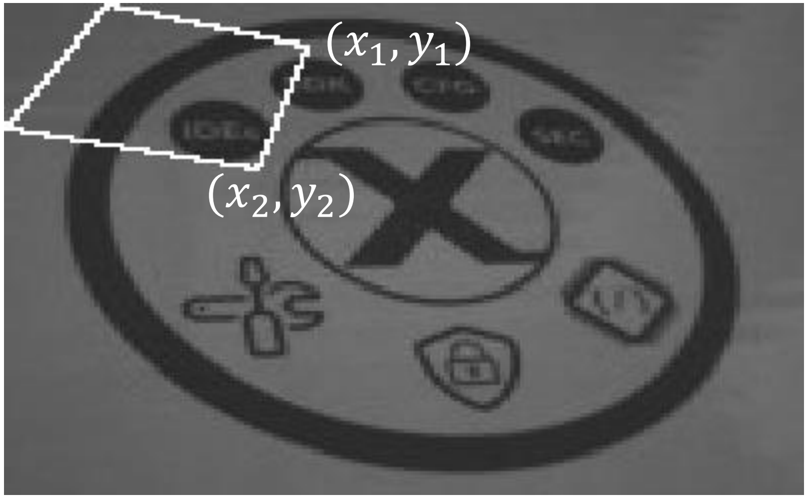
$$y' = \frac{Y}{Z} = \frac{dx + ey + f}{gx + hy + 1} \;\Rightarrow\; \boldsymbol{dx + ey + f - gxy' - hyy' = y'}$$

With these functions for perspective equivalence, the eight transformation matrix coefficients a ... h can be calculated if **four** source coordinates (x, y) and **four** destination coordinates (x', y') are known.

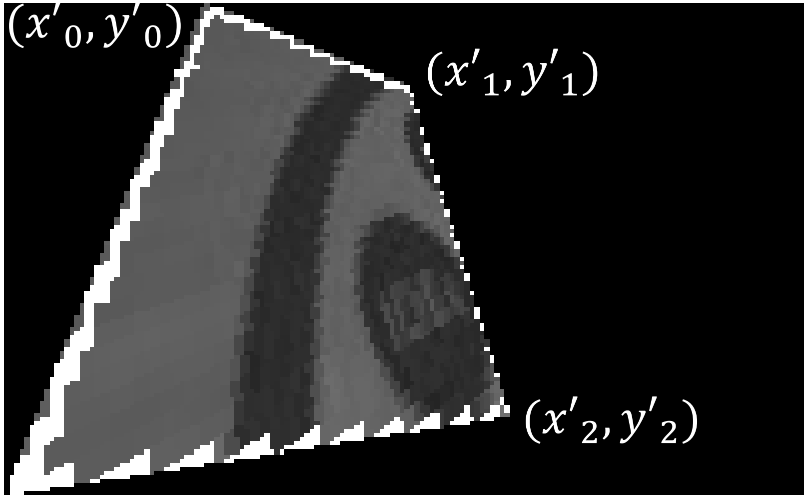
(Because this yields 8 equations in the 8 unknowns a ... h)

Warp - example

From:
$(x_0, y_0) = (20,0)$
$(x_1, y_1) = (60,10)$
$(x_2, y_2) = (50,40)$
$(x_3, y_3) = (0,30)$



To:
$(x'_0, y'_0) = (40, 0)$
$(x'_1, y'_1) = (80, 20)$
$(x'_2, y'_2) = (100,100)$
$(x'_3, y'_3) = (0,119)$



Warp

Let's define these 8 coordinates as follows:

- (x_k, y_k) are the mapping coordinates in the source image
- (x'_k, y'_k) are the mapping coordinates in the destination image

where $k = 0,1,2,3$

And substitute these in the functions for perspective equivalence:

$$\boldsymbol{ax_k + by_k + c - gx_kx'_k - hy_kx'_k = x'_k} \qquad \boldsymbol{dx_k + ey_k + f - gx_ky'_k - hy_ky'_k = y'_k}$$

Then the result is the following 8 equations:

$$\begin{aligned}
ax_0 + by_0 + c - gx_0x'_0 - hy_0x'_0 &= x'_0 \qquad & dx_0 + ey_0 + f - gx_0y'_0 - hy_0y'_0 &= y'_0 \\
ax_1 + by_1 + c - gx_1x'_1 - hy_1x'_1 &= x'_1 \qquad & dx_1 + ey_1 + f - gx_1y'_1 - hy_1y'_1 &= y'_1 \\
ax_2 + by_2 + c - gx_2x'_2 - hy_2x'_2 &= x'_2 \qquad & dx_2 + ey_2 + f - gx_2y'_2 - hy_2y'_2 &= y'_2 \\
ax_3 + by_3 + c - gx_3x'_3 - hy_3x'_3 &= x'_3 \qquad & dx_3 + ey_3 + f - gx_3y'_3 - hy_3y'_3 &= y'_3
\end{aligned}$$

Warp

$$
\begin{aligned}
ax_0 + by_0 + c - gx_0x_0' - hy_0x_0' &= x_0' \\
ax_1 + by_1 + c - gx_1x_1' - hy_1x_1' &= x_1' \\
ax_2 + by_2 + c - gx_2x_2' - hy_2x_2' &= x_2' \\
ax_3 + by_3 + c - gx_3x_3' - hy_3x_3' &= x_3'
\end{aligned}
\qquad
\begin{aligned}
dx_0 + ey_0 + f - gx_0y_0' - hy_0y_0' &= y_0' \\
dx_1 + ey_1 + f - gx_1y_1' - hy_1y_1' &= y_1' \\
dx_2 + ey_2 + f - gx_2y_2' - hy_2y_2' &= y_2' \\
dx_3 + ey_3 + f - gx_3y_3' - hy_3y_3' &= y_3'
\end{aligned}
$$

