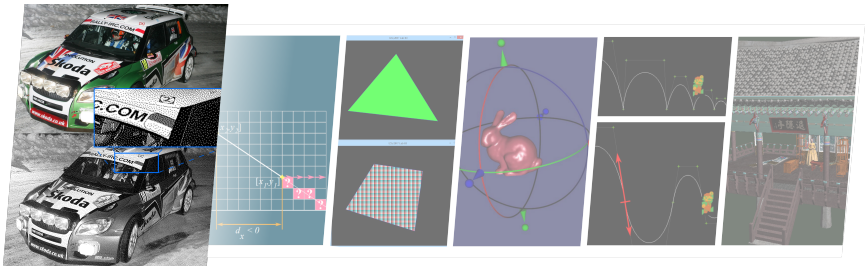
IZG 23/24L Lab I: Color Space Reduction

Ing. Pavol Dubovec (idubovec@fit.vut.cz)

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

IZG 23/24L Labs in a Nutshell

1. **Color space reduction**
2. Basic 2D object rasterization
3. Filling of 2D closed regions
4. 3D transformations
5. 2D spline curves
6. OpenGL basics



You can obtain up to 3 points for each lab by solving individual coding tasks.

Code Base

- Implementations in C/C++, building with CMake.
- SDL 2.0 library for multi-platform multimedia applications (sound, graphics, ...), with HW acceleration.
  - Already within the code structure, **it is unnecessary to download it**.
- You will implement your work **in student.cpp only**.

Plan for Today: Color Space Reduction Algorithms

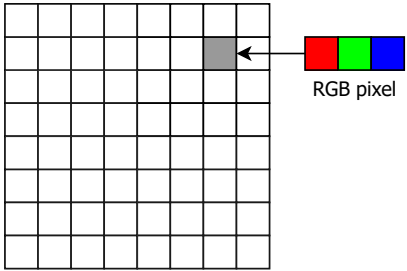
Given the leftmost color image, replace the original colors with black and white values only (rightmost image), with the intermediate step of grayscale conversion (middle image).

Strive for the visually most appropriate appearance.

We need to understand how (i) images are represented in computers, (ii) how colors are represented, and (iii) what algorithms to use.

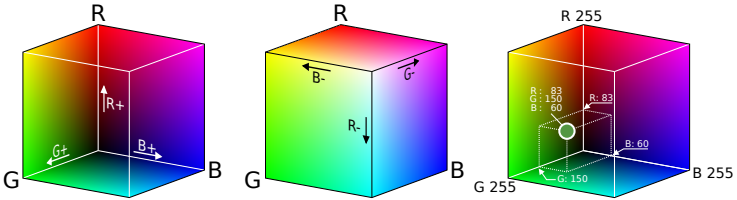
Images in Computers: Frame Buffer

- Images are *usually* represented as arrays of *pixels*
- Pixel = pix (in the sense "pictures") + el (first two letters from element)
- Usually perceived as 2D arrays
- **In IZG labs, frame buffer is internally represented as 1D array (more effective)**



Colors in Computers, RGB Color Model

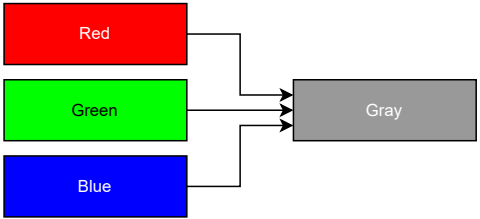
```
struct RGB
{
    uint8_t r;
    uint8_t g;
    uint8_t b;
};
```



- Frame buffer is represented as `RGB* frame_buffer` (three-channel 1D array).
- Majority of operations are based on direct writing into the frame buffer memory. For all algorithms, methods that modify framebuffer have already been implemented.


RALLY-IRC.COM
Skoda
www.skoda.co.uk


RALLY-IRC.COM
Skoda
www.skoda.co.uk



Important step since all the halftoning techniques operate on grayscale images!

