Wykrywanie krawędzi z wykorzystaniem filtra Sobela - Systemy Dedykowane w Układach Programowalnych

Autorzy: **Filip Wierzbinka Daniel Żaba**

Repozytorium projektu: https://github.com/wierzba100/Sobel_Filter/tree/main

Wstęp teoretyczny - Konwersja na skalę szarości

Konwersja obrazu na skalę szarości polega na przekształceniu obrazu kolorowego (**RGB**) w obraz, który zawiera tylko odcienie szarości. W obrazach RGB każdy piksel jest reprezentowany przez trzy wartości: czerwoną (**R**), zieloną (**G**) i niebieską (**B**). Konwersja do skali szarości sprowadza się do obliczenia jednej wartości jasności dla każdego piksela, która zastępuje oryginalne wartości RGB. Polega to na przemnożeniu wartości każdego koloru przez odpowiednie współczynniki, które po dodaniu dadzą wartość 1. Istnieje kilka metod konwersji obrazu na skalę szarości:

Metoda średniej arytmetycznej

Polega na obliczeniu średniej arytmetycznej wartości RGB: **Gray = (R+G+B)/3**

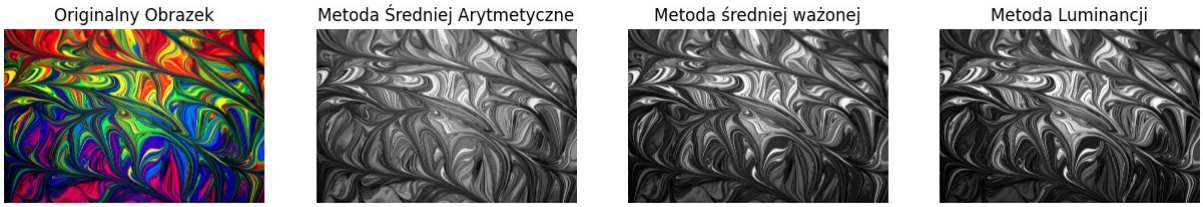
Metoda średniej ważonej

Ta metoda uwzględnia fakt, że ludzkie oko jest bardziej wrażliwe na zielony kolor niż na czerwony i niebieski. Dlatego stosuje się odpowiednie wagi dla każdego koloru: **Gray=0.299·R+0.587·G+0.114·B** Ta metoda jest często stosowana, ponieważ daje wyniki bliższe temu, jak postrzegamy jasność w rzeczywistości.

Metoda luminancji

Jest to bardziej zaawansowana metoda, która również uwzględnia różne wagi dla kolorów, ale bazuje na modelach widzenia człowieka: **Gray=0.2126·R+0.7152·G+0.0722·B** Tę metodę zastosowano w poniższej implementacji filtra Sobela.

Poniżej porównanie 3 powższych metod:



Wstęp teoretyczny - Operator Sobla

Aby wykorzystywać filtr Sobela najpierw konwertuje się obraz na skalę szarości (można używać filtr Sobela na obrazie kolorowym, ale wtedy otrzyma się osobne krawędzie dla 3 kanałów - **R**(czerwonego), **G**(zielonego), **B**(niebieskiego)). Kolejnym etapm jest zamiana obrazu na macież pikseli. Za pomocą filtra Sobela można filtrować krawędzie w różnych kierunkach - poziomym i pionowym (można także filtrować krawędzie ukośne, natomiast najczęściej używa się pionowego i poziomego). Aby przefiltrować krawędzie we wszystkich kierunkach należy wykonać splot otrzymanych wyników z filtracji pionowej i poziomej.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