Rewriting these eight equations in a 8x8 system isolates the coefficients:

$$
\begin{pmatrix}
x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0x_0' & -y_0x_0' \\
x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\
x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\
x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x_3' & -y_3x_3' \\
0 & 0 & 0 & x_0 & y_0 & 1 & -x_0y_0' & -y_0y_0' \\
0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\
0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\
0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y_3' & -y_3y_3'
\end{pmatrix}
\begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix}
=
\begin{pmatrix} x_0' \\ x_1' \\ x_2' \\ x_3' \\ y_0' \\ y_1' \\ y_2' \\ y_3' \end{pmatrix}
$$

Notice that these are all simply x and y values

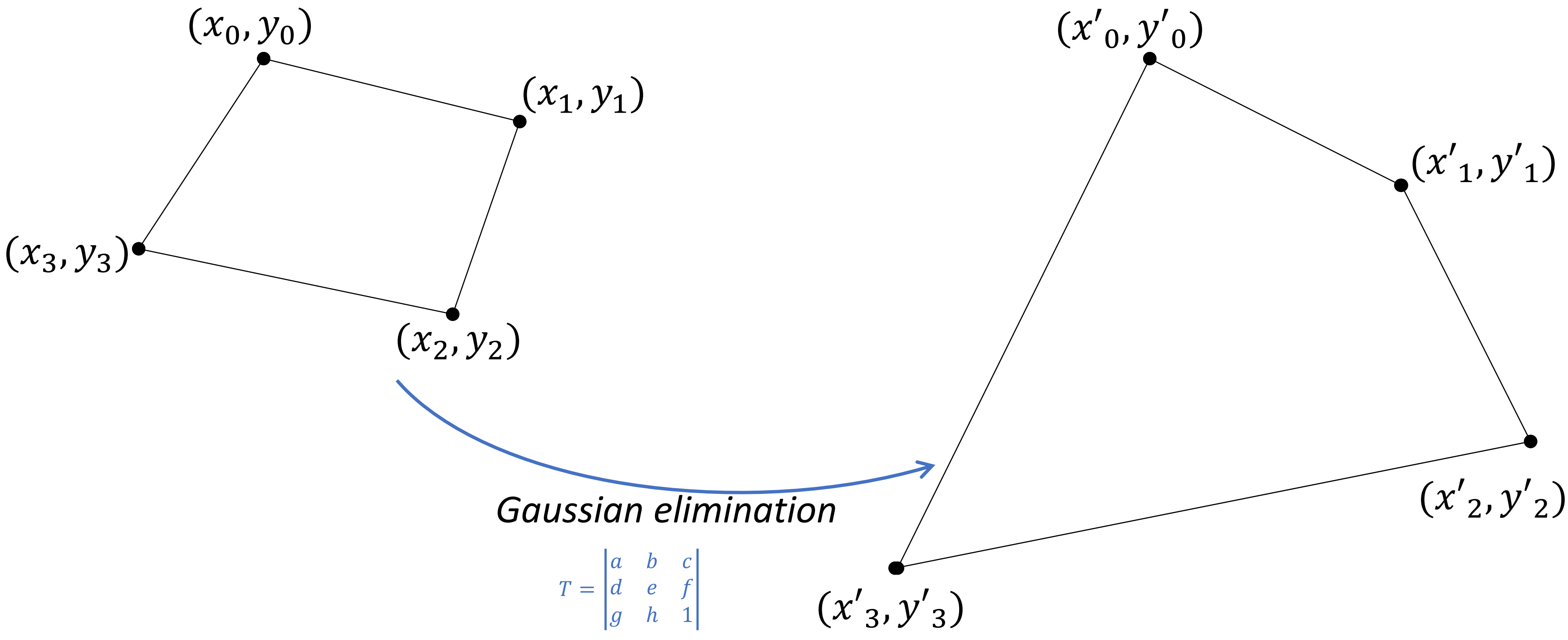
Warp

This 8x8 system can be solved for the coefficients a ... h using Gaussian elimination (amongst other methods).

For example, with a C function for solving this systems of linear equations, provided by:

> Systems of Linear Equations. (n.d.). In *Mathematics Source Library C & ASM*. Retrieved June 6, 2020, from http://www.mymathlib.com/matrices/linearsystems/

