Metody Rozpoznawania Obrazów
Dominik Szot
Zadanie 01
Życie na krawędzi

W ramach tego laboratorium należało zaimplementować uproszczony mechanizm wykrywania krawędzi metodą Canny'ego.

1. Wczytanie obrazu

Pierwszym krokiem wykonania zadania będzie wstępne przygotowanie zdjęcia. Do wykonania tego zadania będę używać frameworku pytorch, natomiast do obsługiwania zdjęć biblioteki pillow. Wybrane zdjęcie importujemy do obiektu Image. Następnie dane zdjęcie konwertujemy na tensor za pomocą funkcji bibliotecznej.

Obraz 1: Zdjęcie wybrane do przeprowadzenia wykrywania krawędzi (1481 × 864 pixels)

2. Zamiana na skalę szarości

Podane zdjęcie upraszczamy zamieniając trzy kanały odpowiadające kolorom – czerwonym, zielonym I niebieskim, na jeden kanał korzystający z skali szarości. W tym celu stosujemy konwolucję 1x1 z jednym kanałem wejściowym.

```
def apply_conv3d(input: torch.Tensor, weight: torch.Tensor, **kwargs) -> torch.Tensor:
        return F.conv3d(input,
                weight=weight,
                bias=None,
                stride=1,
                padding=0
)

# https://en.wikipedia.org/wiki/Rec._709#Luma_coefficients
def apply_grayscale(
        image: torch.Tensor,
        weight: torch.Tensor = torch.tensor([[[[[0.2126]], [[0.7152]], [[0.0722]]]]]),
        **kwargs) -> torch.Tensor:

        image_with_grascale = apply_conv3d(image.unsqueeze(0), weight)
        return image_with_grascale

grayscale_image = apply_grayscale(image_tensor)
show_tensor(grayscale_image)
```

Jako wagi podane zostały współczynniki Luma[1] wynoszące dla odpowiednio kanałów (R,G,B): (0.2126, 0.7152, 0.0722). Współczynniki te odzwierciedlają jak ludzkie oko postrzega jasność każdego koloru.

1 Opis stosowanych współczynników: https://en.wikipedia.org/wiki/Rec._709#Luma_coefficients

Obraz 2: Zdjęcie konwertowane do skali szarości

3. Skalowanie obrazu

W celu zmniejszenia rozmiaru obrazu przeprowadzono operację poolingu. Wybrano **max pooling** który posiada tą zaletę że wydobywa najbardziej wyróżniające się cechy obrazu, w przeciwieństwie do **average pooling** które obraz bardziej wygładza. Do operacji wykrywania krawędzi average pooling będzie bardziej przeszkadzać rozmywając krawędzie a przez to zmniejszy precyzję ich wykrywania w późniejszych etapach. Jako rozmiar poolingu wybrano standardowy rozmiar 2x2.

```
def apply_pool2d(input: torch.Tensor, kernel_size: tuple, **kargs) -> torch.Tensor:
        return F.max_pool2d(input,
                kernel_size = kernel_size
)

def scale_down_image(image: torch.Tensor, kernel_size: tuple=(4,4)):
        return apply_pool2d(grayscale_image, kernel_size = kernel_size)

scaled_down_image = scale_down_image(grayscale_image, (2,2))
show_tensor(scaled_down_image)
```