+1	0	-1
+2	0	-2
+1	0	-1

kernel Gx

+1	+2	+1
0	0	0
-1	-2	-1

kernel Gy

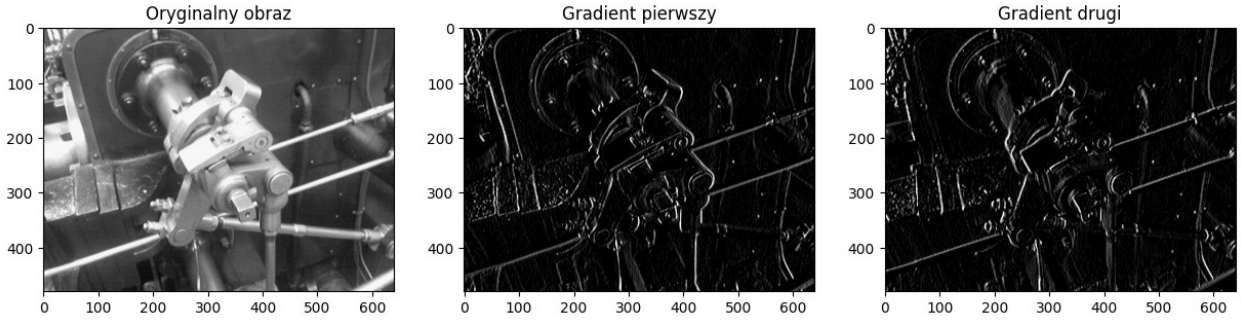
-2	-1	0
-1	0	+1
0	+1	+2

kernel for "/" edges

0	+1	+2
-1	0	+1
-2	-1	0

kernel for "\" edges

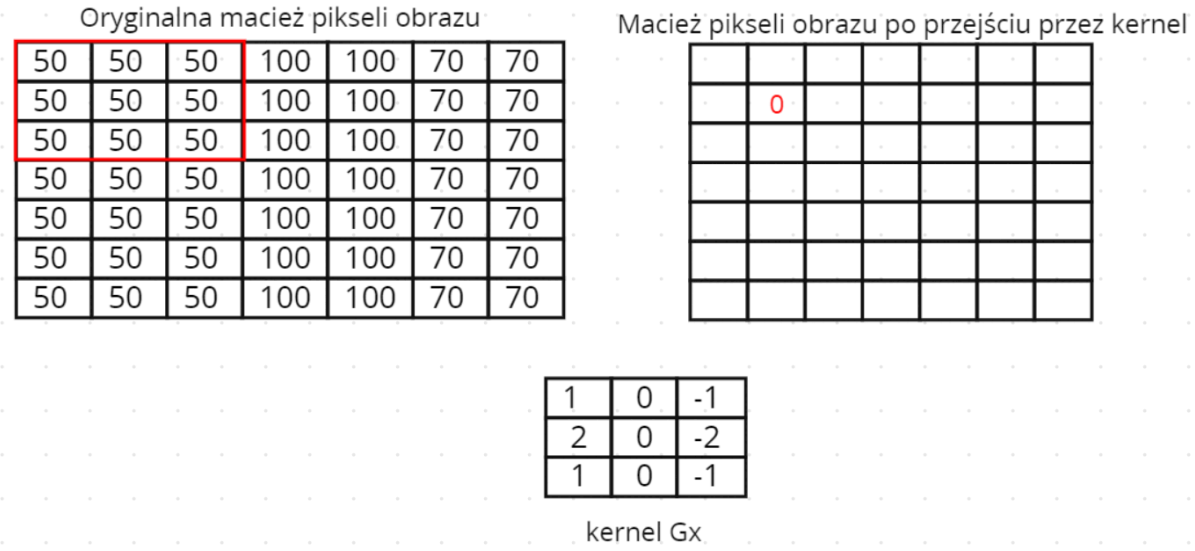
Można zauważyć, że z jednej strony kernela wartości są dodatnie, a z drugiej ujemne. Różnica polega na tym, że jeśli po lewej są wartości dodatnie (**kernel_1**) to jest bardziej wrażliwy gdy zmiana jasności przechodzi od ciemnego do jasnego z lewej do prawej. Natomiast jeśli po lewej są wartości ujemne (**kernel_2**), to jest bardziej wrażliwy gdy zmiana jasności przechodzi od jasnego do ciemnego z lewej do prawej. Różnica między nimi pokazana jest poniżej (poniżej kodu, którego wynikiem jest poniższy obrazek).



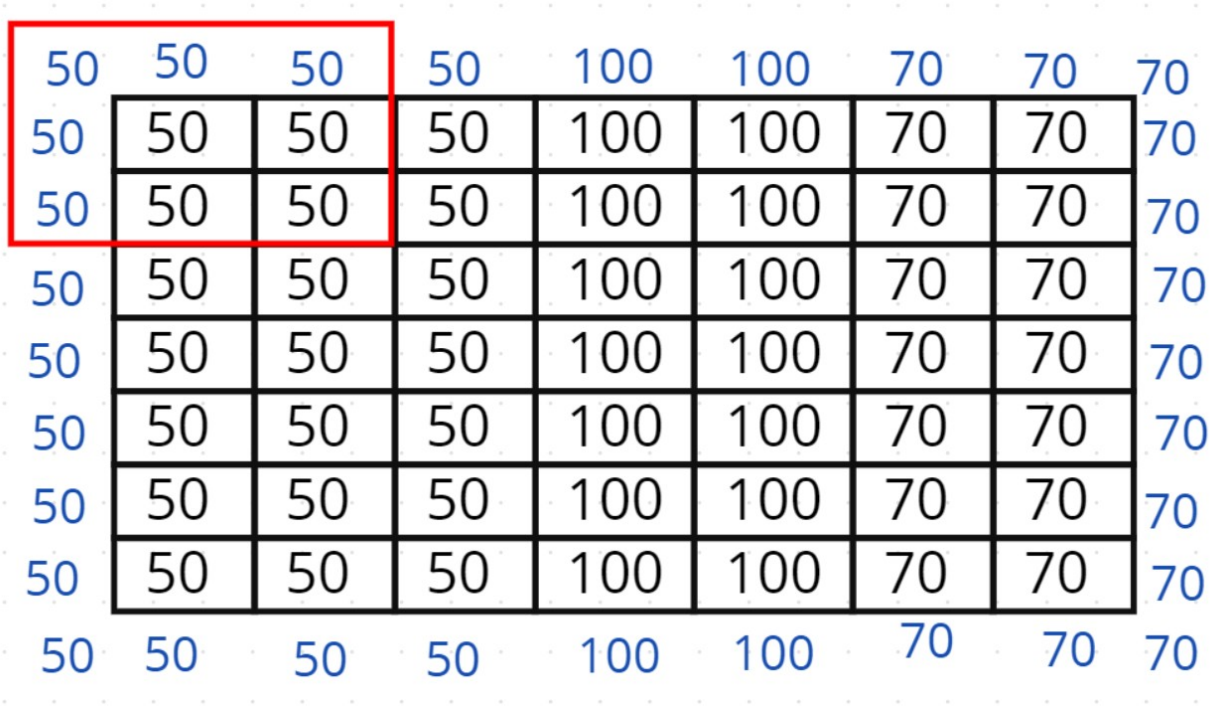
Nie ma to znaczenia po której stronie są ujemne, a po której dodatnie jeśli filtrujemy krawędzie w obu kierunkach (ponieważ i tak podnosimy wynik do kwadratu) [na ten temat więcej informacji niżej] . Kernel **Gx** służy do wykrywania **pionowych** krawędzi (zmiany jasności w kierunku

poziomym), a kernel **Gy** służy do wykrywania **poziomych** krawędzi (zmiany jasności w kierunku pionowym).