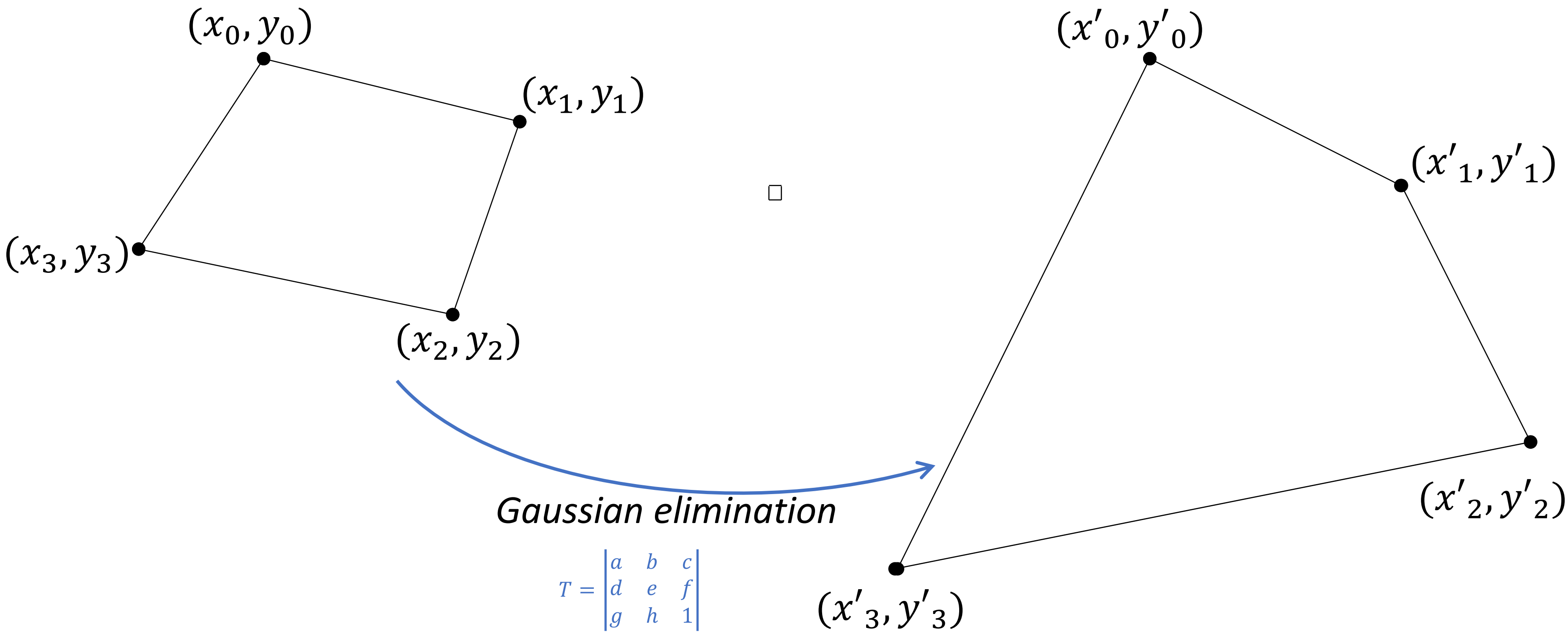
Warp



(x_0, y_0)
(x_1, y_1)
(x_3, y_3)
(x_2, y_2)
(x'_0, y'_0)
(x'_1, y'_1)
(x'_2, y'_2)
(x'_3, y'_3)
Gaussian elimination
$T = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix}$

