Practicum: Kleurpaletten

Multimedia - Multimediatechnieken

16 februari 2021

1 Introductie

Kleurpaletten of Color Lookup Tables (LUTs) worden gebruikt om kleurrijke afbeeldingen weer te geven met een kleiner aantal kleuren. Dit was vroeger zeer belangrijk toen beeldschermen of browsers maar een bepaald aantal kleuren konden weergeven. Tegenwoordig is dit minder een probleem, maar het is nog steeds gebruikelijk in het coderen van bvb. GIF bestanden en hardwarematig coderen van texturen in computer graphics (bvb. DXT1).

In dit practicum zullen we een GIF encoder implementeren. Bij afbeeldings- en videoformaten is het gebruikelijk dat enkel het formaat van de bitstroom gestandaardiseerd is. De exacte implementatie van de encoder wordt dus open gelaten, zolang deze resulteert in een bitstroom die voldoet aan de specificaties en dus gedecodeerd kan worden. Hierdoor is er een bepaalde vrijheid tijdens het encoderen waardoor de ene encoder beter kan presteren dan een andere encoder. Met beter presteren bedoelen we dat we een afbeelding met minder bits kunnen opslaan met dezelfde afbeeldingskwaliteit. In dit practicum zullen we tonen hoe we stelselmatig de GIF encoder kunnen verbeteren, startende van een minimale basisversie.

De GIF-encoder bevat twee stappen: (1) het opstellen van een kleurpalet en (2) het coderen van de LUT-indices voor iedere pixel. De tweede stap wordt gedaan aan de hand van Lempel–Ziv–Welch (LZW) compressie. LZW is een verliesloze compressiemethode die meerdere symbolen (hier dus LUT-indices) samen codeert als een bitsequentie. De bedoeling is dat vaak voorkomende symboolsequenties gerepresenteerd wordt door een kleiner aantal bits.

Belangrijk: De verbetering is grotendeels geautomatiseerd, dus lees aandachtig de instructies (omtrent bestandsnamen en andere conventies) zodat je een correct practicum maakt.

Jullie zullen de finale versie van de Python code indienen, samen met de antwoorden op de afzonderlijke tekstuele vragen. Elke tekstueel antwoord komt in een apart bestand met vooraf bepaalde bestandsnaam. Deze bestanden zijn platte tekst (plain text), en kunnen met eender welke platte tekst verwerker gemaakt worden. Kies zelf uit bijvoorbeeld: vim, Sublime Text, Notepad++, Atom, emacs, ... Het is toegestaan Markdown te gebruiken.

Hoewel de verbetering deels geautomatiseerd is, zal de ingediende code naast functionele correctheid ook **gequoteerd** worden **op stijl en snelheid**. Streef dus naar robuuste, kwalitatieve, en snelle code.

2 Opgave

2.1 Bibliotheken

Gelieve de volgende Python 3 bibliotheken te installeren indien dit nog niet het geval is (`conda install` als je conda gebruikt, of `pip3 install` indien je niet met conda werkt):

- `numpy` : Proceduraal programmeren van met arrays.
- `scikit-learn` : Machine learning algoritmen, waaruit we k-means clustering zullen gebruiken.
- `Pillow` : Bibliotheek voor het inladen, manipuleren en opslaan van afbeelding. We zullen enkel gebruik maken van de inlaadfunctionaliteit.
- `bitarray` : Handig manipuleren van bit arrays. Wordt gebruikt in de LZW compressie. Let op: aangezien dit package een native (niet-Python) component bevat moet er een stukje gecompileerd worden op Linux en macOS. Voor Linux heb je dan ook `python3-dev` nodig (te installeren via `apt`); alternatief kan je eventueel `sudo apt install python3-bitarray` proberen. Miniconda zal bij gebruik van `conda install` geprecompileerde binaries installeren.

Belangrijk: Gebruik maken van `tqdm` mag. Maak verder geen gebruik van extra bibliotheken.