Filtr Sobela oblicza aproksymację gradientu intensywności obrazu za pomocą dwóch macierzy 3x3, które są stosowane osobno w poziomym (Gx) i pionowym (Gy) kierunku. Następnie te dwa gradienty są łączone, aby uzyskać ogólny obraz gradientu. Konwolucja: Każda z macierzy (Gx i Gy) jest konwolutowana z obrazem. Konwolucja polega na przesuwaniu macierzy filtru po obrazie, obliczaniu sumy iloczynów elementów macierzy i odpowiadających im pikseli obrazu, a następnie przypisaniu tej wartości do odpowiedniego piksela w wyniku



Jak widać powyżej krawędzie wyjściowej macieży są puste. Jednym ze sposobów aby temu zaradzić jest wykorzystanie takich samych wartości jak w polach sąsiednich, gdy kernel "wystaje" poza macierz obrazu (pokazano poniżej).



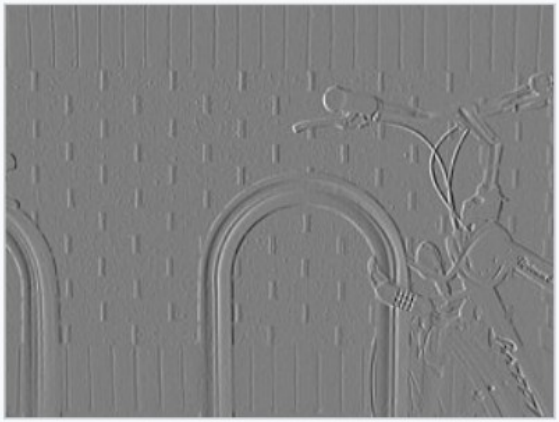
Wykrywanie krawędzi w obu kierunkach robi się w ten sam sposób jak w jednym kierunku z tą różnicą, że nie wpisyje się odrazu wyniku do macieży wyjściowej (jak było to pokazane wyżej). Otrzymane wyniki z kerneli Gx i Gy podnosi się do kwardatu, dodaje, a następnie pierwiastkuje (według wzoru poniżej).

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Tak otrzymany wynik wpisuje się do tabeli wyjściowej Można także do wartości z macierzy wyjściowych z kreneli Gx i Gy zasosować powyższy wzór i wpisać te dane do trzeciej macierzy. Poniżej porównanie wyjściowych gradientów po przepuszczeniu przez kernel Gx, Gy oraz po połączeniu obu.

Grayscale test image of brick wall and bike rack

Normalized gradient magnitude from Sobel–Feldman operator



Normalized *x*-gradient from Sobel–Feldman operator

Normalized *y*-gradient from Sobel–Feldman operator

Czasem dla lepszego efektu przed zastosowaniem filtra Sobela "przepuszcza się" obraz przez filtr Gaussa. Powoduje to wygładzenie obrazu, dzięki czemu różne artefakty na zdjęciu nie są traktowane jako krawędzie.

Nasza implementacja filtra Sobela

Do filtracji obrazu filtrem Sobela wykorzystaliśmy Jupyter Notebook oraz FPGA Kria KV260. Wybrany obraz wgrywamy do Jupyter'a. W nim zamieniamy obraz na tablicę numpy z surowymi danymi (macierz pikseli). Takie dane są przesyłane do FPGA, które sprzętowo dokonuje konwersji RGB na skalę szarości oraz filtracji Sobela, a następnie przesyła dane do Jupyter'a. Wejściowy oraz wyjściowy obraz prezentowane są w Jupyter'ze.

Poniżej znajduje się implementacja

Import wymaganych bibliotek

```
from PIL import Image
from pynq import allocate, Overlay
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

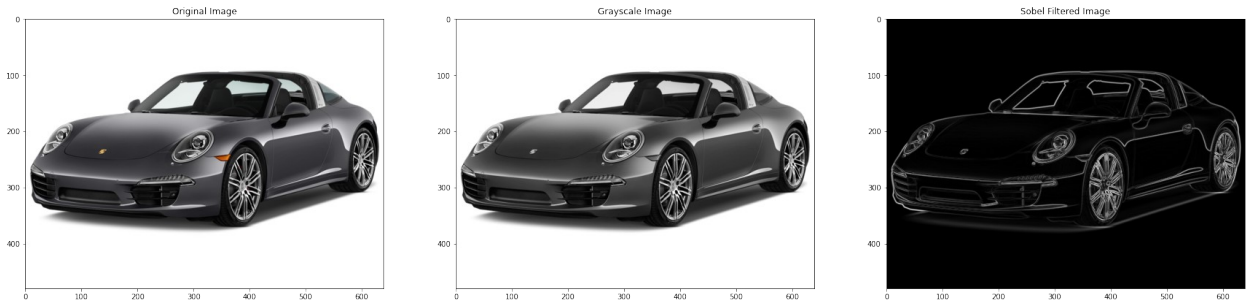
Dane wejściowe

```
image_file = 'Images/car.png'
#image_file = 'Images/engine.png'
#image_file = 'Images/star.png'
#image_file = 'Images/arrows.png'
#image_file = 'Images/steve_jobs.png'
#image_file = 'Images/jupiter_earth_comparison.png'

gamma_mul = 1.0 #range from 0.0 to 2.55

with Image.open(image_file) as img:
    IMAGE_WIDTH, IMAGE_HEIGHT = img.size
```

Model wysokopoziomowy