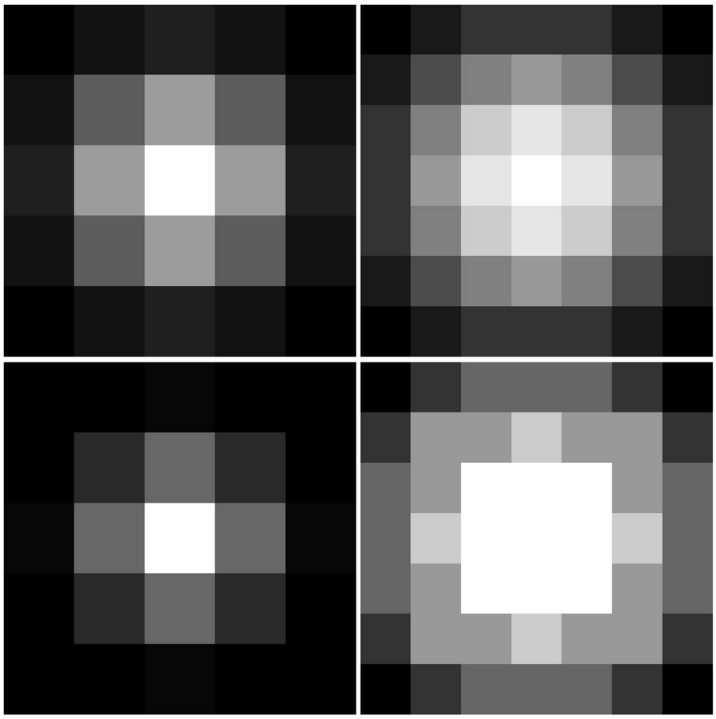
Obraz 3: Zdjęcie po operacji poolingu

4. Pozbywanie się niepotrzebnych szumów I detali

W tym celu zastosowano rozmycie Gaussowskie. Pierwszym etapem będzie przygotowanie oraz wybranie odpowiedniej macierzy z odpowiednimi współczynnikami, takimi że największe współczynniki będą najbliżej środka, natomiast im dalej tym współczynniki będą maleć.

```
def gaussian_kernel(size: int = 5, sigma: int = 2) -> torch.Tensor:
        size = int(size) // 2
        x, y = np.mgrid[-size:size+1, -size:size+1]
        normal = 1 / (2.0 * np.pi * sigma**2)
        g = np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
        return torch.Tensor(g / g.sum())

def check_gaussian_kernel(weights: np.array) -> None:
        assert torch.isclose(weights.sum(), torch.tensor(1.0)), f"Sum is not 1, it's {weights.sum()}"
```

W kolejnym etapie wybrano kilka rozmiarów tablicy z różnymi parametrami funkcji tworzącej rozkład.



Obraz 4: Różne rozmiary tablic z różnymi współczynnikami: od lewej (5, 1), (7, 2), (4, 0.75), (6, 3).

Następnie z użyciem każdego kernela wykonano konwolucję dając efekt rozmycia.

```
def apply_conv2d(input: torch.Tensor, weight: torch.Tensor, **kwargs) -> torch.Tensor:
        assert len(input.shape) == len(weight.shape), f'Input and weights must have same dimensions
        {len(input.shape)} vs {len(weight.shape)}'

        return F.conv2d(input,
                weight=weight,
                bias=torch.Tensor(weight.shape[0]),
                **kwargs
        )

def apply_gaussian_filter(
        input: torch.Tensor,
        weight: torch.Tensor,
        **kwargs) -> torch.Tensor:

        image_with_gaussian_filter = apply_conv2d(input, weight)
        return image_with_gaussian_filter
```

