Color Theory, Part 2: Uniform Color Spaces

Stephen Huan

TJ Vision & Graphics Club, February 3, 2021

k-means

k-means

k-means algorithm

The standard algorithm alternates between two steps until convergence: Given k initial center points,

1. Assign each point to its closest center
2. Update each center to the centroid of the points assigned to it, where the centroid is the arithmetic mean.

Derivation of *k*-means: Point Assignment

Theorem

If the centers are fixed, the best point to center assignment is to pick the closest center for each point.

Proof.

Each point is independent, so consider an arbitrary point. If we don't pick the center closest to it, the distance will be greater. Thus, we pick the closest center. □

Derivation of *k*-means: Center Location

Theorem

If the point to center assignment is fixed, then the best center placement is the centroid of the points assigned to it.

Derivation of k-means: Center Location

Proof.

We want to minimize the sum of squares distance:

$$
\begin{aligned}
\min_{\vec{y}} \sum_{i=0}^{n} \|\vec{y} - \vec{x}_i\|^2 \\
= \sum (\vec{y} - \vec{x}) \cdot (\vec{y} - \vec{x}) \\
= \sum (\vec{y} \cdot \vec{y} - 2\vec{x} \cdot \vec{y} + \vec{x} \cdot \vec{x})
\end{aligned}
$$

Derivation of *k*-means: Center Location

Proof.

We now differentiate with respect to $\vec{y}$ and set equal to $\vec{0}$

$$\sum_{i=0}^{n}(2\vec{y} - 2\vec{x}_i) = \vec{0}$$

$$2n\vec{y} = 2\sum \vec{x}$$

$$\boxed{\vec{y} = \frac{\sum \vec{x}}{n}}$$

$\square$

Implications

- Point to center assignment relies on minimum distance
- Centroid relies on Euclidean distance
  - Minimize norm of ***y*** - ***x***
- Suppose we switch distance metrics
- Point assignment computation trivial
- Centroid computation nontrivial
  - Need to differentiate, unlikely to get clean analytical solution
  - Use numerical methods e.g. gradient descent
  - Some metrics are very complicated/non-continuous or non-differentiable
- Hence, easier to switch *spaces*

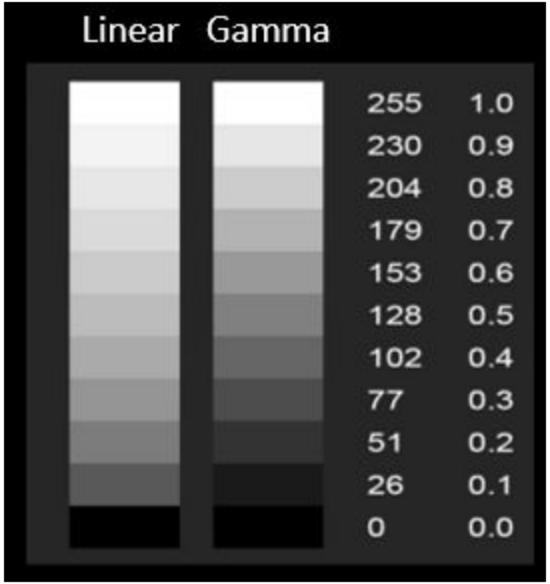
Uniform Color Spaces

- Projection from RGB
- Use Euclidean distance for distance
- Should “just work”
- Goal is perceptual uniformity i.e. MacAdam ellipses

Uniform Color Spaces

sRGB vs RGB

- RGB: “linear”, e.g. adding intensities for greyscale
  - Corresponds to voltage
- sRGB: nonlinear, based off actual monitor
  - Corresponds to brightness, $L = V^{\gamma}$
  - So we take $V \text{ -> } V^{1/\gamma}$
- 8-bit = only 256 colors
  - Prioritize black for perceptual uniformity



Y'UV

- Y': brightness, U/V: color
- Matrix multiplication from RGB
- Which RGB?

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- Y′UV is computed from RGB (linear RGB, not gamma corrected RGB or sRGB for example)
- Wait, but

```
return _convert(yuv_from_rgb, rgb)
```

scikit-image

```
kernel = ops.convert_to_tensor(
    _yuv_to_rgb_kernel, dtype=images.dtype, name='kernel')
ndims = images.get_shape().ndims
return math_ops.tensordot(images, kernel, axes=[[ndims - 1], [0]])
```

tensorflow

- In particular, `catimg`, scikit-image, and tensorflow do *not* gamma correct
- No one really knows

Color Appearance Models

- CIECAM02/CAM16 are color *appearance* models
- That is, given XYZ (wavelengths) predict how color appears
- CAM16 output: Correlates of lightness J, chroma C, hue composition H, hue angle h, colorfulness M, saturation s, and brightness Q
- We want color *space*
- Luckily, uniform color space (UCS) variants
- J lightness, a red/green, b blue/yellow
  - CIELAB basically

Implementing the CAMs

- CIECAM02 and CAM16 have the same “body”, just a different initial “color appearance transformation” (CAT)

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \mathbf{M}_{\mathrm{HPE}} \mathbf{M}^{-1} \boldsymbol{\Lambda}(D) \mathbf{M} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Step 1: Calculate ‘cone’ responses