Input

- RGB: $[0, 255] \times [0, 255] \times [0, 255]$

Output

- Grayscale value: $[0, 255]$
- : $I = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$

In RGB (i.e., in our case):

- All components are set to the grayscale value: $R = G = B = I$

Task 1: implement the following function (0.5 points)

- `void ImageTransform::grayscale()`

Helper variables

- `uint32_t cfg->w`
- `uint32_t cfg->h`

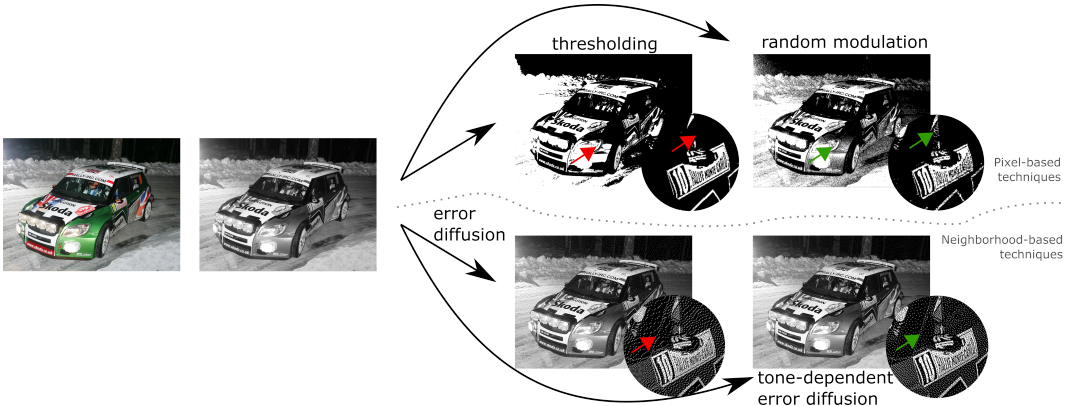
Helper methods and functions

- `void setPixel(uint32_t x, uint32_t y, RGB color)`
- `RGB getPixel(uint32_t x, uint32_t y)`
- `std::round(x)` e.g.: `unsigned x = std::round(42.45f);`

How to empirically test the functionality?

1. Load one of the images (press "L" or "K").
2. Press "G"to apply the conversion.

Core of this Lab: Digital Halftoning Techniques



You will implement two techniques (one base, one improved version) of *pixel-based* and *neighborhood-based* halftoning algorithms (4 in total).

You should be familiar with the base versions from the lecture. Improved versions are new and address typical artifacts of their base versions.

Always convert to grayscale first!

Various methods exist:

- Thresholding (Task 2.1)
- Random Modulation (Task 2.2 variant A)
- Ordered Bayer Dithering (Task 2.2 variant B)
- Clustered-Dot Dithering (Task 2.2 variant C)
- *Random Dithering* (*Task 2.2 sample solution*)
- ...

Methods evaluate one pixel at a time. They are fast but ignore pixel interactions. Thus, they produce less visually plausible outputs.

Task 2.1: implement the following function (0.75 points)

- `void ImageTransform::threshold()`

Helper variables

- `COLOR_WHITE`
- `COLOR_BLACK`
- `uint32_t cfg->w`
- `uint32_t cfg->h`

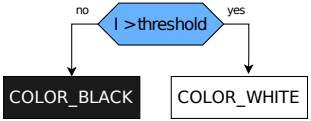
Helper methods and functions

- `void ImageTransform::grayscale()`

How to empirically test the functionality?

1. Load one of the images (press "L" or "K").
2. Press "T"to apply the conversion.

For each pixel in grayscale image:



It does its job, but *a lot* of visual information is lost.