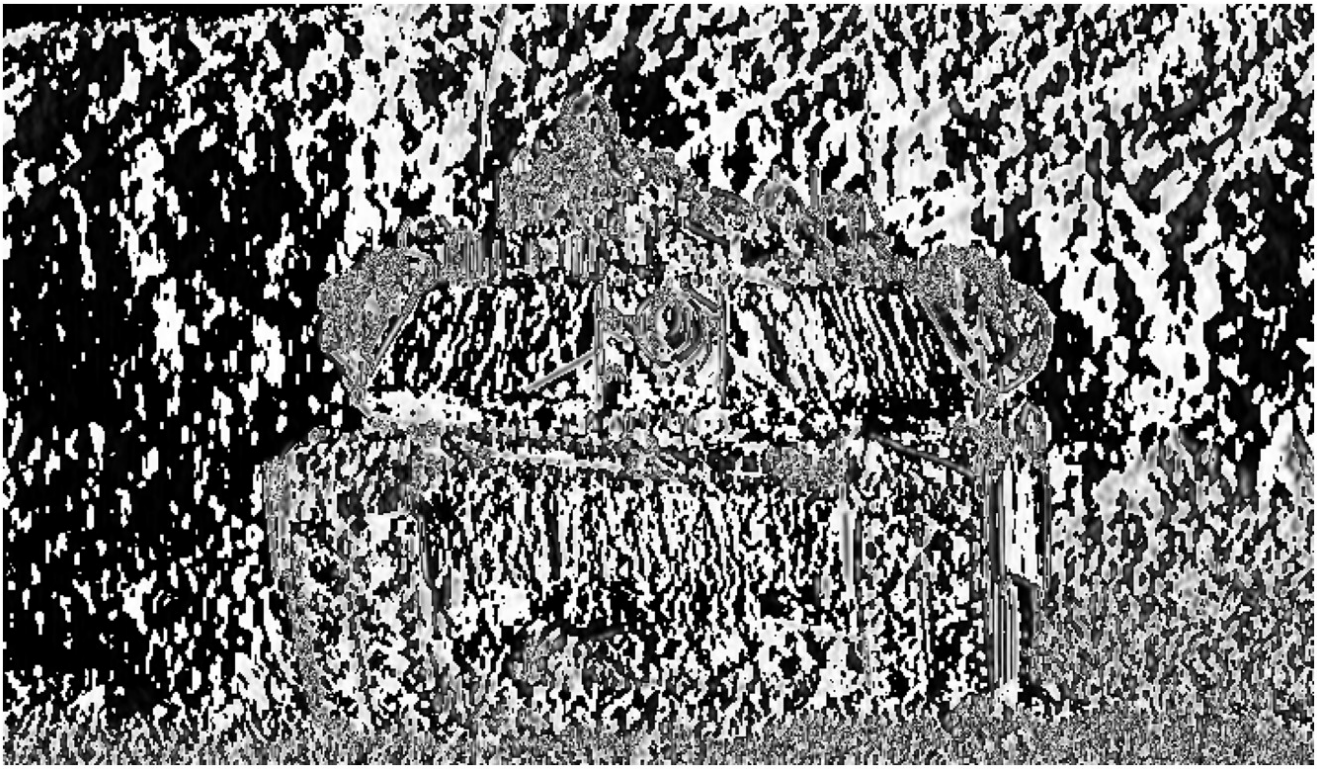
Obraz 5: Zdjęcia powstałe z użyciem różnych kerneli gaussowskich

Zdjęcie 3 powstałe z użycia kernela 5x5, z współczynnikiem sigma = 0.75 zostało wybrane do kolejnych etapów.