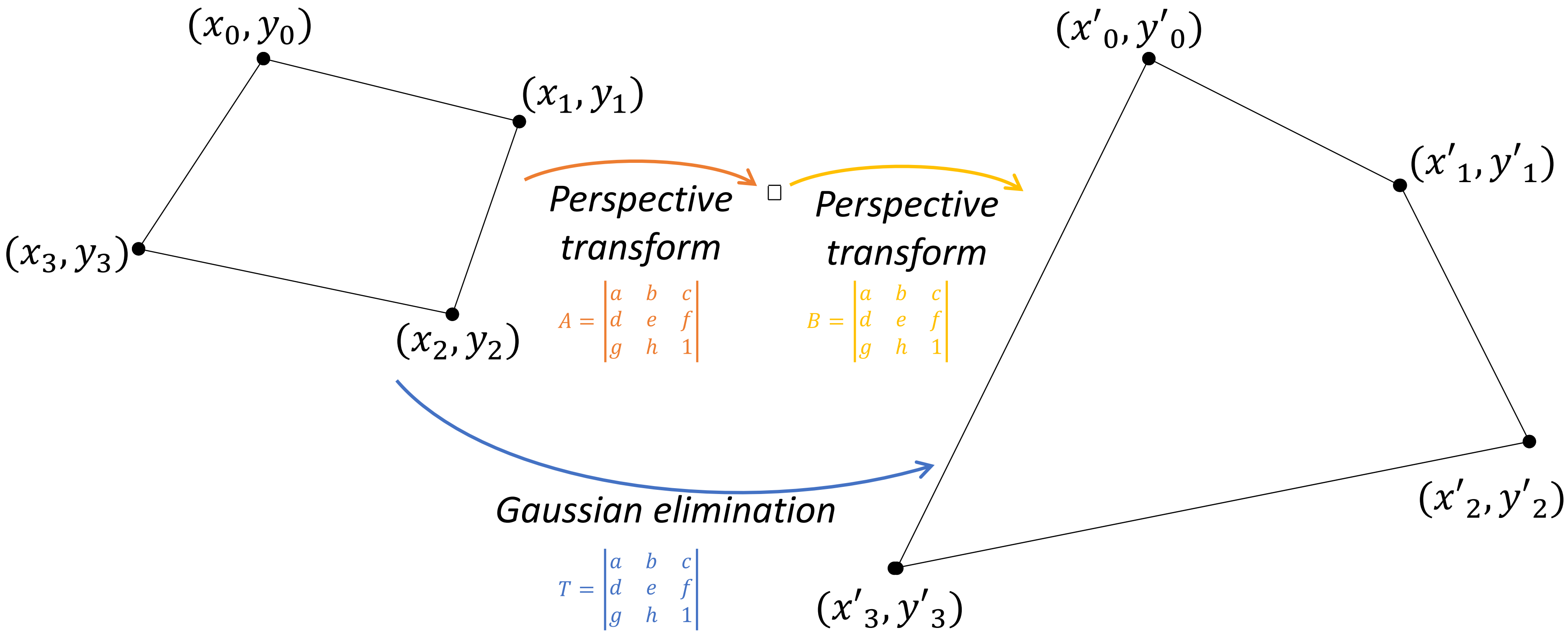
Warp

Use the unit square



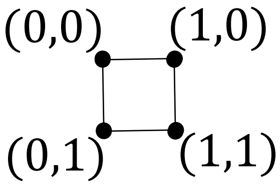
Warp

Use the unit square



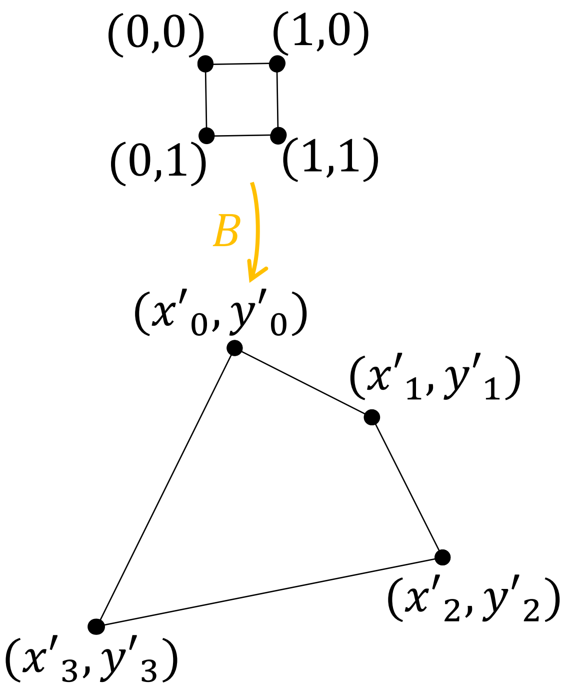
Warp

Use the unit square



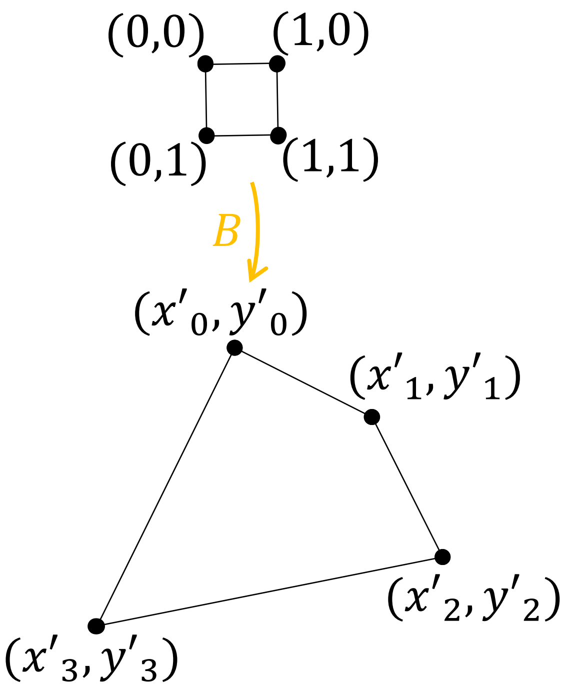
Warp

Use the unit square



Warp

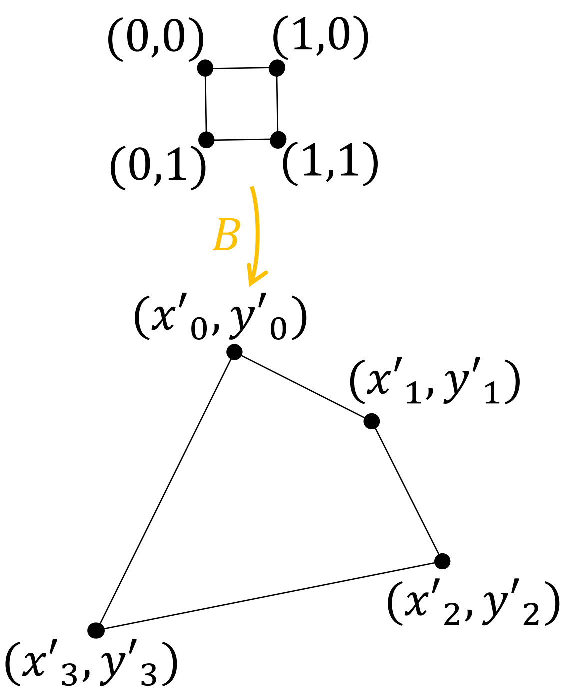
Use the unit square



$$\begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -x_0x'_0 & -y_0x'_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -x_0y'_0 & -y_0y'_0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{pmatrix}$$

Warp

Use the unit square

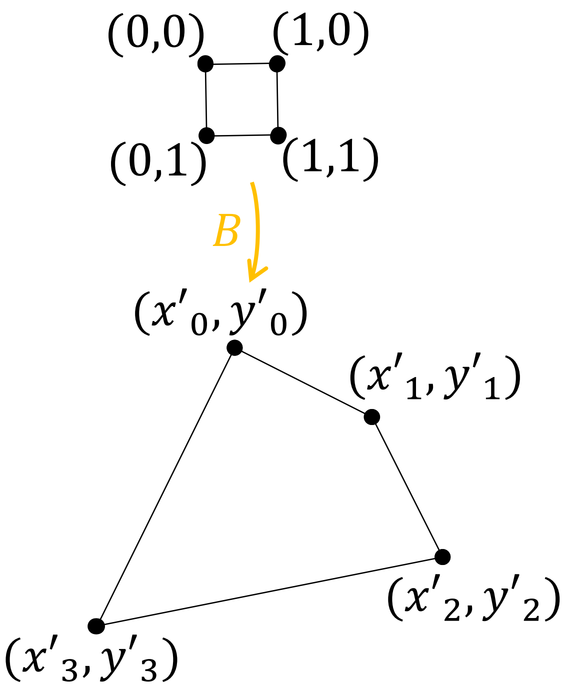


$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -x_1' & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & -x_2' & -x_2' \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & -x_3' \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & -y_1' & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & -y_2' & -y_2' \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & -y_3' \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0' \\ x_1' \\ x_2' \\ x_3' \\ y_0' \\ y_1' \\ y_2' \\ y_3' \end{pmatrix}$$