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \mathbf{M}_{16} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Step 2: Complete the color adaptation of the illuminant in the corresponding cone response space (considering various luminance levels and surround conditions included in D, and hence in D_{R}, D_{G}, and D_{B})

$$\begin{pmatrix} R_c \\ G_c \\ B_c \end{pmatrix} = \begin{pmatrix} D_{\mathrm{R}} \cdot R \\ D_{\mathrm{G}} \cdot G \\ D_{\mathrm{B}} \cdot B \end{pmatrix}$$

- Thus, most efficient to *subclass* CAM
  - Only difference is M and M^{-1}
- CAMXY inherits methods from base CAM

CAM Forward Implementation

```python
def CAT(self, c: tuple) -> tuple:
    """ Linear transformation. """
    # combines step 1: cone responses and step 2: color adaptation
    return mulv(self.M, c)

def CAM(self, c: tuple) -> tuple:
    """ Color appearance model after color appearance transform. """
    # step 3: postadaptation cone response
    RGBp = tmap(lambda x: sign(x)*self.post(abs(x)), c)
    # step 4: color components a/b, hue angle h, auxililary variables pp2/u
    pp2, a, b, u = mulv(Mdot, RGBp)
    h = hue_angle(b, a)
    # step 5: eccentricity
    hp = h + (h < hue_data[0][0])*360
    for i in range(len(hue_data) - 1):
        if hue_data[i][0] <= hp < hue_data[i + 1][0]:
            break
    et = 1/4*(cos(hp + deg(2)) + 3.8)
    hi, ei, Hi, h1, e1, H1 = hue_data[i] + hue_data[i + 1]
    H = Hi + 100*e1*(hp - hi)/(e1*(hp - hi) + ei*(h1 - hp))
    # PL, PR = round(H1 - H), round(H - Hi)
    A = pp2*self.Nbb                          # step 6: achromatic response
    J = 100*(A/self.Aw)**(self.c*self.z) # step 7: correlate of lightness
    # step 8: correlate of brightness
    Q = 4/self.c*sqrt(J/100)*(self.Aw + 4)*self.FL**0.25
    # step 9: correlate of chroma C, colorfulness M, saturation s
    t = 50000/13*self.Nc*self.Ncb*et*sqrt(a**2 + b**2)/(u + 0.305)
    alpha = t**0.9*(1.64 - 0.29**self.n)**0.73
    C = alpha*sqrt(J/100)
    M = C*self.FL**0.25
    s = 50*sqrt(alpha*self.c/(self.Aw + 4))
    return (J, C, H, h, M, s, Q)
```

CAM Inverse Implementation

```python
def CAMinv(self, c: tuple) -> tuple:
    """ Reverse model of the color appearance transform. """
    J, _, _, h, M, _, _ = c
    # step 1: get J, t, and h
    C = M/self.FL**0.25
    alpha = 0 if J == 0 else C/(sqrt(J/100))
    t = (alpha/(1.64 - 0.29**self.n)**0.73)**(1/0.9)
    # step 2: et, A, pp1, pp2
    et = 1/4*(cos(h + deg(2)) + 3.8)
    A = self.Aw*(J/100)**(1/(self.c*self.z))
    pp1 = et*50000/13*self.Nc*self.Ncb
    pp2 = A/self.Nbb
    # step 3: a and b
    gamma = 23*(pp2 + 0.305)*t/(23*pp1 + 11*t*cos(h) + 108*t*sin(h))
    a, b = gamma*cos(h), gamma*sin(h)
    RGBp = scale(mulv(Mdotinv, (pp2, a, b)), 1/1403) # step 4: RGBp
    RGBc = tmap(self.postinv, RGBp)                  # step 5: RGBc
    return RGBc

def CATinv(self, c: tuple) -> tuple:
    """ Undo the color appearance transformation. """
    # combines step 6: RGB and step 7: X, Y, Z
    return mulv(self.Minv, c)
```

Uniform Color Space

```python
def from_xyz(self, c: tuple) -> tuple:
    """ XYZ to CAMXY color space. """
    return self.CAM(self.CAT(c))

def to_xyz(self, c: tuple) -> tuple:
    """ CAMXY to XYZ color space. """
    return self.CATinv(self.CAMinv(c))

def to_ucs(self, c: tuple, c1: float=0.007, c2: float=0.0228) -> tuple:
    """ CAMXY to CAMXY-UCS color space. """
    J, _, _, h, M, _, _ = c
    Jp, Mp = (1 + 100*c1)*J/(1 + c1*J), ln(1 + c2*M)/c2
    return (Jp, Mp*cos(h), Mp*sin(h))

def from_ucs(self, c: tuple, c1: float=0.007, c2: float=0.0228) -> tuple:
    """ CAMXY-UCS to CAMXY color space. """
    Jp, ap, bp = c
    Mp, h = sqrt(ap**2 + bp**2), hue_angle(bp, ap)
    M, J = (exp(c2*Mp) - 1)/c2, Jp/(1 + c1*(100 - Jp))
    return (J, None, None, h, M, None, None)
```