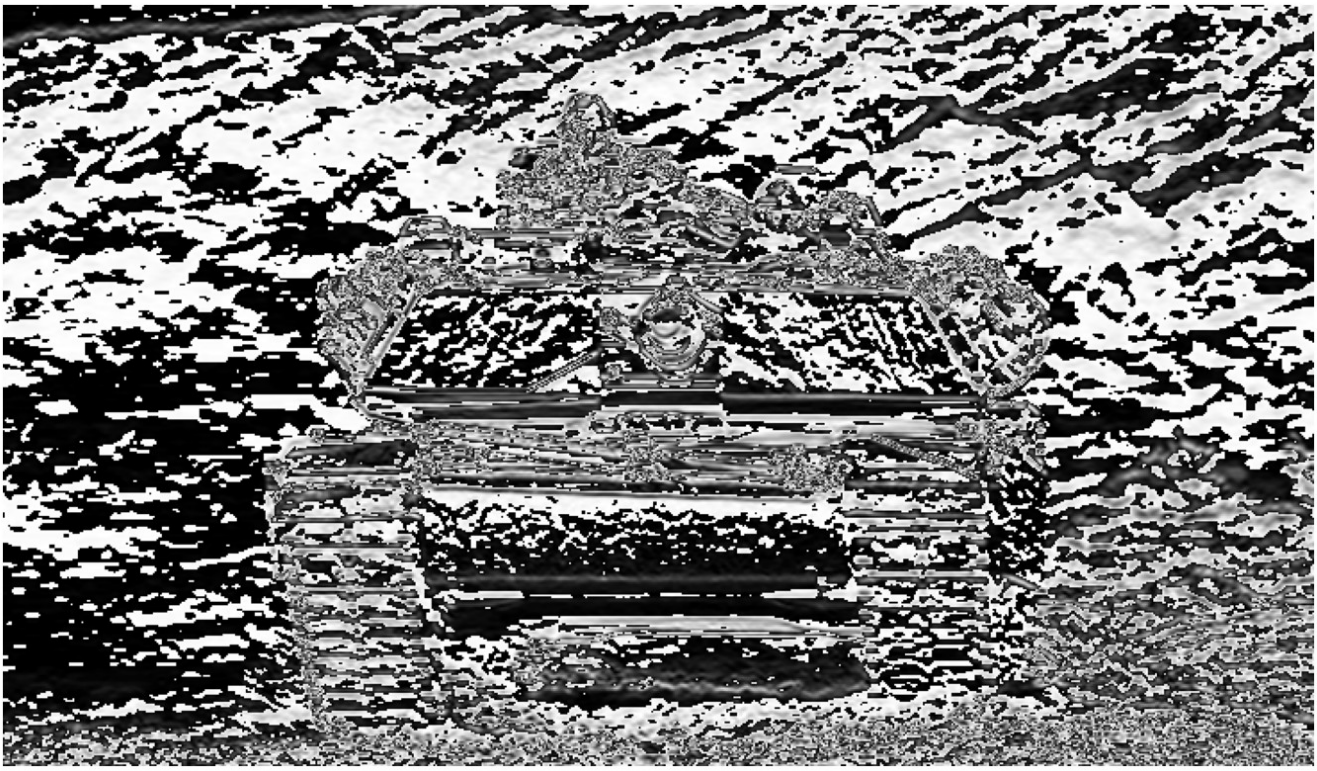
5. Gradient intensywności

W celu poznania poziomą I pionową składową gradientu przetwarzam wybrane zdjęcie przez filtry Sobela wykrywające krawędzie pionowe I poziome.

```
def sobel_filters(image: torch.Tensor):
        Kx = torch.tensor([[[-1, 0, 1], [-2, 0, 2],[-1,0,1]]], dtype=torch.float32)
        Ky = torch.tensor([[[1, 2, 1], [0, 0, 0],[-1, -2,-1]]], dtype=torch.float32)

        Ix = apply_conv2d(image, Kx.unsqueeze(0))
        Iy = apply_conv2d(image, Ky.unsqueeze(0))

        G = np.hypot(Ix, Iy)
        theta = np.arctan2(Iy, Ix)

        return (G, theta)
```