2.2 De GIF encoder

Voor dit practicum hebben we een eenvoudige GIF encoder geïmplementeerd in Python 3. Deze encoder kan enkel één afbeelding met één kleurenpalet encoderen. De volledige standaard bevat echter meer functionaliteit, zoals meerdere sequentiële afbeeldingen en lokale kleurenpaletten.

De encoder kan je oproepen aan de hand van het volgende commando:

`python3 gifencoder.py -i` *`IN.png`* `-o` *`OUT.gif`* `-b` *`BITS`* `[-m` *`LUT_METHOD`*`] [-d` *`DITHER`*`]`

Waarbij argumenten in *cursief* door jou dienen ingevuld te worden, en onderdelen tussen vierkante haken [optioneel] zijn:

- `-i IN.png` : (het pad naar) de **i**nput afbeelding die we wensen te encoderen met de GIF codec.
- `-o OUT.gif` : (het pad naar) waar we het resulterende GIF-bestand willen opslaan (**o**utput file).
- `-b BITS` : bepaalt het aantal **b**its dat kan gebruikt worden per pixel. Met andere woorden, de kleurtabel zal 2^{BITS} groot zijn. We noemen dit verder de b-waarde.
- `-m METHOD` : bepaalt het algoritme (**m**ethode) om de kleurtabel te construeren.
- `-d DITHER` : bepaalt het **d**ithering algoritme. Keuze uit 0, 1 of 2.

De laatste twee argumenten zijn dus optioneel, en komen later in het practicum aan bod. Voor meer info, bekijk: `python3 gifencoder.py -h`

Belangrijk: Respecteer deze command line argumenten, aangezien de automatische verbetering jullie ingediende code zo zal oproepen.

Vraag 0

Wanneer je bovenstaand commando succesvol uitvoert (met zelf ingevulde argumenten), dan krijg je terug:

```
PSNR: nan dB
```

NaN staat voor *not a number*. PSNR is een (te) simpele wiskundige maat die uitdrukt hoe sterk twee afbeeldingen op elkaar lijken. Hoe hoger de PSNR, des te meer dat de afbeeldingen op elkaar lijken. Wat we dus willen is een hoge PSNR na compressie. Stel twee grijswaarden afbeeldingen A en B met m rijen en n kolommen, dan worden de Mean Squared Error (MSE) en PSNR als volgt berekend:

$$\text{MSE} = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(A_{i,j} - B_{i,j})^2 \tag{1}$$

$$\text{PSNR} = 10 \cdot \log_{10}\big(\text{MAX}^2 / \text{ MSE}\big) \tag{2}$$

Hierbij is MAX de maximale waarde die de waarden kunnen bereiken; aangezien de afbeelding in dit practicum 8-bit per kleurkanaal hebben, en we de waarden niet herschalen, is dit dus 255. Wanneer twee afbeeldingen identiek zijn, neem je het logaritme van $+\infty$, wat $+\infty$ blijft.

Wanneer we echter met kleurafbeeldingen werken, wordt deze formule toegepast op elk kleurkanaal en wordt een (gewogen) gemiddelde van de MSE of PSNR genomen. Het gemiddelde berekenen op de MSE en dan overgaan naar PSNR geeft uiteraard niet hetzelfde resultaat wanneer je de PSNR-waarden gebruikt om het gemiddelde te berekenen. Meestal krijgt blauw heel wat minder gewicht in het gemiddelde aangezien we er als mens minder gevoelig voor zijn (zie hoofdstuk over kleuren en volgend practicum). We vragen hier om het gemiddelde van de MSE te gebruiken. Merk op dat dit louter een wiskundige bepaling is dat niet volledig overeenkomt met het menselijk visueel systeem. Het is echter wel snel en gemakkelijk te berekenen.

De NaN die je terug krijgt komt van een nog-niet-geïmplementeerde PSNR-functie. Implementeer PSNR (met gelijke gewichten voor R, G, B) in `gifencoder.py` in volgende functie:

```
def calculate_psnr(img_A, img_B):
```

Hierbij zijn `img_A` en `img_B` al numpy arrays. Maak hierbij gebruik van `numpy`! Python loops (`for`, `while`) zijn niet toegestaan, aangezien Python onwaarschijnlijk traag is (zoals gezien in het introductiepracticum). Numpy is relatief gezien snel, aangezien bijna alle functionaliteit geïmplementeerd is in C.

