Laboratorium 1.

Laboratorium wykonałem korzystając z frameworku `tinygrad`[1]. Poniżej zamieściłem wykorzystany obraz o rozdzielczości 2560 x 1426 pikseli.

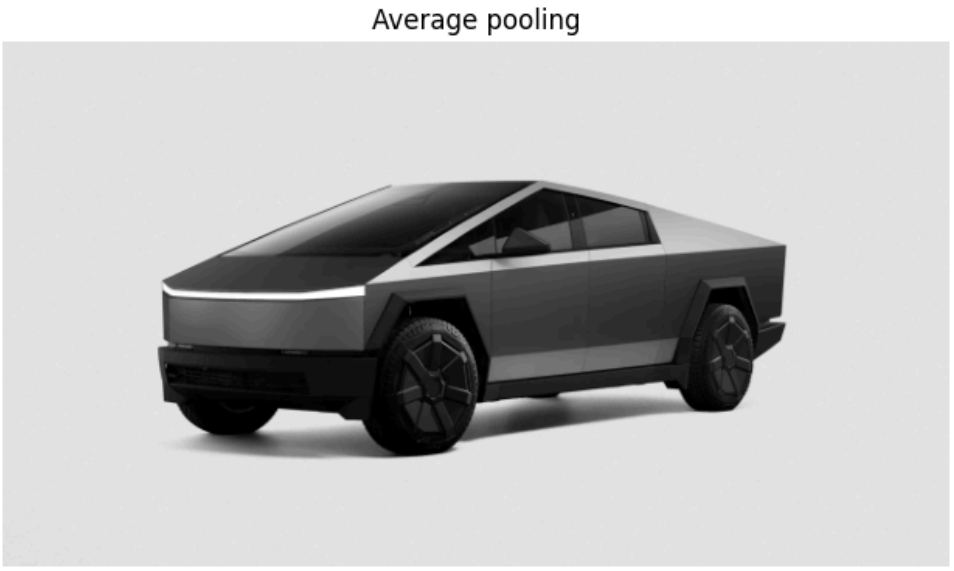
Po wczytaniu obrazu i przeskalowaniu wartości do przedziału [0;1], przekształciłem obraz do skali szarości korzystając z konwolucji. Jako współczynników w filtrze konwolucyjnym użyłem wartości ze wzoru NTSC[2].

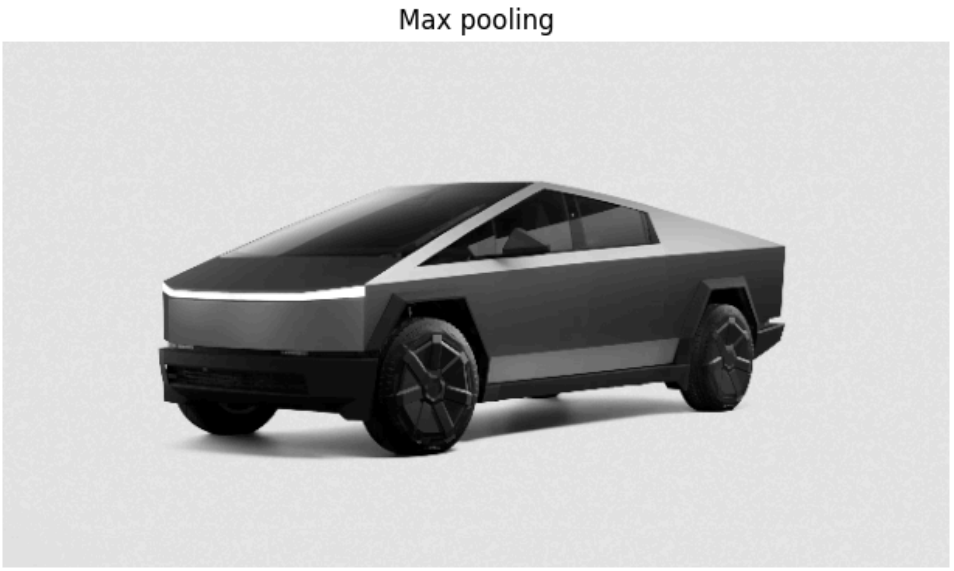
```
# Convert image to greyscale using a 1x3x1x1 convolution map. We use the NTSC formula
# Y = 0.299*r + 0.587*g + 0.114*b
r_coef, g_coef, b_coef = 0.299, 0.587, 0.114
filter_gray = Tensor([[[r_coef]], [[g_coef]], [[b_coef]]]).unsqueeze(0)
img = img.conv2d(filter_gray, padding=0, stride=1)

show(img)
```