$$x_0' = c$$

Warp

Use the unit square

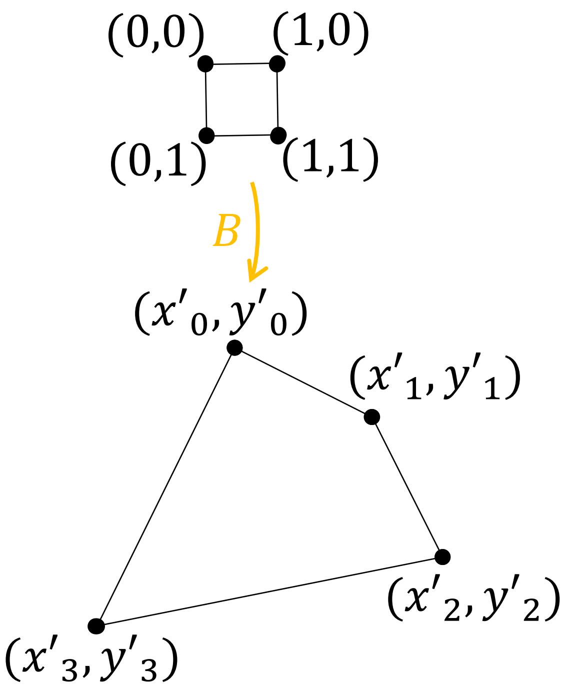


$$
\begin{aligned}
x'_0 &= c \\
x'_1 &= a + c - gx'_1 \\
x'_2 &= a + b + c - gx'_2 - hx'_2 \\
x'_3 &= b + c - hx'_3
\end{aligned}
$$

$$
\begin{aligned}
y'_0 &= f \\
y'_1 &= d + f - gy'_1 \\
y'_2 &= d + e + f - gy'_2 - hy'_2 \\
y'_3 &= e + f - hy'_3
\end{aligned}
$$

Warp

Use the unit square



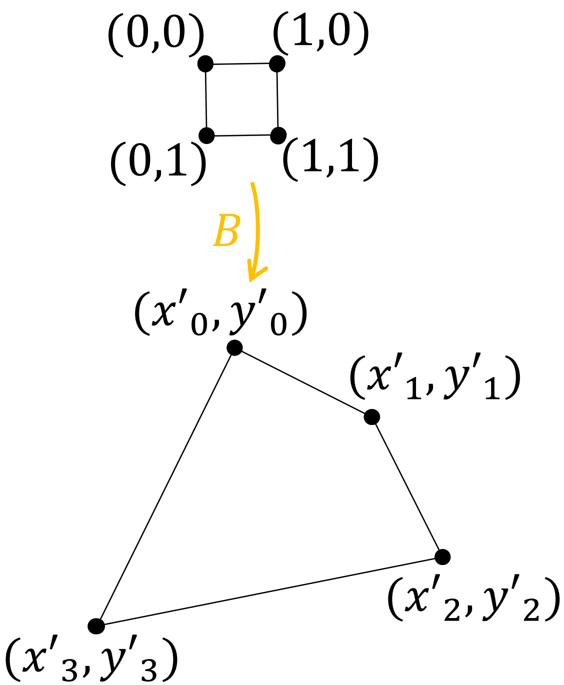
$$
\begin{aligned}
a &= x_1' - x_0' + g x_1' \\
b &= x_3' - x_0' + h x_3' \\
c &= x_0' \\
d &= y_1' - y_0' + g y_1' \\
e &= y_3' - y_0' + h y_3' \\
f &= y_0'
\end{aligned}
$$

$$g = \frac{(y_0' - y_1' + y_2' - y_3')(x_3' - x_2') + (x_0' - x_1' + x_2' - x_3')(y_2' - y_3')}{(y_1' - y_2')(x_3' - x_2') - (y_3' - y_2')(x_1' - x_2')}$$

$$h = \frac{(y_0' - y_1' + y_2' - y_3')(x_2' - x_1') + (x_0' - x_1' + x_2' - x_3')(y_1' - y_2')}{(y_1' - y_2')(x_3' - x_2') - (y_3' - y_2')(x_1' - x_2')}$$