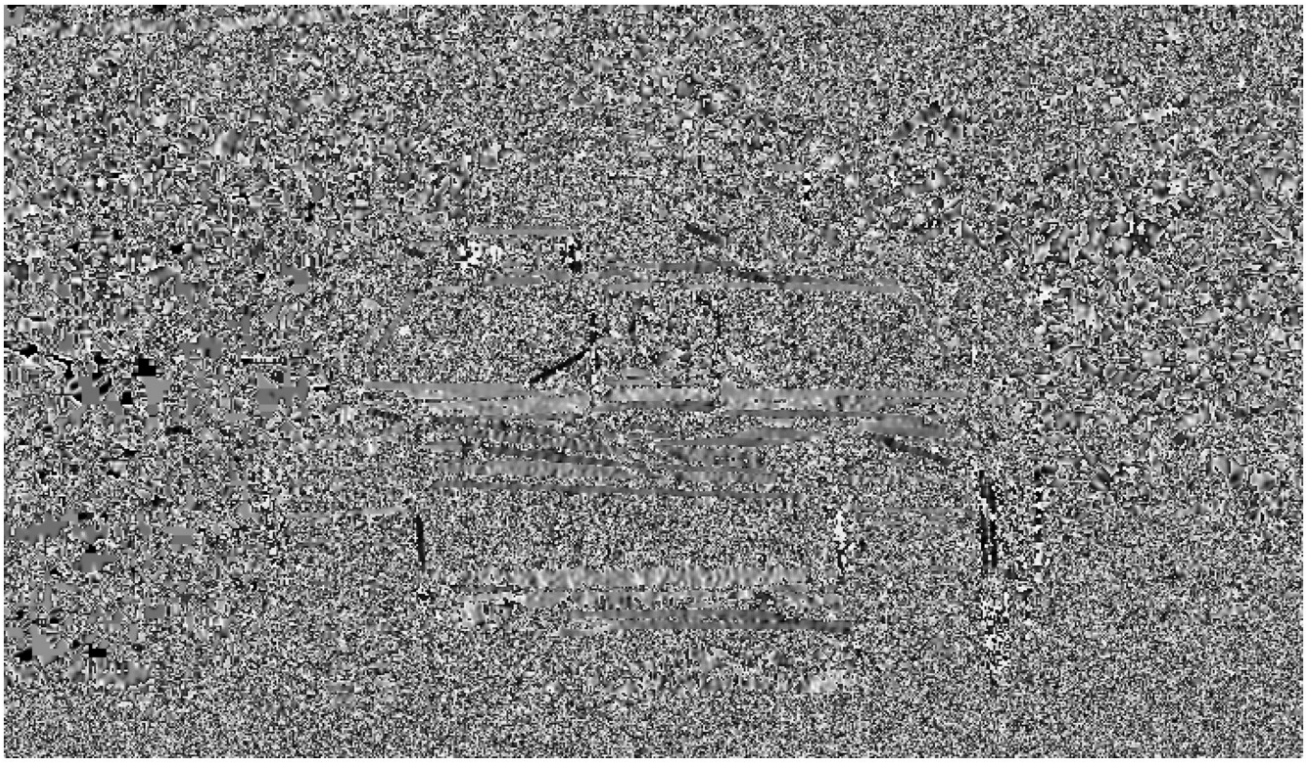
Obraz 6: Obraz powstały przez nałożenie filtra Sobela wykrywającego krawiędzie poziome



Obraz 7: Obraz powstały przez nałożenie filtra Sobela wykrywającego krawiędzie pionowe

Powstałe kanały scalamy w jeden reprezentujący intensywność gradientu. Dodatkowo tworzymy kanał który będzie przechowywać informacje o kierunku gradientu.