[1] https://github.com/tinygrad/tinygrad (warto zostawić gwiazdkę 🙂)
[2] https://en.wikipedia.org/wiki/Grayscale

Następnie zmniejszyłem obraz korzystając z operacji poolingu. Na podstawie wyników dla max pooling i average pooling przedstawionych poniżej wybrałem ostatecznie average pooling, gdyż obraz był bardziej ostry (ze względu na dużą rozdzielczość źródłowego pliku różnica jest słabo dostrzegalna na poniższych, pomniejszonych obrazach).

```
# We use pooling to slightly decrease the image size
img_avgpool = img.avg_pool2d(kernel_size=(4, 4))
img_maxpool = img.max_pool2d(kernel_size=(4, 4))

show(img_avgpool, title="Average pooling")

show(img_maxpool, title="Max pooling")

# Based on the results we choose average pooling as it makes the image less pixelated
img = img_avgpool
```





Aby pozbyć się szumu i niepotrzebnych detali użyłem rozmycia gaussowskiego. W tym celu wykorzystałem wzór na nie znormalizowaną gęstość prawdopodobieństwa dla dwuwymiarowego rozkładu normalnego i wygenerowałem macierz rozmiaru *n* x *n*, której wartości były równe wartościom tej gęstości dla kolejnych indeksów macierzy. Zamieszczona poniżej funkcja generuje odpowiedni filtr konwolucyjny dla zadanego rozmiaru filtra *n*.

```
def filter_gauss(n: int) -> Tensor:
    phi = lambda mx, my, s: lambda x, y: Tensor.exp(-1/(2*s**2)*((x - mx)**2 + (y - my)**2))
    x = Tensor.arange(n).repeat(n, 1)
    y = x.T
    f = phi(mx=(n - 1)/2, my=(n - 1)/2, s=n/4)(x, y)
    f /= f.sum()
    return f
```

Dla wartości *n* = 3, 5, 7 wygenerowałem rozmyty obraz poprzez konwolucję odpowiedniego filtra z obrazem i na podstawie uzyskanych wyników postanowiłem użyć wartości *n* = 3, gdyż obraz był wówczas estetycznie rozmyty.

```
for n in (3, 5, 7):
    show(img.conv2d(filter_gauss(n).reshape(1, 1, n, n), stride=1, padding=n // 2))

# Based on the results above we choose a 3x3 gaussian kernel for blurring
n = 3
img = img.conv2d(filter_gauss(n).reshape(1, 1, n, n), stride=1, padding=n // 2)
```