Warp

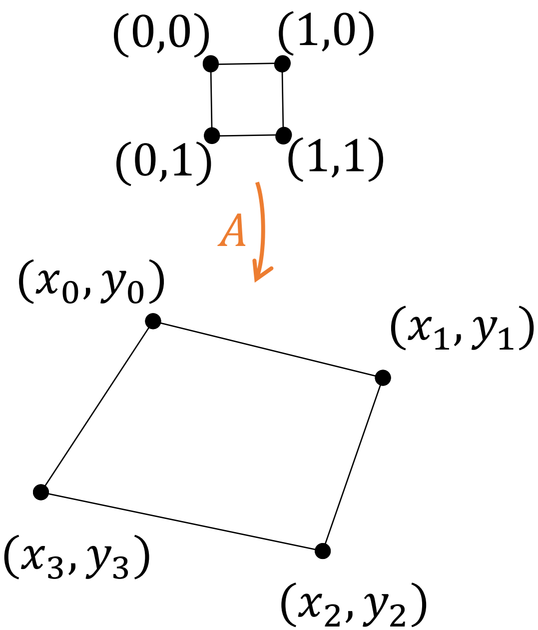
Use the unit square



$$B = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix}$$

Warp

Use the unit square



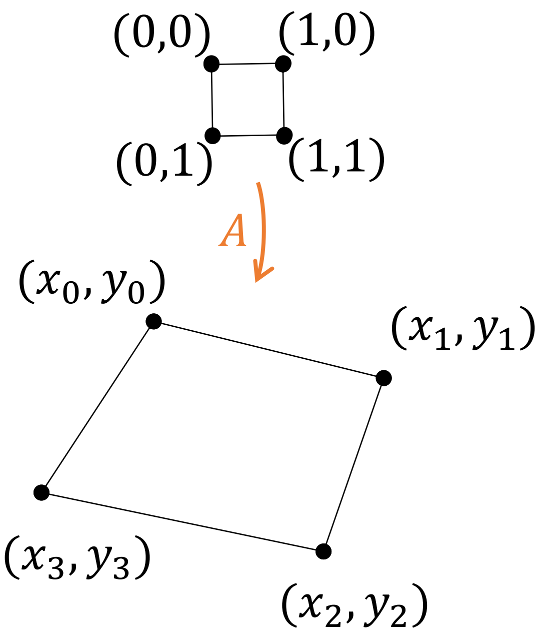
$$
\begin{aligned}
a &= x_1 - x_0 + gx_1 \\
b &= x_3 - x_0 + hx_3 \\
c &= x_0 \\
d &= y_1 - y_0 + gy_1 \\
e &= y_3 - y_0 + hy_3 \\
f &= y_0
\end{aligned}
$$

$$g = \frac{(y_0 - y_1 + y_2 - y_3)(x_3 - x_2) + (x_0 - x_1 + x_2 - x_3)(y_2 - y_3)}{(y_1 - y_2)(x_3 - x_2) - (y_3 - y_2)(x_1 - x_2)}$$

$$h = \frac{(y_0 - y_1 + y_2 - y_3)(x_2 - x_1) + (x_0 - x_1 + x_2 - x_3)(y_1 - y_2)}{(y_1 - y_2)(x_3 - x_2) - (y_3 - y_2)(x_1 - x_2)}$$