Hint: Bekijk https://scipy-lectures.org/intro/numpy/operations.html#basic-reductions en raadpleeg de laatste API documentatie van numpy voor informatie. Bekijk ook zeker de resulterende GIF bestanden.

Hint: Denk goed na over de datatypes en de bijhorende berekeningen die je uitvoert. Zorg dat wat je programmeert ook weldegelijk het juiste antwoord produceert. Denk aan precisie en ranges van getaltypes. Test bijvoorbeeld of jouw implementatie symmetrisch is: `psnr(A,B) == psnr(B,A)`.

Belangrijk: Python-loops zijn niet toegestaan! Licht verder (heel!) kort toe wat je gedaan hebt voor deze implementatie in: `vraag-0.md`.

Vraag 1

Encodeer een `kodim`-testafbeelding (kies zelf welke!) met b-waardes: 2, 5, en 8. Herhaal de encodering driemaal per b-waarde. Wat kan je zeggen over de kwaliteit wanneer b groter wordt? Wat valt er je op als je de encodering meerdere malen uitvoert voor éénzelfde b-waarde?

Belangrijk: Formuleer jullie bevindingen in `vraag-1.md`

Belangrijk: Noteer de bestandsnaam (bijvoorbeeld: `kodim23.png`) van de gekozen afbeelding in: `image.txt`; gebruik opnieuw het volledige bestand. Gebruik voor de rest van het practicum dezelfde afbeelding tijdens het rapporteren.

2.3 Kleurenpalet opstellen

Vraag 2.1: Gegeven methode

In de gegeven implementatie wordt het kleurenpalet op een volgende manier opgesteld:

```
def make_random_color_tabel(self):
    self.color_table = np.random.randint(0, 256, (self.color_table_size, 3),
                                         dtype=np.uint8)
```

Dit algoritme wordt opgeroepen met command-line argumenten: `-m random-colors`. Wat voert deze code uit? Bespreek de argumenten van de functieoproep, en leg uit waarvoor deze dienen. Aan wat is `self.color_table_size` gelijk? Waarom is dit een goed of geen goed kleurenpalet voor deze afbeelding?

Belangrijk: Formuleer jullie antwoord in: `vraag-2-1.md`

Vraag 2.2: Grijswaarden methode

Implementeer volgende functie waarbij je 2^b tinten grijs als kleurtabel opstelt (lopende van zwart naar wit).

```
def make_grayscale_color_tabel(self):
```

Deze kan je oproepen met volgende commandolijn argumenten: `-m grayscale`.

Belangrijk: Hier hoef je niets over te schrijven: de code zal gewoon gequoteerd worden. Je hebt deze functie later nodig in het practicum.

Vraag 2.3: Random sampling

We proberen een beter kleurenpalet op te stellen aan de hand kleuren die effectief voorkomen in het beeld. Schrijf de code voor de volgende functie waarbij 2^b kleuren willekeurig gesampled worden uit de afbeelding:

```
def make_random_sample_color_tabel(self):
```

Dit algoritme wordt opgeroepen met command-line argumenten: `-m random-sampling`.

Codeer afbeelding met dezelfde b-waarden (2, 5, en 8). Doe de encodering opnieuw enkele malen per b-waarde en bekijk de resultaten. Wat zijn de bevindingen over de PSNR en de kwaliteit? Een oplijsting van de PSNR-waarden is nutteloos. Geef kwalitatieve inzichten: observeer en probeer te verklaren.