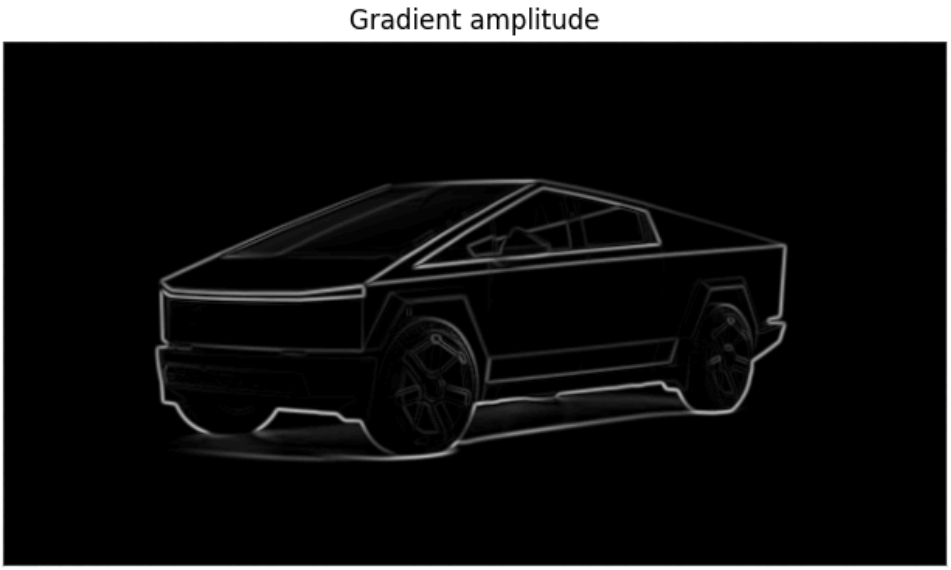
Gaussian blur n=5

Gaussian blur n=7

Następnym wykonanym krokiem było wykorzystanie pary filtrów Sobela do wyciągnięcia informacji o gradiencie wartości pikseli na obrazie – jego wartości (amplitudzie) i kierunkowi. Nie obliczałem bezpośrednio kąta korzystając z funkcji *arctan* z dwóch powodów – prozaicznym powodem był brak funkcji Tensor.arctan() w używanym frameworku, drugim natomiast fakt, iż jest to zupełnie niepotrzebny wydatek obliczeniowy jak zobaczymy za chwilę.

```
# We now apply the Sobel filters Kx, Ky and compute gradient information -- amplitude and
# tangent

Kx = Tensor([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype="float").reshape(1, 1, 3, 3)
Ky = Tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype="float").reshape(1, 1, 3, 3)

Ix = img.conv2d(Kx, stride=1, padding=1)
Iy = img.conv2d(Ky, stride=1, padding=1)

G_amp = Tensor.sqrt(Ix**2 + Iy**2)
G_tan = Iy / Ix
```

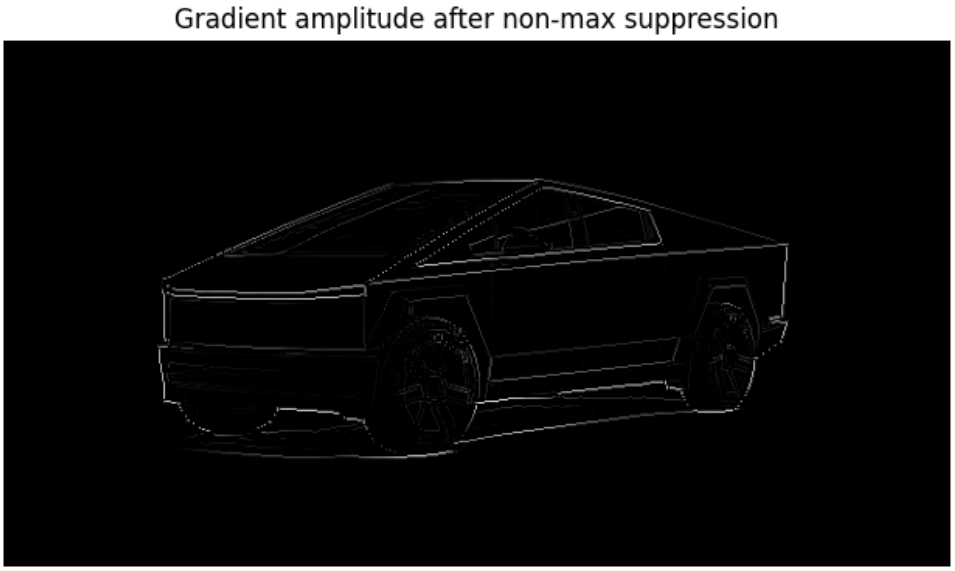
Uzyskany obraz przedstawiający wartości gradientu zamieściłem poniżej.