Warp

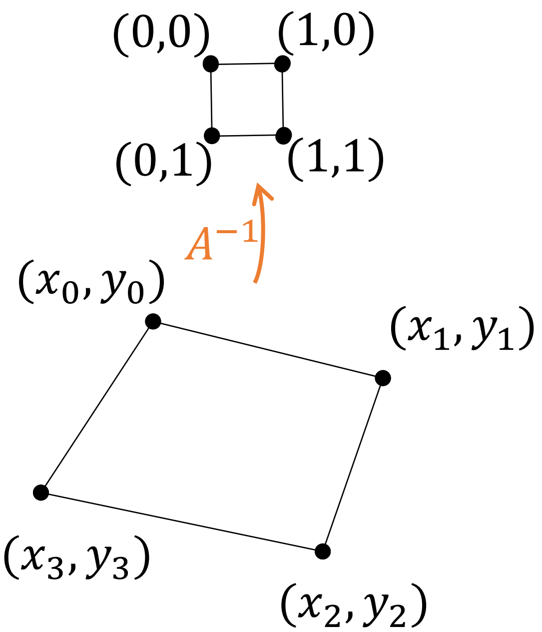
Use the unit square



$$A = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix}$$

Warp

Use the unit square

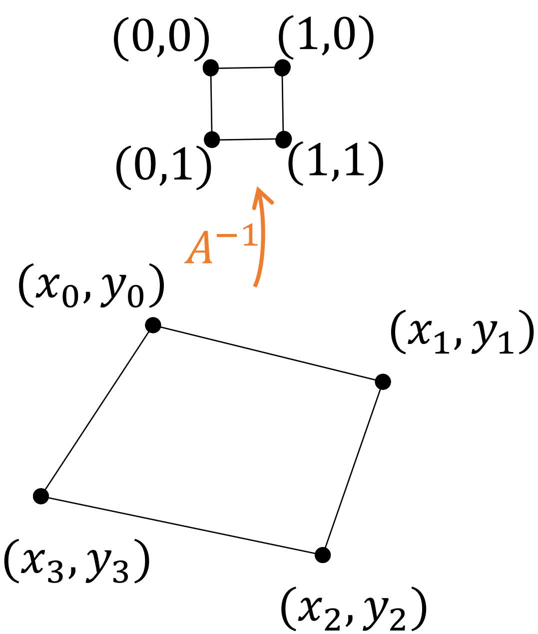


$$A^{-1} = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{vmatrix}^{-1}$$

wolframalfa.com

Warp

Use the unit square

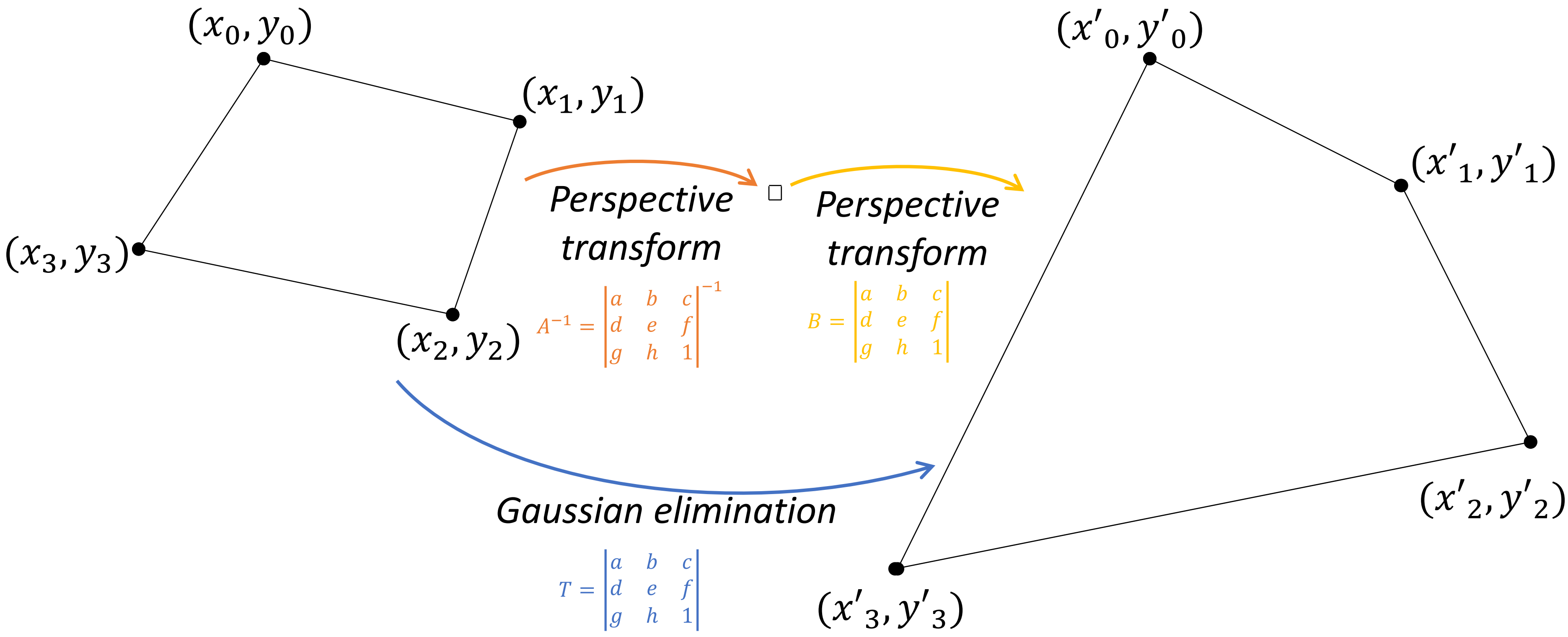


$$A^{-1} = \begin{vmatrix} e - fh & ch - b & bf - ce \\ fg - d & a - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{vmatrix} \cdot \frac{1}{ae - afh - bd + bfg + cdh - ceg}$$

wolframalfa.com

Warp

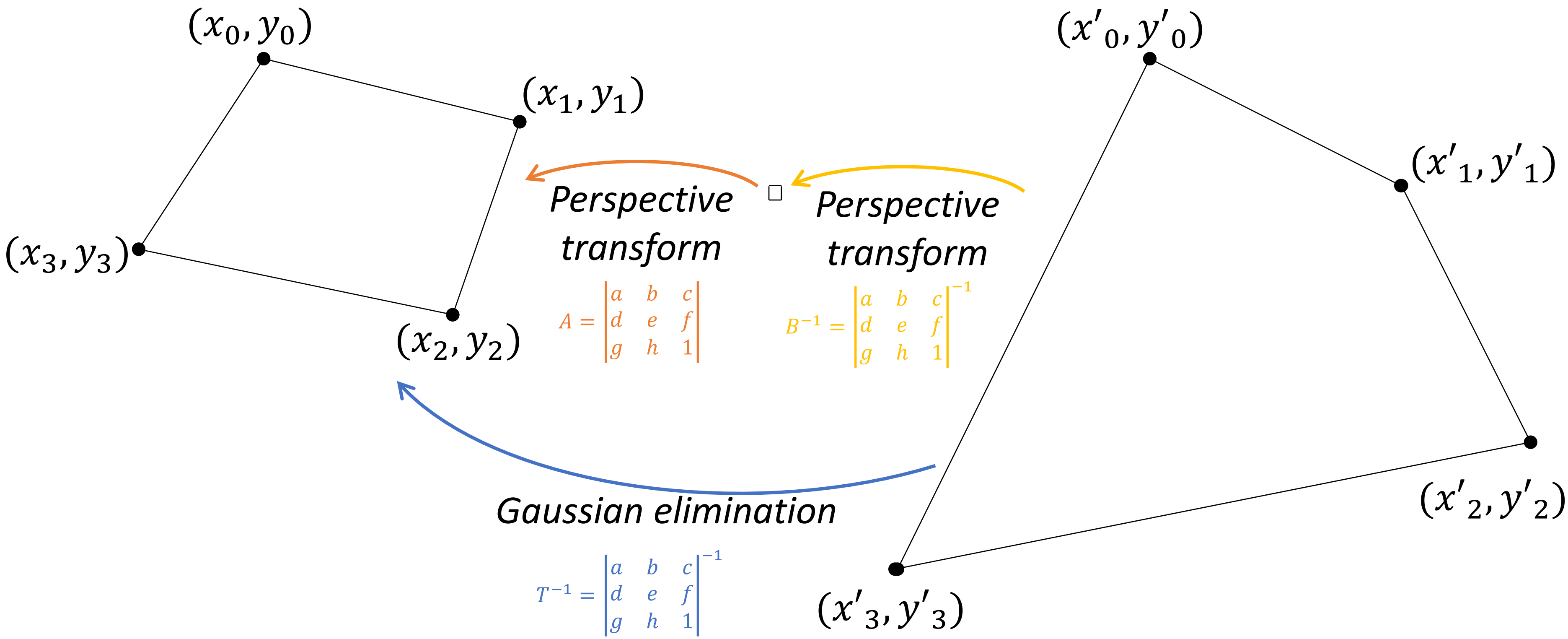
Use the unit square for forward transformation



$$T = B \times A^{-1}$$

Warp

Use the unit square for backward transformation



$$T^{-1} = A \times B^{-1}$$

Warp

And finally, for each pixel in the destination image:

- Given a destination coordinate (x, y), the source coordinate (x', y') is then calculated by the functions for perspective equivalence where the coefficients are given by the inverse matrix T^{-1}

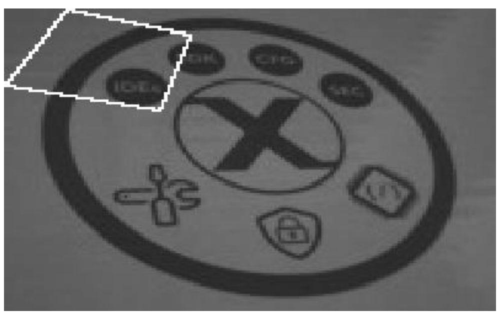
$$x' = \frac{X}{Z} = \frac{ax + by + c}{gx + hy + 1}$$

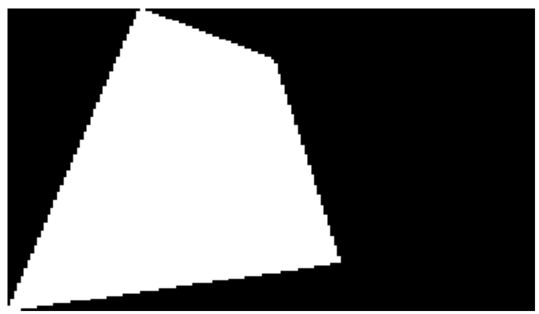
$$y' = \frac{Y}{Z} = \frac{dx + ey + f}{gx + hy + 1}$$

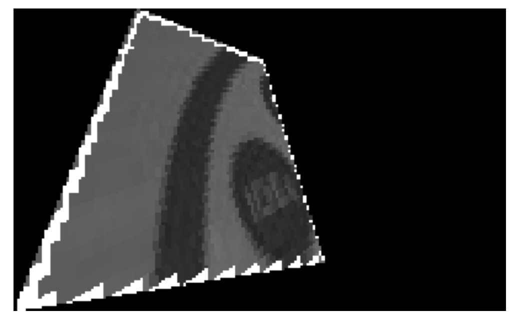
Warp – algorithm

void **warpPerspective**(const image_t ***src**, image_t ***dst**,
const point_t ***from**, const point_t ***to**,
eTransformDirection **d**);

See file **EVDK_Operators\graphics_algorithms.c**







The function additionally creates a temporary mask image to set only pixels that are inside the polygon

Warp – algorithm

void **warpPerspective**(const image_t ***src**, image_t ***dst**, const point_t ***from**, const point_t ***to**, eTransformDirection **d**);

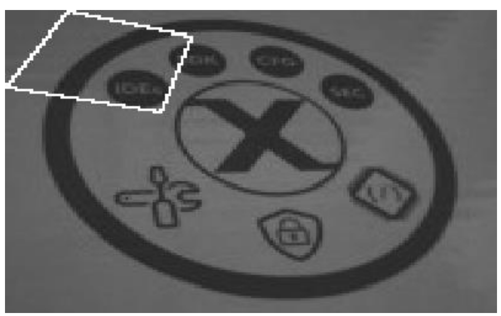
See file **EVDK_Operators\graphics_algorithms.c**

```
const point_t from[4] =
{
    {20, 0},{60, 10},{50, 40},{0, 30}
};

const point_t to[4] =
{
    {40, 0},{80, 20},{100, 100},{0, 119}
};

warpPerspective(src, dst, from, to, TRANSFORM_BACKWARD);
```

Warp – algorithm

void **warpPerspectiveFast**(const image_t ***src**, image_t ***dst**,
const point_t ***from**,
eTransformDirection **d**);

See file **EVDK_Operators\graphics_algorithms.c**



The “to” points are omitted in the function prototype, so it always warps to the dst full range.

Warp – algorithm

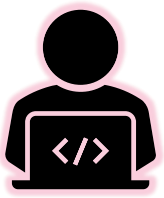
void **warpPerspectiveFast**(const image_t ***src**, image_t ***dst**,
const point_t ***from**,
eTransformDirection **d**);

See file **EVDK_Operators\graphics_algorithms.c**

```
const point_t from[4] =
{
    {20, 0},{60, 10},{50, 40},{0, 30}
};

warpPerspectiveFast(src, dst, from, TRANSFORM_BACKWARD);
```


HAN_UNIVERSITY OF APPLIED SCIENCES

EVD1 – Assignment



Study guide

Week 2

4 Graphics algorithms – affineTransformation()

References

- Myler, H. R., & Weeks, A. R. (2009). *The pocket handbook of image processing algorithms in C*. Prentice Hall Press.

- Affine transformation. (n.d.). In Wikipedia. Retrieved December 12, 2019, from https://en.wikipedia.org/wiki/Affine_transformation

- Heckbert, P. S. (1989). *Fundamentals of texture mapping and image warping*. p. 9-29

- Systems of Linear Equations. (n.d.). In *Mathematics Source Library C & ASM*. Retrieved June 6, 2020, from http://www.mymathlib.com/matrices/linearsystems/

References

- Wikipedia contributors. (2024, May 3). Rotation (mathematics). In *Wikipedia, The Free Encyclopedia*. Retrieved 08:04, October 7, 2024,from https://en.wikipedia.org/w/index.php?title=Rotation_(mathematics)&oldid=1222018142