Belangrijk: Formuleer jullie antwoord in: `vraag-2-3.md`

Vraag 2.4: Median-cut algoritme

Typisch wordt voor de gif-encoder het *median-cut* algoritme voorgesteld. In het median-cut algoritme wordt de kleurenruimte steeds verder onderverdeeld in kleinere blokken tot dat er zoveel blokken zijn als dat er plaatsen zijn in het kleurenpalet. Het algoritme werk min of meer als volgt:

> Zolang er niet genoeg blokken zijn, neem een blok en:
>
> 1. Zoek kleurkanaal (r, g, of b) met de grootste variantie,
> 2. Zoek de mediaan kleurwaarde voor dit kanaal,
> 3. Splits het blok op langs deze mediaan op in twee blokken.

Subvraag a: Implementeer het median-cut algoritme voor het opstellen van het kleurenpalet. Het algoritme kan zowel recursief als niet-recursief geïmplementeerd worden. Er is niet één juist antwoord: er kunnen keuzes gemaakt worden. Vul het algoritme aan in:

```
def make_median_cut_color_tabel(self):
```

Het median-cut algoritme wordt opgeroepen met command-line argument: `-m median-cut`.

Verklaar welke ontwerpkeuzes jullie gemaakt hebben:

- Beschrijf de algemene stappen van jullie aanpak.
- Wat gebeurt er als je een blok met maar één unieke kleurwaarde splitst?

- Met welke kleur stel je een blok voor?
- In welk blok komt de mediaan waarde zelf terecht?

Belangrijk: Formuleer jullie antwoord in: `vraag-2-4a.md`

Subvraag b: Codeer opnieuw de gekozen testafbeelding met opnieuw dezelfde b-waarden (2, 5, en 8). Rapporteer de PSNR waarden in het antwoordbestand. Vervolgens, codeer de afbeeldingen `test_circle.ppm` en `test_circle2.ppm` met $b = 1$ en $b = 2$. Rapporteer in het verslag samen met de PSNR waarden.

Belangrijk: Formuleer jullie antwoord in: `vraag-2-4b.md`

Subvraag c: Codeer de afbeeldingen `test_noise_bin.ppm` en `test_bw.ppm` met $b = 1$. Beide afbeeldingen hebben evenveel pixels en evenveel unieke kleuren, namelijk zwart en wit. Bekijk de resulterende bestandsgroottes: Wat valt er op? Wat is een mogelijke verklaring? Noteer ook de PSNR waarden in het verslag.

Belangrijk: Formuleer jullie bevindingen in: `vraag-2-4c.md`

Vraag 2.5: Vector Quantization

Heel gelijkaardig aan kleurpaletten opbouwen is het concept van *Vector Quantization*. In plaats van een sequentie van scalaire getallen te kwantiseren, kwantiseren we een sequentie van vectoren. In ons geval willen we dus een lijst van (r,g,b)-vectoren kwantiseren. De bedoeling is om de tabel te vullen met (r,g,b)-vectoren die representatief zijn voor alle kleuren in de afbeelding. Idealiter willen we dat iedere waarde in de tabel een gelijk aantal originele pixels representeert.

Een goede manier om dit te doen is via de cluster methode *k-means*, een relatief simpele machine learning methode waarvan we de details achterwege laten. Je geeft aan hoeveel clusters je wilt hebben. Aan de hand van de `fit`-functie leert het algoritme wat een goede clusterverdeling is. Iedere cluster heeft dan een center-waarde.

```
from sklearn import cluster
kmeans = cluster.MiniBatchKMeans(n_clusters, n_init=4)
kmeans.fit(data)
```

Implementeer aan de hand van bovenstaande informatie een k-means kleurentabel als de volgende functie:

```
def make_kmeans_color_tabel(self):
```

Raporteer de PSNR waardes voor dezelfde b-waardes (2, 5, en 8) opnieuw voor jullie gekozen testafbeelding.