Uzyskany efekt jest już całkiem niezły, ale krawędzie powinny być jak najcieńsze. W tym celu wykonałem non-max suppression polegający na pozostawieniu jedynie tych wartości gradientu, dla których jest to lokalne maksimum w obrębie sąsiadów wyznaczonych przez kierunek gradientu. Zamieszczona poniżej implementacja nie zawiera nieefektywnej podwójnej pętli w czystym pythonie oraz pokazuje, iż wystarczy obliczyć jedynie dwie wartości funkcji tangens, aby wykonać non-max suppression (które na marginesie można wyrazić w zwartej formie korzystając jedynie z pierwiastka kwadratowego).

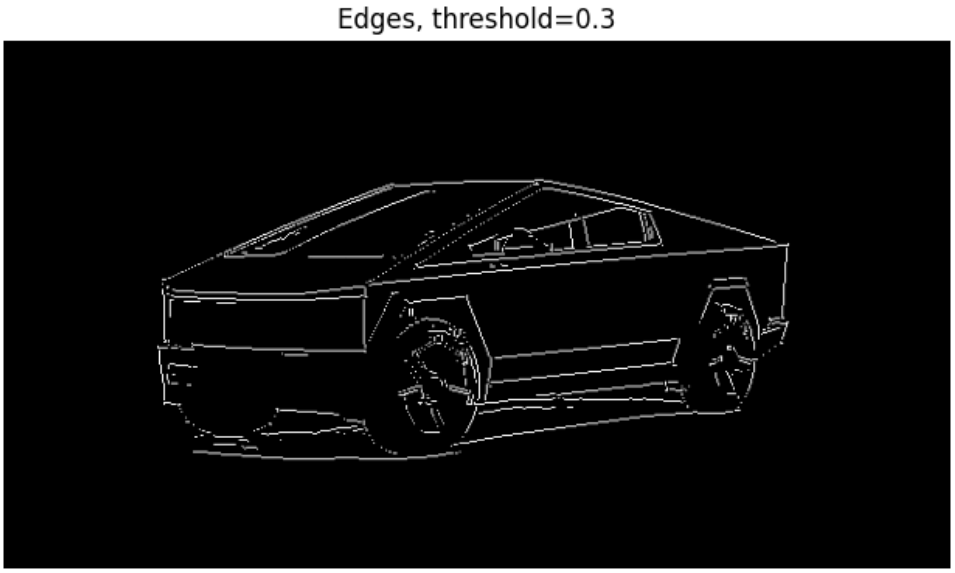
```
from math import tan, pi

# First we compute masks for every direction (E-W, N-S, NE-SW, NW-SE). Here we also note that
# directly computing the angle using arctan in the previous step is inefficient.
tan22_5 = tan(1 * pi / 8)
tan67_5 = tan(3 * pi / 8)

mask_00 = ((G_tan < +tan22_5) * (G_tan > -tan22_5)).flatten()
mask_90 = ((G_tan < -tan67_5) + (G_tan > +tan67_5)).flatten()
mask_p45 = ((G_tan >= +tan22_5) * (G_tan <= +tan67_5)).flatten()
mask_n45 = ((G_tan <= -tan22_5) * (G_tan >= -tan67_5)).flatten()

# We compute strides which determine the location of neighbors according to gradient
# direction. Note here that we use a flattened representation as it makes indexing easier.
*_, H, W = G_amp.shape
strides = 1 * mask_00 + W * mask_90 + (W + 1) * mask_p45 + (W - 1) * mask_n45

G_amp_flat = G_amp.flatten()
idxs = Tensor.arange(W * H)
mask = (G_amp_flat >= G_amp_flat[idxs + strides]) * (G_amp_flat >= G_amp_flat[idxs - strides])

# Now we only need to take care of border artifacts
mask = mask.reshape(1, 1, H, W)
mask = mask[..., 1 : H - 1, 1 : W - 1]
mask = mask.pad2d((1, 1, 1, 1))

G_amp *= mask
```

Uzyskane efekt zamieściłem na poniższym obrazie.