```
input_image = plt.imread(image_file)

r_img, g_img, b_img = input_image[:, :, 0], input_image[:, :, 1],
input_image[:, :, 2]

gamma = 1.4
r_const, g_const, b_const = 0.2126, 0.7152, 0.0722
grayscale_image = r_const * r_img * gamma + g_const * g_img * gamma +
b_const * b_img * gamma

"""
     |                 |                |                  |
     | 1.0   0.0  -1.0 |                |  1.0   2.0   1.0 |
Gx = | 2.0   0.0  -2.0 |    and     Gy = |  0.0   0.0   0.0 |
     | 1.0   0.0  -1.0 |                | -1.0  -2.0  -1.0 |
     |                 |                |                  |
"""
```

```
Gx = np.array([[1.0, 0.0, -1.0], [2.0, 0.0, -2.0], [1.0, 0.0, -1.0]])
Gy = np.array([[1.0, 2.0, 1.0], [0.0, 0.0, 0.0], [-1.0, -2.0, -1.0]])

[rows, columns] = np.shape(grayscale_image)
sobel_filtered_image = np.zeros(shape=(rows, columns))

for i in range(rows - 2):
    for j in range(columns - 2):
        gx = np.sum(np.multiply(Gx, grayscale_image[i:i + 3, j:j +
3]))
        gy = np.sum(np.multiply(Gy, grayscale_image[i:i + 3, j:j +
3]))
        sobel_filtered_image[i + 1, j + 1] = np.sqrt(gx ** 2 + gy **
2)

fig = plt.figure(figsize=(30, 30))

fig.add_subplot(1, 3, 1)
plt.imshow(input_image)
plt.title('Original Image')

fig.add_subplot(1, 3, 2)
plt.imshow(grayscale_image, cmap=plt.get_cmap('gray'))
plt.title('Grayscale Image')

fig.add_subplot(1, 3, 3)
plt.imshow(sobel_filtered_image, cmap=plt.get_cmap('gray'))
plt.title('Sobel Filtered Image')

plt.show()

# plt.imsave('Images/sobel_filtered_image.png', sobel_filtered_image,
cmap=plt.get_cmap('gray'))
```



Implementacja na Kria KV260 - Model FPGA

```
def convert_png_to_raw(jpg_path):
```

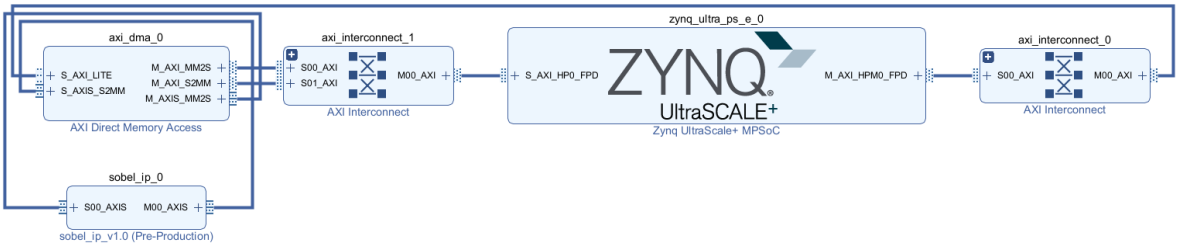
```
    img = Image.open(jpg_path)

    img_array = np.array(img)

    raw_rgb_data = np.zeros((IMAGE_HEIGHT, IMAGE_WIDTH),
dtype=np.uint32)

    raw_rgb_data = img_array[:, :, 0] + (img_array[:, :, 1] << 8) +
(img_array[:, :, 2] << 16) + (np.uint8(gamma_mul* 100) << 24)

    return raw_rgb_data

sobel_filter_ov = Overlay("sobel_design_wrapper.xsa")

sobel_filter_ov?

Type:            Overlay
String form:     <pynq.overlay.Overlay object at 0xffff7fbaad40>
File:            /usr/local/share/pynq-venv/lib/python3.10/site-
packages/pynq/overlay.py
Docstring:
Default documentation for overlay sobel_design_wrapper.xsa. The
following
attributes are available on this overlay:

IP Blocks
----------
axi_dma_0            : pynq.lib.dma.DMA
zynq_ultra_ps_e_0    : pynq.overlay.DefaultIP

Hierarchies
-----------
None

Interrupts
----------
None

GPIO Outputs
------------
None

Memories
------------
PSDDR                : Memory
Class docstring:
This class keeps track of a single bitstream's state and contents.

The overlay class holds the state of the bitstream and enables run-
time
protection of bindings.
```

```
Our definition of overlay is: "post-bitstream configurable design".
Hence, this class must expose configurability through content
discovery
and runtime protection.

The overlay class exposes the IP and hierarchies as attributes in the
overlay. If no other drivers are available the `DefaultIP` is
constructed
for IP cores at top level and `DefaultHierarchy` for any hierarchies
that
contain addressable IP. Custom drivers can be bound to IP and
hierarchies
by subclassing `DefaultIP` and `DefaultHierarchy`. See the help
entries
for those class for more details.

This class stores four dictionaries: IP, GPIO, interrupt controller
and interrupt pin dictionaries.

Each entry of the IP dictionary is a mapping:
'name' -> {phys_addr, addr_range, type, config, state}, where
name (str) is the key of the entry.
phys_addr (int) is the physical address of the IP.
addr_range (int) is the address range of the IP.
type (str) is the type of the IP.
config (dict) is a dictionary of the configuration parameters.
state (str) is the state information about the IP.