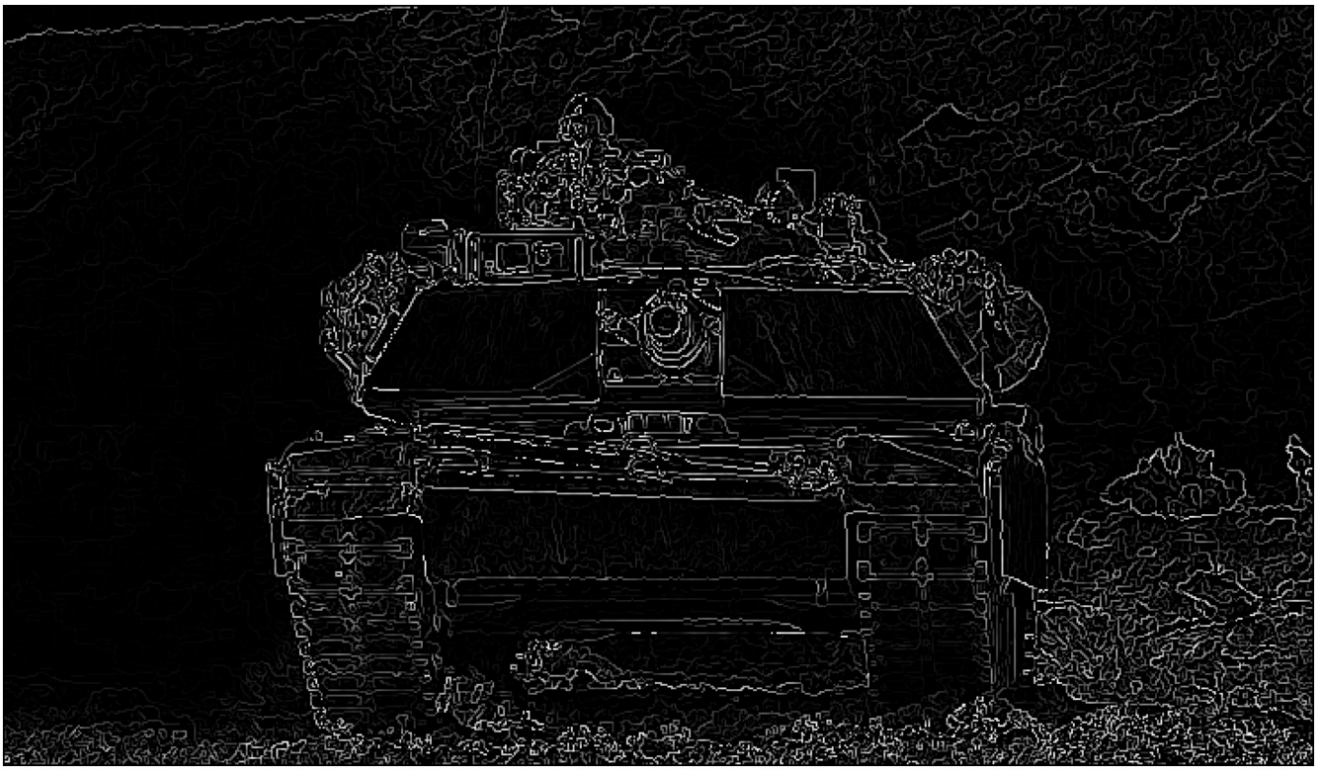
Obraz 8: Obraz przedstawiający scalone kanały wykrywające krawędzie