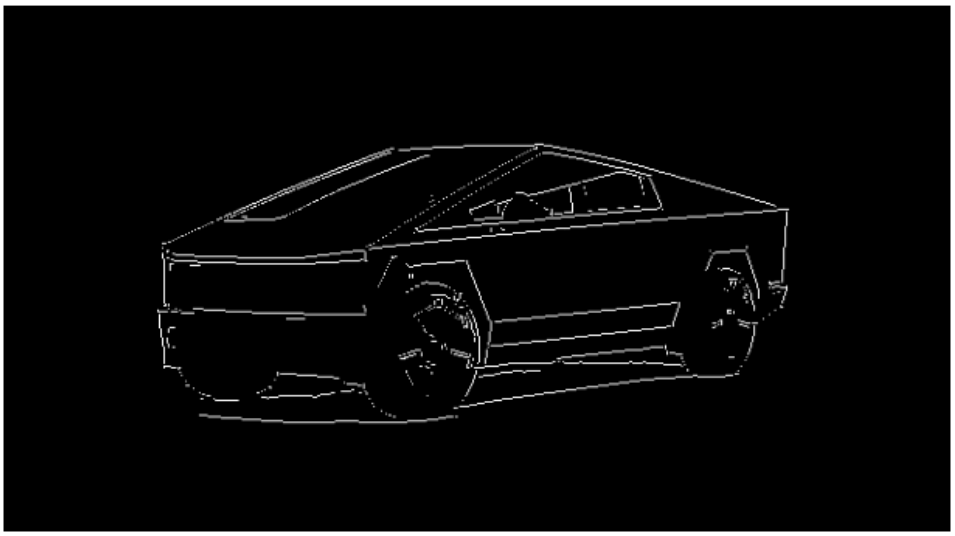


Kolejnym krokiem było odfiltrowanie regionów o niewielkiej wartości gradientu. Ponieważ ostatecznie wartości mają być równe 0 lub 1, więc progowane ReLU tak naprawdę nie jest potrzebne i wystarczy sprawdzić, czy wartość gradientu jest większa od ustalonego progu.

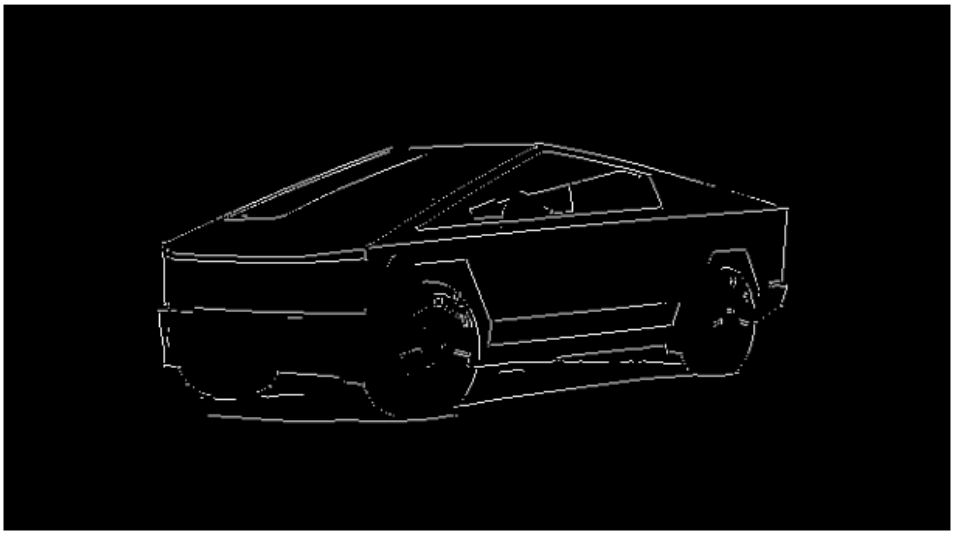
```
for threshold in (0.3, 0.4, 0.5, 0.6):
    show((G_amp - threshold) > 0, title=f"Edges, threshold={threshold}")

# Based on the results above we choose a threshold of 0.4
threshold = 0.4
G_amp_threshold = (G_amp - threshold) > 0
```



