1. Scaling

1.1. Namen en datum

Sander Kolman
Alexander Freeman
3-6-2016

1.2. Doel

Voordat iedere afbeelding door de rest van de beeldherkenning heen gaat moet er gezorgd worden dat alle foto's een gelijk en niet te groot formaat hebben. Een gelijk formaat hebben voor iedere afbeelding is comfortabel omdat er directe aanpassingen gedaan kunnen worden op de pixel i.p.v altijd rekening te moeten houden met de grootte afbeelding waarin gewerkt wordt. Belangrijker is dat de afbeelding niet te groot is omdat er dan meer pixels uitgerekend worden bij iedere stap. Tot een aantal pixels is dit noodzakelijk omdat iedere pixel cruciale informatie bevat, maar te veel pixels geven geen nuttige informatie meer voor onze beeldherkenning en wel extra overhead.

1.3. Methoden

Bij iedere vorm van transformatie op een afbeelding (schalen, translatie, rotatie) is de eerste gedachte om voor iedere oude pixel in de afbeelding de nieuwe pixel te berekenen en daar de kleur van de oude pixel in schrijven. Dit wordt Forward Mapping genoemd en heeft een ongewenst effect wat mogelijk niet onmiddellijk duidelijk is maar zeker direct zichtbaar wordt als het uitgevoerd wordt. Omdat er voor iedere oude pixel de nieuwe positie van de pixel wordt berekent is er geen garantie dat op die manier iedere nieuwe pixel een waarde krijgt. Dit is goed voor te stellen als je een afbeelding 2 keer zo lang maakt. Plots heb je in de nieuwe afbeelding 2 keer zo veel pixels in de hoogte dan in de oude afbeelding. Als je nu voor iedere oude pixel één nieuwe pixel zoekt ben je ervan verzekerd dat je op zijn hoogst maar de helft van de nieuwe afbeelding kan vullen. Als je voor alle oude pixels een nieuwe plek hebt gevonden, heb je nog maar de helft van de nieuwe pixels gehad.
De oplossing hiervoor heet Backward Mapping en wordt overduidelijk de beste oplossing als de vraag opnieuw geformuleerd wordt als: "welke kleur moet iedere pixel in de nieuwe afbeelding worden als het een transformatie is van de oude afbeelding". Hierbij moet er wel aan gedacht worden de inverse te gebruiken van de transformatie van Forward Mapping. Deze oplossing is echter geen pure magie (zoals je waarschijnlijk ook niet dacht) maar het

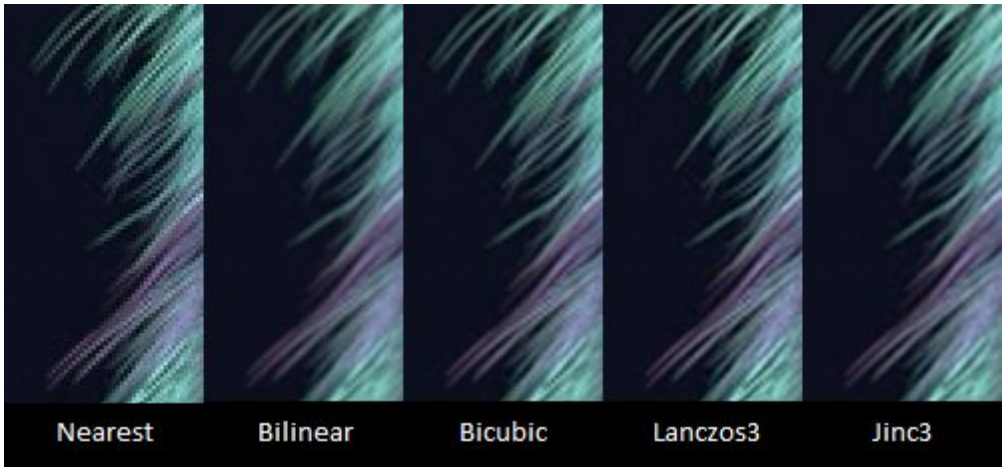
heeft ook fouten. Alhoewel nu iedere pixel garantie heeft een kleur te krijgen, kan het zijn dat de berekening een getal als uitkomst heeft die niet correspondeert met een enkele pixel maar met meerdere door er als het ware tussen te liggen.
Er zijn verschillende methodes om met dit verschijnsel om te gaan, een echte oplossing kan het echter niet genoemd worden aangezien er nooit zeker te stellen is dat de pixel de correcte kleur krijgt. Hieronder een opsomming van de methodes op volgorde van efficiënt en slordiger naar inefficiënt en netter:

- Nearest Neighbour, 0th Order
- Bilinear, 1st Order
- Bicubic, 2nd Order

Nearest Neighbour:
Deze methode is verreweg het eenvoudigst maar dat is ook te merken in de kwaliteit. De methode werkt door te kijken welke oude pixel het meest in de buurt ligt van de waarde die berekent is door Backward Mapping. De naam sluit er overduidelijk ook erg goed op aan, je kijkt naar de buur die het dichtst bij staat, vervolgens wordt deze pixel gebruikt voor de nieuwe pixel. In de meeste gevallen treed hier heftige aliasing op. Hieronder een voorbeeld daarvan:

Bilinear:
Deze methode kijkt naar alle 4 direct omliggende pixels en berekent gewichten voor ieder van die pixels. Hoe verder de pixel, hoe lager het gewicht. De kleur van iedere pixel wordt vermenigvuldigd met het gewicht van de desbetreffende pixel en alle kleuren worden hierna bij elkaar opgeteld. Alle gewichten bij elkaar opgeteld komen altijd uit op 1 en daarom komt de uiteindelijke kleur van de pixel ook uit op een waarde die in de normale range van die pixel ligt.
Dit zorgt voor een veel evenwichtiger effect.

Bicubic:
Deze methode is te vergelijken met Bilinear, het algoritme neemt i.p.v. 4 omliggende pixels, 16 omliggende pixels in acht. Hier worden extra gewichten aangelegd om de mate dat een pixel mee moet tellen goed te berekenen.



1.5. Keuze

We hebben er voor gekozen bilinear te implementeren, omdat we verwachten dat het resultaat hiervan goed genoeg zal zijn voor alleen het schalen van afbeeldingen. De implementatie hiervan is ook nog redelijk eenvoudig en efficiënt. Beide eigenschappen maken dat dit naar verwachting de beste keuze is.
Dit wordt geïmplementeerd via backward mapping, aangezien dit verreweg de beste methode is en de minste problemen veroorzaakt.

1.5. Implementatie

Wij implementeren de scaling in de StudentPreProcessing class. Dit wordt gedaan door langs elke pixel te gaan en uit te rekenen waar de pixel vandaan kwam. Dit doen we door te schalen met de verhouding tussen source.width / result.width en source.height / result.height. Hier kunnen dan pixels uit komen als (2.5, 6.43). Om dit op te lossen interpoleren wij met pixels out de source afbeelding. Deze interpolatie wordt geïmplementeerd in de ImageUtils class. Hier wordt een method interpolate_first_order en interpolate_nearest_neighbour gemaakt. Deze worden gebruikt in het scaling algoritme.

Om verhoudingen goed te houden, houden we de verhouding aan van het origineel. We schalen de breedte naar 200 en vermenigvuldigen dan de originele hoogte met (200/old_ width). Hiermee verkrijgen we de nieuwe hoogte met de juiste verhouding.

1.5. Evaluatie

Wij gaan een aantal attributen controleren. Allereerst willen we testen hoe snel ons algoritme is geïmplementeerd in vergelijking met de opencv standaard. Daarnaast willen we weten wat de performance implicaties zijn van first order vs nearest neighbour interpolatie.

Naast performance willen we ook weten wat het resultaat is, met de testset als input. Dit willen we weten, omdat de algoritmes met onze input zich misschien anders kunnen gedragen dan de voorbeelden als internet. Een oorzaak zou bijvoorbeeld kunnen zijn dat wij interpoleren in intensities in plaats van kleuren. Daarnaast zou de resolutie van onze afbeeldingen invloed kunnen hebben op de resultaten.