Obraz 9: Obraz przechowujacy informacje o kierunku gradientu

6. Non-Maximum Suppression

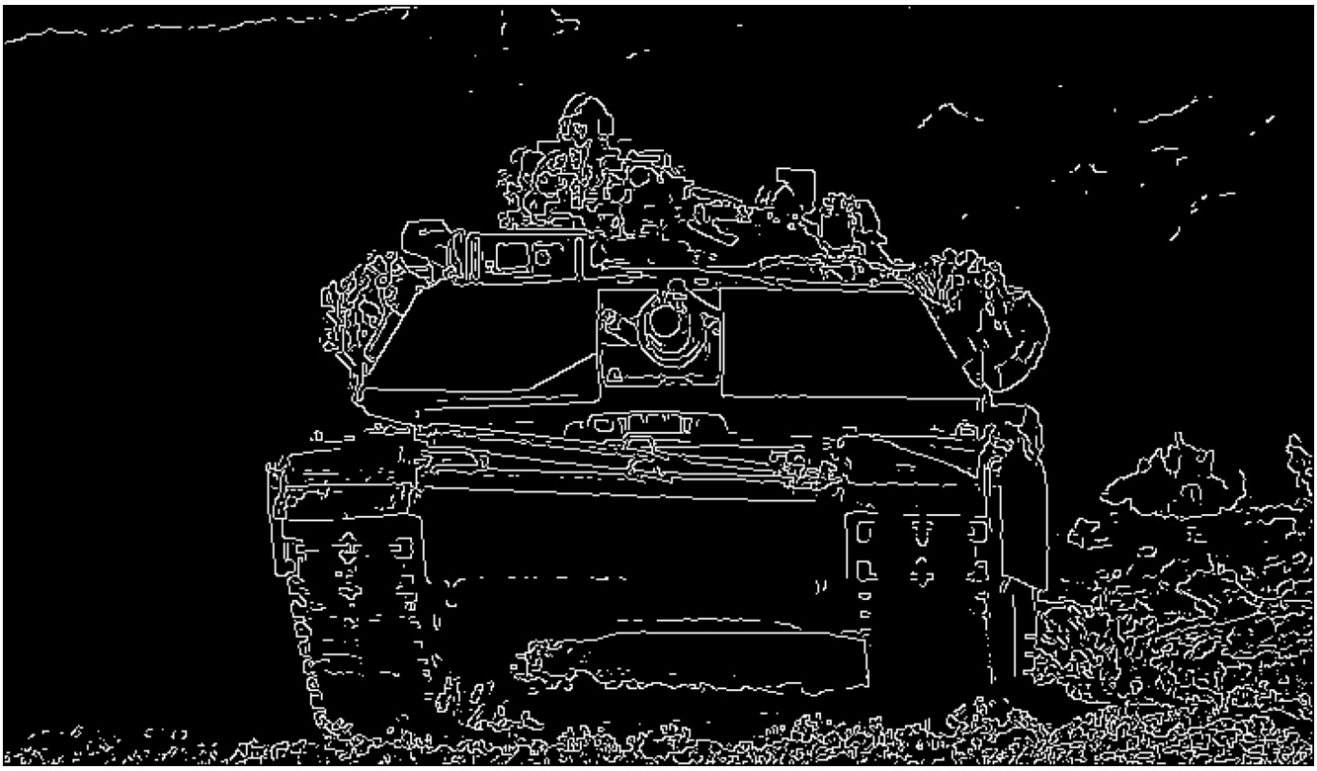
W celu wyselekcjonowania najbardziej znaczących obiektów (w tym przypadku najbardziej znaczących wartości krawędzi) używamy Non-Maximum Suppression. Posiadamy już ampitudy oraz krawędzie więc musimy sprawdzić czy dana krawędź posiada inną, prostopadłą do niej krawędź o wyższej ampitudzie, Jeśli tak, eliminujemy aktualnie analizowaną krawędź; jeśli nie, uznajemy ją za lokalne maksimum i zachowujemy w ostatecznym obrazie krawędzi.



Obraz 10: Wynik działania Non-Maximum Suppression

7. Filtrowanie regionów z niewielkim gradientem

W tym kroku używamy pojedyńczego progowania, które dla danego progu pozostawi krawędź, a krawędzie poniżej progu zostaną usunięte. Odbiegnięcie od **wzorcowej** metody Canny'ego może posiadać kilka różnych, negatywnych konsekwencji, takie jak brak eliminacji krawędzi zbyt słabych co doprowadzi to artefaktów oraz szumów na końcowym obrazie.



Obraz 11: Obraz z pojedyńczym progowaniem krawędzi

8.Scalanie uzyskanych wyników z pierwotnym obrazem.

W tym punkcie wystarczy dokonać upscalingu uzyskanego wyniku oraz nałożyć obraz na nasz początkowy. Jak można zauważyć algorytm poradził sobie całkiem dobrze wykrywając główny obiekt na zdjęciu (czołg). Widać też pewne artefakty w postaci mało znaczących krawędzi tła oraz trawy która nie została dostatecznie dobrze rozmyta.

Obraz 12: Końcowy wynik algorytmu wykrywania krawędzi