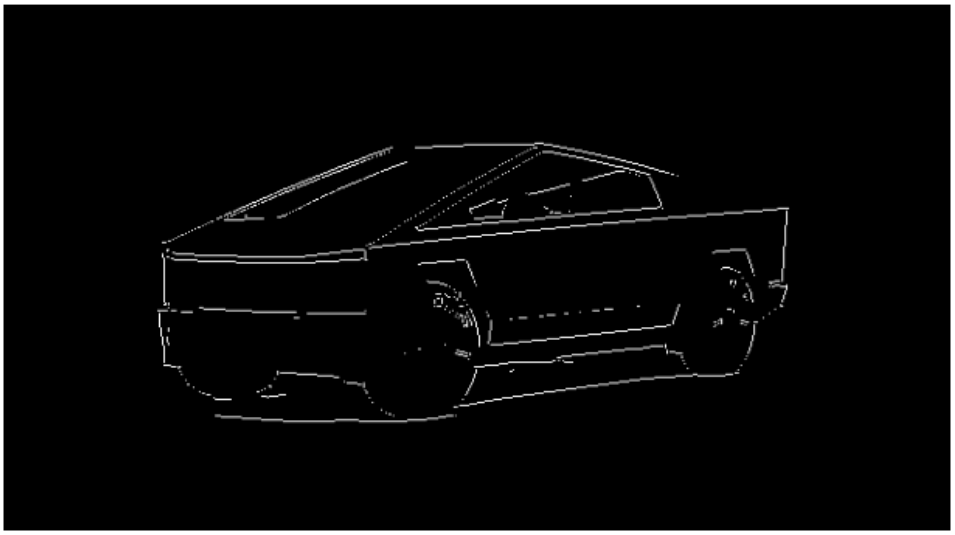
Edges, threshold=0.4



Edges, threshold=0.5



Edges, threshold=0.6



Na podstawie uzyskanych wyników wybrałem wartość progu równą 0.4 jako dobry balans między ilością szczegółów, a czytelnością krawędzi. Ponieważ nie używamy tutaj double thresholding, więc zapewne tracimy część pikseli będących elementem jakiejś krawędzi, które jednak nie są silne (strong). We wzorcowym algorytmie Canny'ego część słabych (weak) pikseli zostałaby włączona do ostatecznego obrazu.

Ostatnim krokiem był upscaling obrazu zawierającego same krawędzie i nałożenie go na oryginalny obraz. Jako metodę upscalingu zastosowałem transponowaną konwolucję z filtrem składającym się z samych jedynek i odpowiednimi parametrami, aby wynik miał odpowiedni kształt. Końcowy rezultat jest bardzo sensowny.

```
edges = G_amp_threshold
edges = edges.conv_transpose2d(Tensor.ones(1, 1, 3, 3), stride=4, output_padding=(3, 1))

amplification = 0.8
r_channel = (img_og[:, 0, ...] - amplification * edges[:, 0, ...]).clip(0, 1)
g_channel = (img_og[:, 1, ...] + amplification * edges[:, 0, ...]).clip(0, 1)
b_channel = (img_og[:, 2, ...] - amplification * edges[:, 0, ...]).clip(0, 1)

img_with_edges = Tensor.stack(r_channel, g_channel, b_channel, dim=1)

show(img_with_edges)
```

Dodatek

```
def show(img: Tensor, title=None):
    plt.imshow(img[0, ...].permute(1, 2, 0).numpy(), cmap="grey")
    plt.axis("off")
    plt.title(title) if title is not None else ...
    plt.tight_layout()
    plt.show()
```