Belangrijk: Raporteer jullie bevindingen in: `vraag-2-5.md`

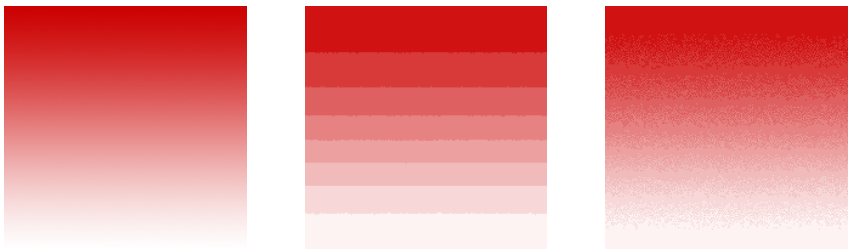
2.4 Dithering

De eerste afbeelding in Fig. 1 toont David van Michelangelo in 256 grijswaarden, de tweede toont slechts 2 grijswaarden (zwart en wit). Merkwaardig genoeg toont de derde afbeelding ook maar twee grijswaarden. Enkel is in de derde afbeelding niet telkens gekozen voor het dichtstbijzijnde kleur in kleurenpalet. In de plaats wordt nu en dan gekozen voor een ander kleur.

(a) 256 grijswaarden (b) Twee grijswaarden (c) Dithered

Figuur 1: https://en.wikipedia.org/wiki/Dither

Er bestaat een groep aan methodes die deze patronen bepalen om de illusie te wekken dat er tussenliggende kleuren zijn, nl. *dithering*. In de lessen hebben jullie *ordered dithering* gezien, dewelke gebruik maakt van een dither matrix. Bij *pattern dithers* worden de lengte en breedte van de afbeelding vergroot met een factor k en wordt iedere pixel dan voorgesteld door een patroon van $k \times k$ pixels. Hierbij is er dus duidelijk een trade-off tussen spatiale resolutie en kleurenresolutie. Er bestaan ook andere dither-methoden. Bijvoorbeeld, aan de hand van ruistoevoeging wordt de gemaakte kwantisatiefout gerandomiseerd. M.a.w. er wordt niet telkens voor de dichtstbijzijnde kleur gekozen. Dit zorgt er voor dat grote structurele artefacten minder zichtbaar worden, zoals in het Fig. 2.



(a) 256 kleuren (b) 8 kleuren (c) Dithered

Figuur 2: https://www.lifewire.com/dithering-gif-images-4122770

De middelste afbeelding van Fig. 2 toont een veelvoorkomend coderingsartefact, nl. *color banding*. Dit ontstaat doordat er een discrete sprong is waarbij consistent voor een andere dichtstbijzijnde kleur wordt gekozen. De rechtse afbeelding toont hoe dithering hier de indruk geeft van een veel vloeiendere gradiënt. Hoewel deze methode de spatiale resolutie niet verhoogt, kan deze zelfde methode ook toegepast worden in combinatie met het verhogen van de resolutie. Het resultaat zal er dan nog beter uitzien, aangezien onze ogen de kleuren dan beter kunnen mengen omdat de ruis fijner is. In dit practicum kunnen jullie de resolutie gelijk houden.

Vraag 3.1: Floyd-Steinberg diffusie

In onze huidige versie van de GIF-encoder wordt steevast voor de meest naburige kleurwaarde in het kleurenpalet gekozen. Dit wilt zeggen dat er consistent naar boven of naar onder "afgerond"wordt.

In deze opgave zullen we hierop voortbouwen door een dithering methode te implementeren die de afbeeldingsgrootte niet vergroot. Diffusie dithering is een methode die kwantisatiefout naar de naburige pixels doorschuift. Het idee is dat eenmaal een pixel naar beneden is afgerond, dat je dan de kans wilt verhogen dat de volgende pixel eens naar boven wordt afgerond.