Task 2.2 variant A: implement the following function (0.5 points)

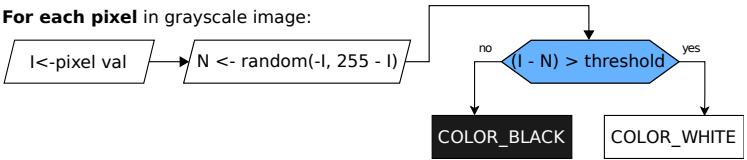
- `void ImageTransform::randomModulation()`

Helper methods and functions are the same as for the thresholding function, plus

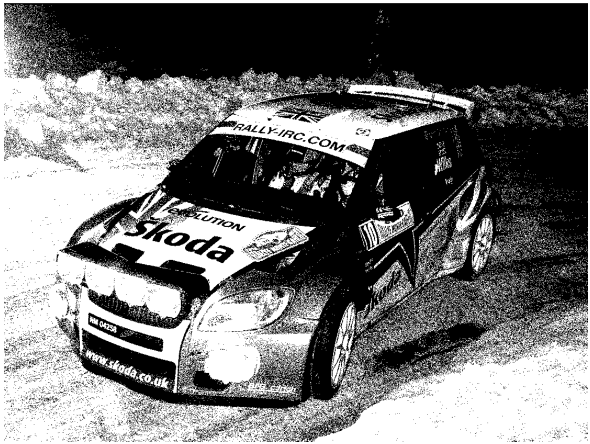
- `int getRandomFromRange(int lowerBound, int upperBound)`

Hints: Get inspired by thresholding method. Random number might be negative so be careful with conversions (`int` vs. `uint8_t`).

To empirically test the functionality, load the image and apply the algorithm by pressing "M".



Section II: Pixel-based Ht., Random Modulation, Result



With minor yet clever extension (by adding noise), we managed to improve the visual appearance of the B/W result.

Task 2.3 variant B: implement the following function (0.5 points)

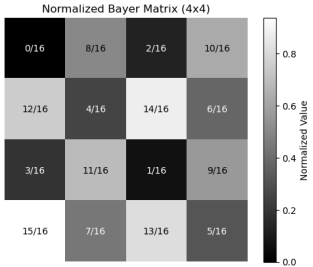
- `void ImageTransform::orderedBayerDithering()`

Helper variables

- `bayerMatrix2x2, bayerMatrix4x4, bayerMatrix8x8`

Hints:

- Use a predefined (**already normalised**) Bayer dithering matrix.
- You will need a variable to store the size of the matrix used.
- For each pixel in the image, use the corresponding value in the matrix as the threshold.
- The matrix should be repeated (tiled) across the entire image. Use the modulo operator (%) to achieve this. This ensures that the matrix repeats correctly.



Example Normalised Bayer Matrix

To empirically test the functionality, load an image and apply the algorithm by pressing "B".

Section II: Pixel-based Ht., Ord. Bayer Dithering, Result

By distributing the quantization error in a structured manner, we managed to enhance the image, making it more detailed and aesthetically pleasing.

Task 2.3 variant C: implement the following function (0.5 points)

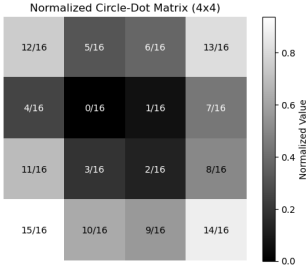
- `void ImageTransform::clusteredDotDithering()`

Helper variables

- `circleBayerMatrix4x4, circleBayerMatrix8x8`

Hints:

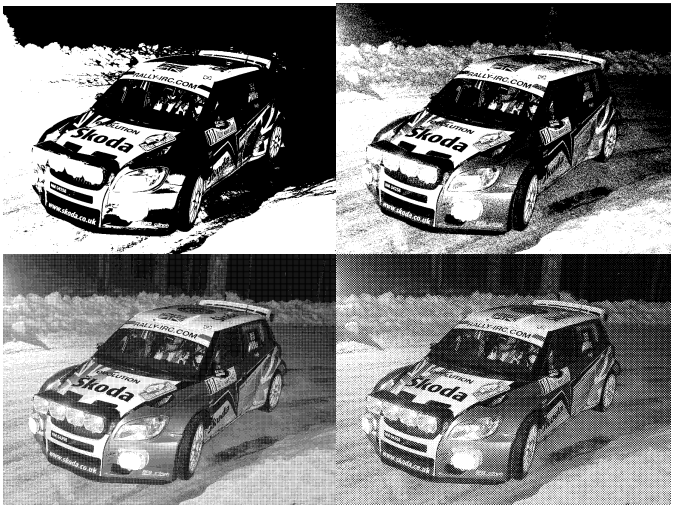
- Use a predefined (**already normalised**) clustered dot dithering matrix.
- You will need a variable to store the size of the matrix used.
- For each pixel in the image, use the corresponding value in the matrix as the threshold.
- The matrix should be repeated (tiled) across the entire image. Use the modulo operator (%) to achieve this. This ensures that the matrix repeats correctly.