Each entry of the GPIO dictionary is a mapping:
'name' -> {pin, state}, where
name (str) is the key of the entry.
pin (int) is the user index of the GPIO, starting from 0.
state (str) is the state information about the GPIO.

Each entry in the interrupt controller dictionary is a mapping:
'name' -> {parent, index}, where
name (str) is the name of the interrupt controller.
parent (str) is the name of the parent controller or '' if attached
directly to the PS.
index (int) is the index of the interrupt attached to.

Each entry in the interrupt pin dictionary is a mapping:
'name' -> {controller, index}, where
name (str) is the name of the pin.
controller (str) is the name of the interrupt controller.
index (int) is the line index.

Attributes
----------
```

```
bitfile_name : str
    The absolute path of the bitstream.
dtbo : str
    The absolute path of the dtbo file for the full bitstream.
ip_dict : dict
    All the addressable IPs from PS. Key is the name of the IP; value
is
    a dictionary mapping the physical address, address range, IP type,
    parameters, registers, and the state associated with that IP:
    {str: {'phys_addr' : int, 'addr_range' : int,
'type' : str, 'parameters' : dict, 'registers': dict,
'state' : str}}.
gpio_dict : dict
    All the GPIO pins controlled by PS. Key is the name of the GPIO
pin;
    value is a dictionary mapping user index (starting from 0),
    and the state associated with that GPIO pin:
    {str: {'index' : int, 'state' : str}}.
interrupt_controllers : dict
    All AXI interrupt controllers in the system attached to
    a PS interrupt line. Key is the name of the controller;
    value is a dictionary mapping parent interrupt controller and the
    line index of this interrupt:
    {str: {'parent': str, 'index' : int}}.
    The PS is the root of the hierarchy and is unnamed.
interrupt_pins : dict
    All pins in the design attached to an interrupt controller.
    Key is the name of the pin; value is a dictionary
    mapping the interrupt controller and the line index used:
    {str: {'controller' : str, 'index' : int}}.
pr_dict : dict
    Dictionary mapping from the name of the partial-reconfigurable
    hierarchical blocks to the loaded partial bitstreams:
    {str: {'loaded': str, 'dtbo': str}}.
device : pynq.Device
    The device that the overlay is loaded on
Init docstring:
Return a new Overlay object.

An overlay instantiates a bitstream object as a member initially.

Parameters
----------
bitfile_name : str
    The bitstream name or absolute path as a string.
dtbo : str
    The dtbo file name or absolute path as a string.
download : bool
    Whether the overlay should be downloaded.
```

```
ignore_version : bool
    Indicate whether or not to ignore the driver versions.
device : pynq.Device
    Device on which to load the Overlay. Defaults to
    pynq.Device.active_device
gen_cache: bool
    if true generates a pickeled cache of the metadata

Note
----
This class requires a HWH file to be next to bitstream file
with same name (e.g. `base.bit` and `base.hwh`).
```

Diagram blokowy systemu

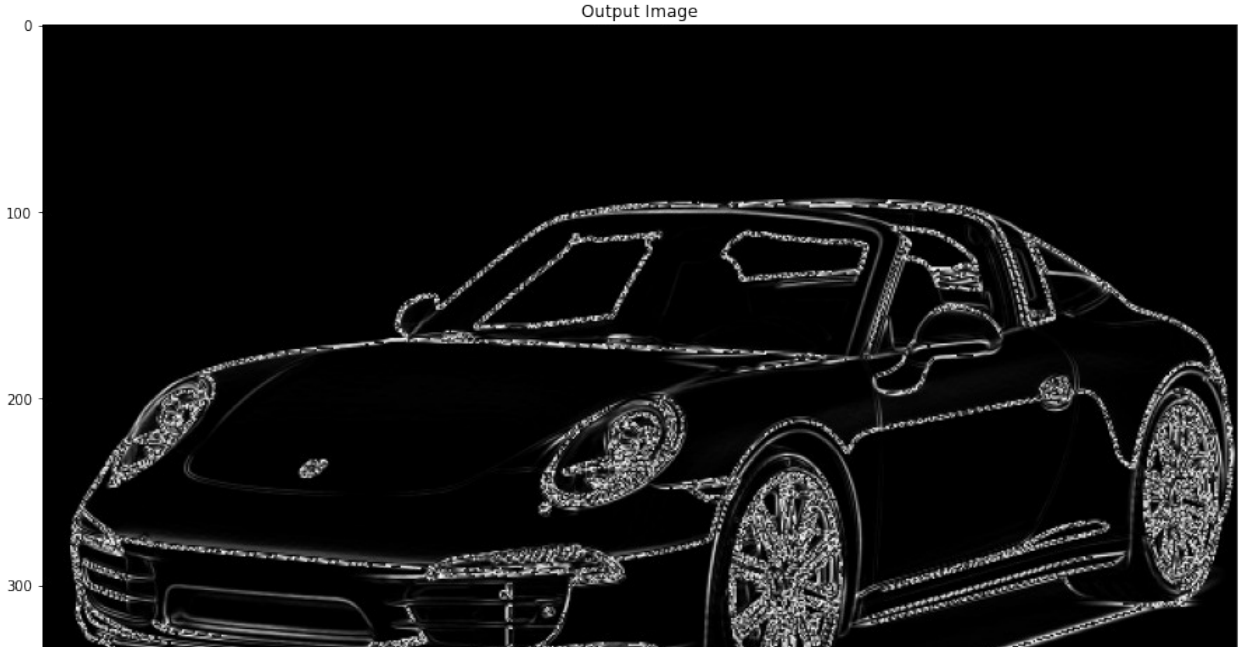