Het Floyd-Steinberg diffusie-dithering algoritme neemt de volgende matrix aan om de kwantisatiefout door te schuiven:

$$\frac{1}{16}\begin{bmatrix} - & * & 7 \\ 3 & 5 & 1 \end{bmatrix}. \tag{3}$$

Het algoritme gaat over de rijen van de afbeelding naar beneden. De pixel die nu gekwantiseerd wordt is aangeduid met een ster. Delen van de kwantisatiefout worden doorgeschoven naar de pixels rechts en onder de huidige pixel. Dit vertaalt zich naar de volgende pseudocode:

```
for each y from top to bottom
    for each x from left to right
        old_pixel  :=  clip(pixel[x, y], 0, 255)
        new_pixel  :=  find_closest_palette_color(old_pixel)
        pixel[x, y]  :=  new_pixel
        quant_error  :=  old_pixel − new_pixel
        pixel[x + 1, y    ] ←  pixel[x + 1, y    ] + quant_error * 7 / 16
        pixel[x − 1, y + 1] ←  pixel[x − 1, y + 1] + quant_error * 3 / 16
        pixel[x    , y + 1] ←  pixel[x    , y + 1] + quant_error * 5 / 16
        pixel[x + 1, y + 1] ←  pixel[x + 1, y + 1] + quant_error * 1 / 16
```

Implementeer het bovenstaande algoritme in de functie `transform_image(dithering)` wanneer `dithering == 1`. Gebruik van Python-loops is toegestaan.

Hint: De afbeeldingen zitten in *row-major order* in het geheugen. Dit betekent dat de afbeelding rij per rij in het geheugen zit. Twee horizontaal naburige pixels zitten dus naast elkaar in het geheugen. Twee verticaal naburige pixels zitten dus ver uit elkaar in het geheugen. Numpy indexeert ook in row-major volgorde: de eerste index is de rij, de tweede is de kolom.

Bespreek enkele nuttige en/of subtiele elementen van jullie implementatie.

Belangrijk: Rapporteer in: `vraag-3-1.md`

Vraag 3.2: Eerste tests

Codeer jullie kozen testafbeelding én `david.png` met enkele b waarden. Maak gebruik van jullie `median-cut` algoritme en van de `grayscale` kleurentabel en vergelijk met en zonder dithering. Wat werkt het beste, is het mooiste, vind je?

Belangrijk: Rapporteer in: `vraag-3-2.md`

Vraag 3.3: Bestandsgroottes

Vergelijk de bestandsgroottes met en zonder dithering. Wat valt er op en hoe kan je het verklaren? Bestandgroottes zelf hoef je niet te rapporteren: bespreek kwalitatief.

Belangrijk: Formuleer jullie antwoord in: `vraag-3-3.md`

Vraag 3.4: Kwaliteit

Vergelijk de kwaliteit visueel en PSNR waarden met en zonder dithering. Licht toe en verklaar? Bespreek opnieuw kwalitatief.

Belangrijk: Formuleer jullie antwoord in: `vraag-3-4.md`

Vraag 3.5: Minimized Averaged Error

Het "minimized average error" diffusie dithering methode werkt op dezelfde manier. Enkel wordt de kwantisatiefout verder verspreid. De spreidingsmatrix ziet er namelijk zo uit:

$$\frac{1}{48}\begin{bmatrix} - & - & * & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

Implementeer deze methode wanneer `dithering == 2` en vergelijk de twee methoden visueel voor gelijke b waardes. Gebruik opnieuw dezelfde twee afbeeldingen: de zelf gekozen afbeelding en `david.png`. Welke methode verkies je?

Belangrijk: Formuleer jullie antwoord in: `vraag-3-5.md`

3 In te dienen bestanden

Pak jullie oplossing in in een zip-file, en stuur jullie door via de Opdracht op Ufora, met de bestandsnaam:

Voor Multimedia: `practicum_kleurpaletten_MM_XX.zip`
Voor Multimediatechnieken: `practicum_kleurpaletten_MMT_XX.zip`

Hierbij vervangen jullie `XX` door jullie groepsnummer (**twee cijfers!**). Hierin zitten jullie Python files (`gifencoder.py` en `lzw.py`) samen met jullie antwoord bestanden (`*.md` en `image.txt`). Stuur dus **geen afbeeldingen** mee.

Inpakken in zip-file kan met (bijvoorbeeld met Bash, Zshell, etc...):

```
zip practicum_kleurpaletten_MM_04.zip *.md *.py image.txt
```