Example Normalised Clustered-Dot Matrix

To empirically test the functionality, load an image and apply the algorithm by pressing "C".

By distributing the quantization error in a structured manner, we managed to enhance the image, making it more detailed and aesthetically pleasing.



(↖) Thresholding
- high-frequency artifacts.

(↗) Random Modulation
- adds noise to reduce artifacts.

(↙) Bayer Dithering
- uses a structured matrix to distribute quantization error.

(↘) Clustered-Dot Dithering
- groups dots to create a more visually appealing pattern, often used in printing.

Previous pixel-based methods omitted pixel interactions – which does not reflect reality and how the human visual cortex works!

Always convert to grayscale first!

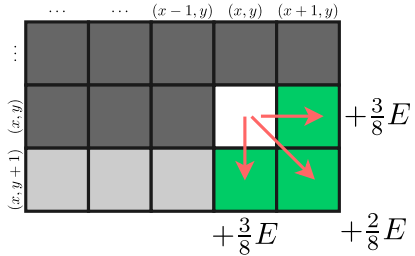
Various methods exist:

- Error Distribution (Task 3.1 variant A)
- Floyd-Steinberg Dithering (Task 3.1 variant B)
- Tone-Dependent Error Distribution (Task 3.2)
- Error Distribution with Hilbert Curves (BTS of this exercise)
- Error Distribution with Threshold Modulation (BTS of this exercise)
- ...

Each pixel depends on multiple pixels in its vicinity. Extra computations need to be done (runs slower), but more visually plausible results are generated.

$$G(x,y) = \begin{cases} 255 & I(x,y) > \mathit{threshold} \\ 0 & \text{else} \end{cases}$$

$$E = \begin{cases} I(x,y) - 255 & \text{if } G(x,y) = 255 \\ I(x,y) & \text{if } G(x,y) = 0 \end{cases}$$



1. Extract resulting value G (black or white) for processed pixel by comparing with threshold (you know how to perform this).
2. Compute error E according to formula above. Hint: the error value might be negative.
3. Distribute the error to surrounding pixels. Use distribution coefficients as shown in figure above. For this, call (three times) prepared function `updatePixelWithError(uint32_t x, uint32_t y, float err)`!
4. Do not forget to set the value of currently processed pixel to G.

Task 3.1 variant A: Implement the following function (0.75 points)

- `void ImageTransform::errorDistribution()`

Helper methods and functions are the same as in previous tasks, plus

- `void updatePixelWithError(uint32_t x, uint32_t y, float err)`
  - Use this to distribute calculated errors to neighborhood pixels.
  - Please note that the function already checks invalid access (*out-of-array*) for you.

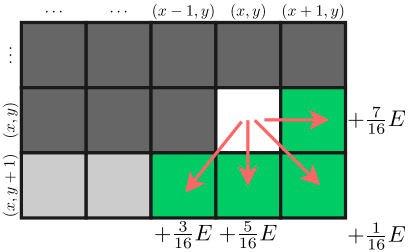
How to empirically test the functionality?