```
dma = sobel_filter_ov.axi_dma_0

in_buffer = allocate(shape=(IMAGE_HEIGHT, IMAGE_WIDTH),
                             dtype=np.uint32, cacheable=1)
out_buffer = allocate(shape=(IMAGE_HEIGHT, IMAGE_WIDTH),
                              dtype=np.uint32, cacheable=1)

data_to_send = convert_png_to_raw(image_file)
in_buffer[:] = np.array(data_to_send)

def run_kernel():
    dma.sendchannel.transfer(in_buffer)
    dma.recvchannel.transfer(out_buffer)
    dma.sendchannel.wait()
    dma.recvchannel.wait()

run_kernel()
print(out_buffer)
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
```

```
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]

img_in = Image.open(image_file)

mode = 'L'  # 'L' - grayscale
tablica_uint8 = out_buffer.astype(np.uint8)
img = Image.fromarray(tablica_uint8, mode=mode)
fig = plt.figure(figsize=(40, 40))

fig.add_subplot(3, 1, 1)
plt.imshow(img_in, cmap=plt.get_cmap('gray'))
plt.title('Input Image')

fig.add_subplot(3, 1, 2)
plt.imshow(img, cmap=plt.get_cmap('gray'))
plt.title('Output Image')

img.save('Images/output_image.png', 'PNG')
```


Input Image
0
100
200
300
400
0
100
200
300
400
500
600

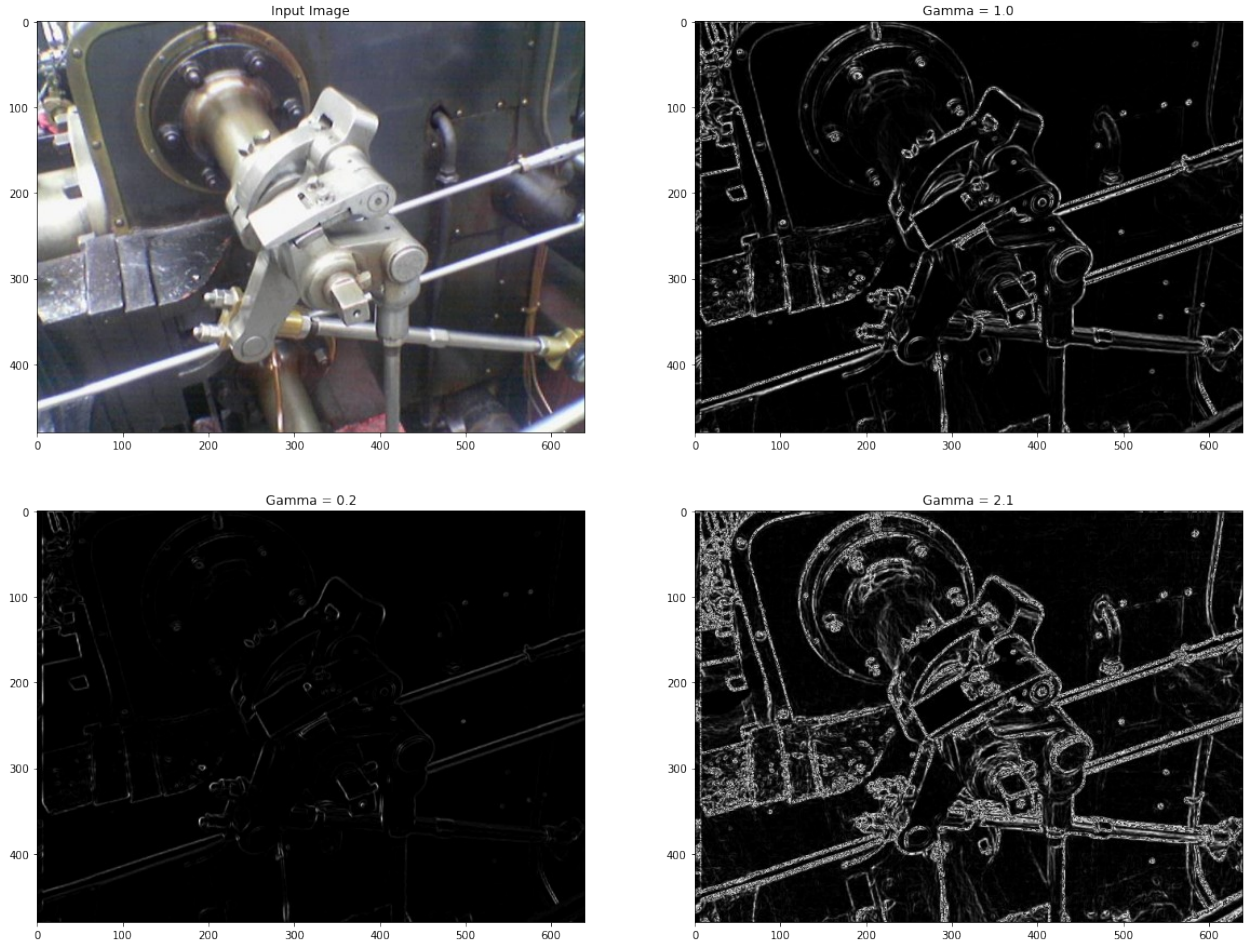

Output Image
0
100
200
300

```
del in_buffer
del out_buffer
```

Wyniki

Porównanie wyników z różnym współczynnikiem gamma

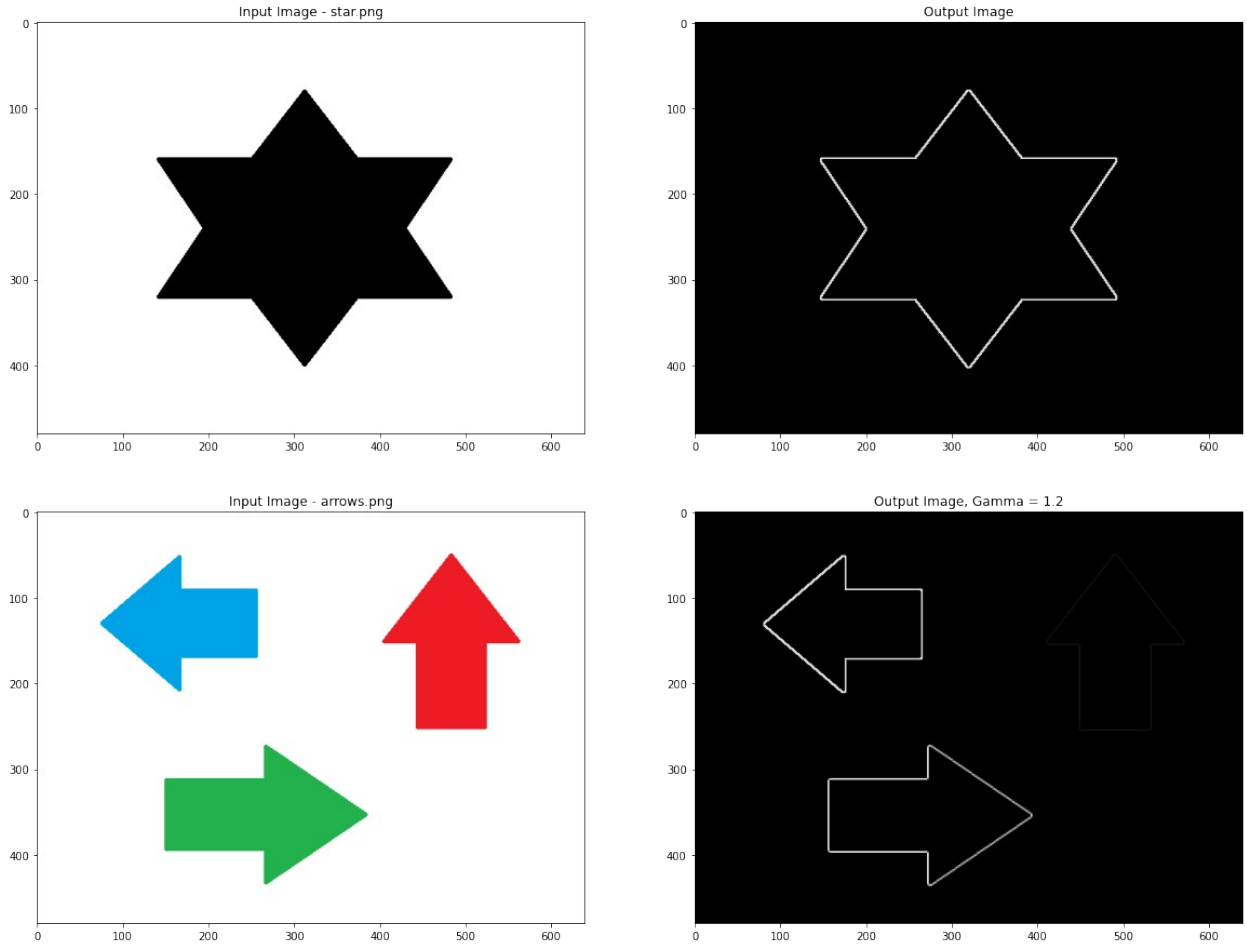
```
img_1 = Image.open('Images/engine.png')
img_2 = Image.open('Images/engine_image_gamma_1.png')
img_3 = Image.open('Images/engine_image_gamma_0_2.png')
img_4 = Image.open('Images/engine_image_gamma_2_1.png')

fig = plt.figure(figsize=(20, 25))

fig.add_subplot(1, 2, 1)
plt.imshow(img_1, cmap=plt.get_cmap('gray'))
plt.title('Input Image')

fig.add_subplot(1, 2, 2)
plt.imshow(img_2, cmap=plt.get_cmap('gray'))
plt.title('Gamma = 1.0')

fig.add_subplot(2, 2, 3)
plt.imshow(img_3, cmap=plt.get_cmap('gray'))
plt.title('Gamma = 0.2')

fig.add_subplot(2, 2, 4)
plt.imshow(img_4, cmap=plt.get_cmap('gray'))
plt.title('Gamma = 2.1')

fig.subplots_adjust(hspace=1)
```



Proste figury geometryczne

```
img_1 = Image.open('Images/star.png')
img_2 = Image.open('Images/star_output.png')
img_3 = Image.open('Images/arrows.png')
img_4 = Image.open('Images/arrows_output.png')

fig = plt.figure(figsize=(20, 25))

fig.add_subplot(1, 2, 1)
plt.imshow(img_1, cmap=plt.get_cmap('gray'))
plt.title('Input Image - star.png')

fig.add_subplot(1, 2, 2)
plt.imshow(img_2, cmap=plt.get_cmap('gray'))
plt.title('Output Image')