1. Load one of the images (press "L" or "K").
2. Press "E" to apply the conversion.

This looks really good, but if you zoomed in, you would notice typical error-diffusion artifacts (see slide 28).

$$G(x,y) = \begin{cases} 255 & I(x,y) > \mathit{threshold} \\ 0 & \text{else} \end{cases}$$

$$E = \begin{cases} I(x,y) - 255 & \text{if } G(x,y) = 255 \\ I(x,y) & \text{if } G(x,y) = 0 \end{cases}$$



1. Extract resulting value G (black or white) for processed pixel by comparing with threshold (you know how to perform this).
2. Compute error E according to formula above. Hint: the error value might be negative.
3. Distribute the error to surrounding pixels. Use distribution coefficients as shown in figure above. For this, call (four times) prepared function `updatePixelWithError(uint32_t x, uint32_t y, float err)`!
4. Do not forget to set the value of currently processed pixel to G.

Task 3.1 variant B: Implement the following function (0.75 points)

- `void ImageTransform::FloydSteinbergDithering()`

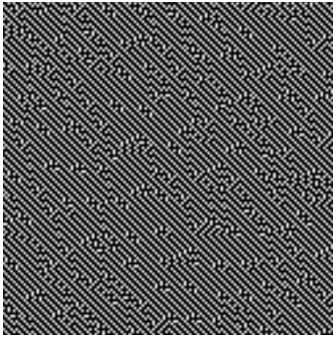
Helper methods and functions are the same as in previous tasks, plus

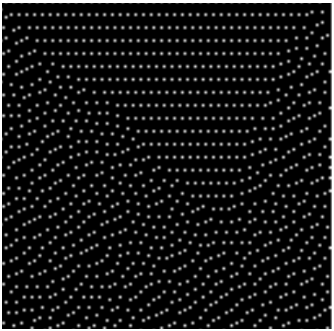
- `void updatePixelWithError(uint32_t x, uint32_t y, float err)`
  - Use this to distribute calculated errors to neighborhood pixels.
  - Please note that the function already checks invalid access (*out-of-array*) for you.

How to empirically test the functionality?

1. Load one of the images (press "L" or "K").
2. Press "F" to apply the conversion.

This looks really good, but if you zoomed in, you would notice typical error-diffusion artifacts (see slide 28).

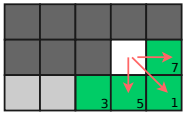




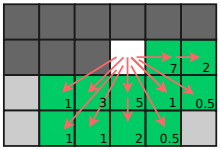
Typical artifacts: maze-like structures, worm-like alignments and edge instabilities.

Let's try to suppress such artifacts with the tone-dependent version of the algorithm!

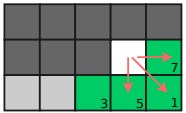
Section III: Tone-Dependent Error Dist.



Distribution Type A



Distribution Type B



Distribution Type A

The distribution of error varies according to the intensity value of the currently processed pixel. All other steps are similar to classical error diffusion (see 28).

If `I(x,y) in (0,40) or I(x,y) in (215,255)`, apply distribution version A (see Figure above), otherwise, for mid-tones, apply version B.

Task 3.2: Implement the following function (0.5 points)

- `void ImageTransform::toneDependentErrorDistribution()`

Helper methods and functions are the same as in previous tasks, plus

- `void updatePixelWithError(uint32_t x, uint32_t y, float err)`
  - Use this to distribute calculated errors to neighborhood pixels.
  - Hint: this time, the parameters and number of function calls will be dependent on the value of the processed pixel.
  - Please note that the function already checks invalid access (*out-of-array*) for you.

How to empirically test the functionality?

1. Load one of the images (press "L" or "K").
2. Press "W" to apply the conversion.

Many artifacts of error distribution are suppressed. Best visible when zoomed in (see slide 32).

Now it's time for you to implement the algorithms!

- Ask your TA (teaching assistant) for any help! We are happy to assist you when you struggle with coding for too long or if you do not understand the assignment.
- Note: most of the problems you may encounter are caused by incorrect or no explicit conversion between data types.
- When you are done, raise your hand and your TA will look at your solution. Be ready to answer some simple questions about your solution.