fig.add_subplot(2, 2, 3)
plt.imshow(img_3, cmap=plt.get_cmap('gray'))
plt.title('Input Image - arrows.png')
```

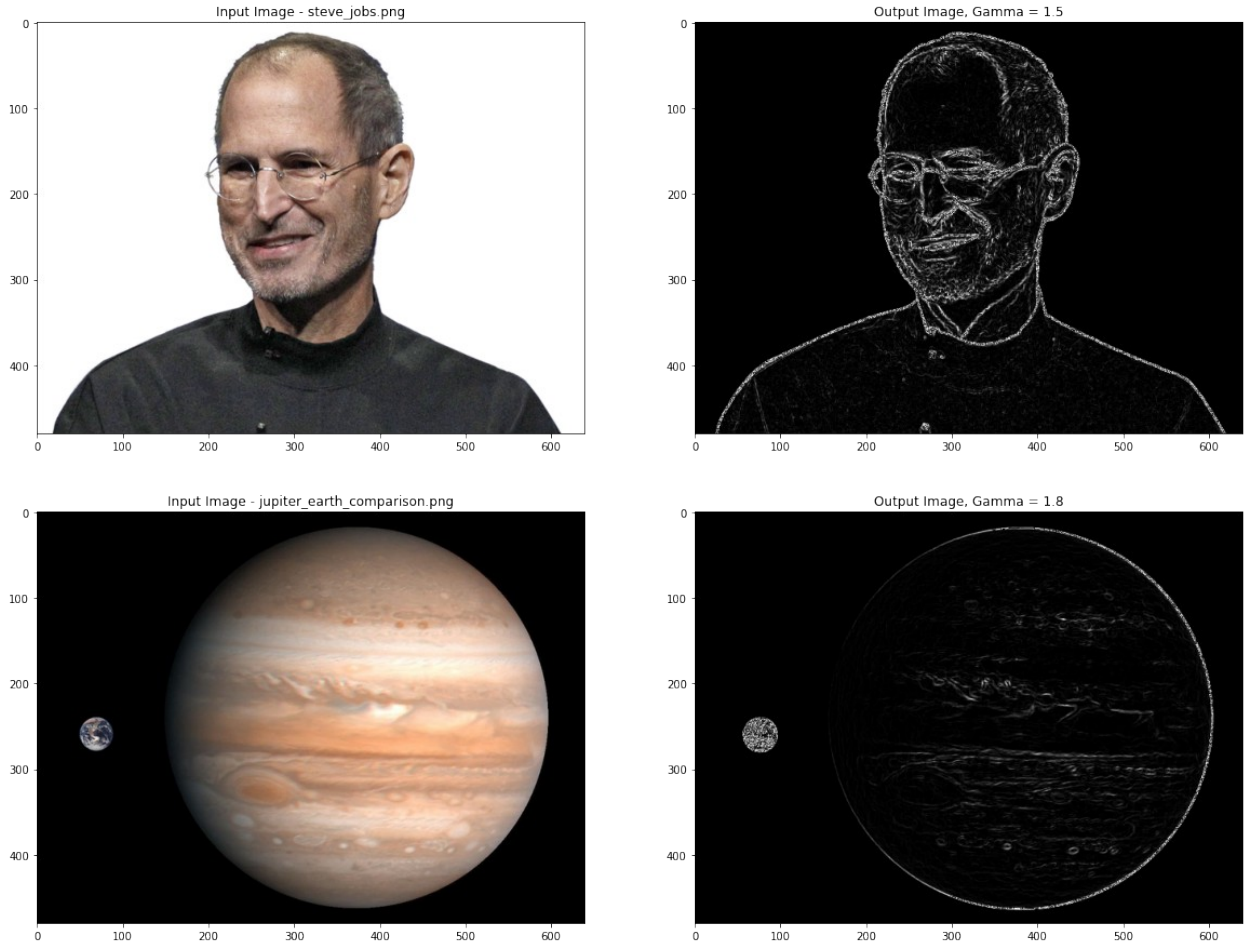
```
fig.add_subplot(2, 2, 4)
plt.imshow(img_4, cmap=plt.get_cmap('gray'))
plt.title('Output Image, Gamma = 1.2')

fig.subplots_adjust(hspace=1)
```



Inne Obrazy

```
img_1 = Image.open('Images/steve_jobs.png')
img_2 = Image.open('Images/steve_jobs_output.png')
img_3 = Image.open('Images/jupiter_earth_comparison.png')
img_4 = Image.open('Images/jupiter_earth_comparison_output.png')

fig = plt.figure(figsize=(20, 25))

fig.add_subplot(1, 2, 1)
plt.imshow(img_1, cmap=plt.get_cmap('gray'))
plt.title('Input Image - steve_jobs.png')

fig.add_subplot(1, 2, 2)
plt.imshow(img_2, cmap=plt.get_cmap('gray'))
```

```
plt.title('Output Image, Gamma = 1.5')

fig.add_subplot(2, 2, 3)
plt.imshow(img_3, cmap=plt.get_cmap('gray'))
plt.title('Input Image - jupiter_earth_comparison.png')

fig.add_subplot(2, 2, 4)
plt.imshow(img_4, cmap=plt.get_cmap('gray'))
plt.title('Output Image, Gamma = 1.8')

fig.subplots_adjust(hspace=1)
```



Bibliografia

https://upel.agh.edu.pl/pluginfile.php/348698/mod_folder/content/0/T8_pynq_ver0_2_1.pdf
https://en.wikipedia.org/wiki/Sobel_operator
https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm
https://docs.amd.com/v/u/2018.3-English/ug901-vivado-synthesis