$$J' = \frac{1.7J}{1+0.007J}$$

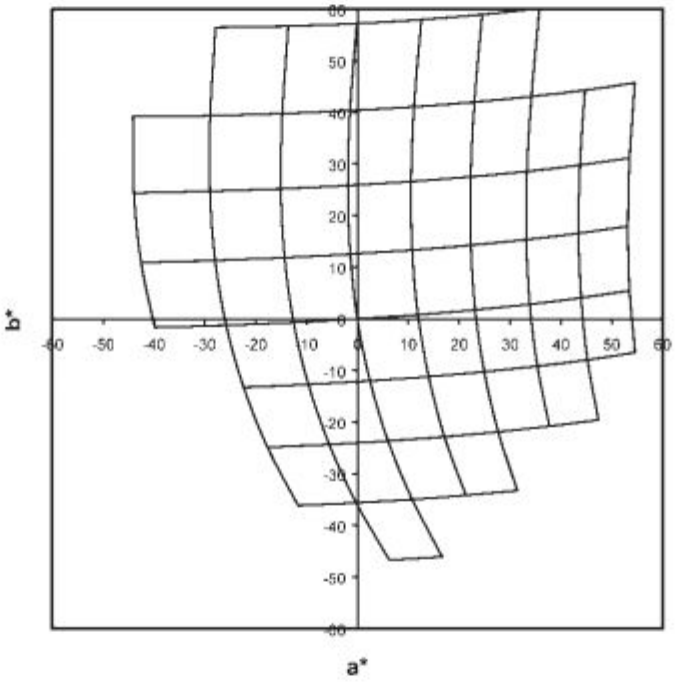
$$M' = \ln(1+0.0228M)/0.0228$$

$$a' = M'\cos(h)$$

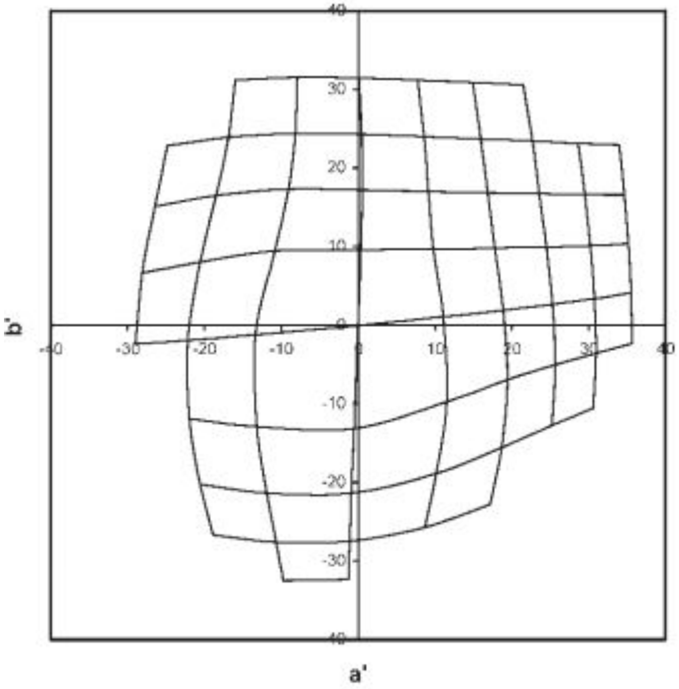
$$b' = M'\sin(h)$$

$$h' = h$$

Qualitative Performance

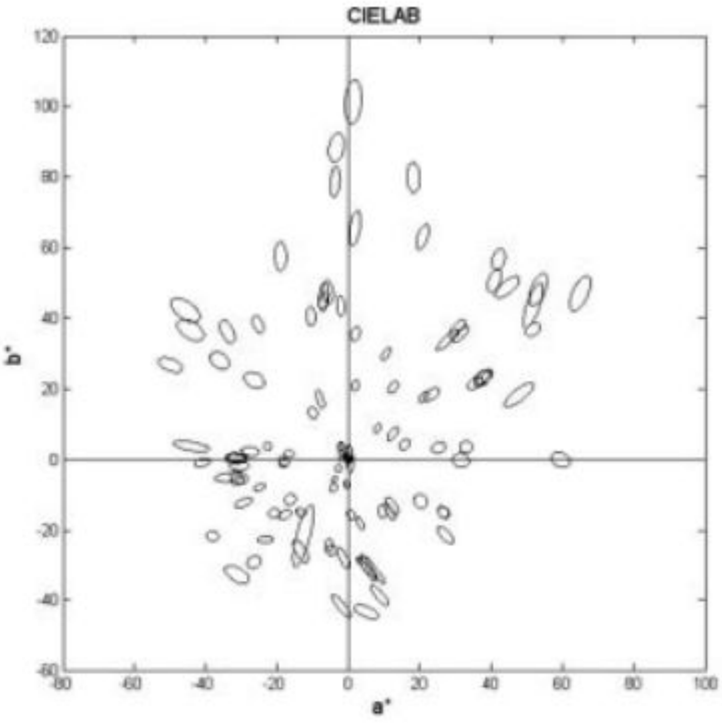


(a) CIELAB

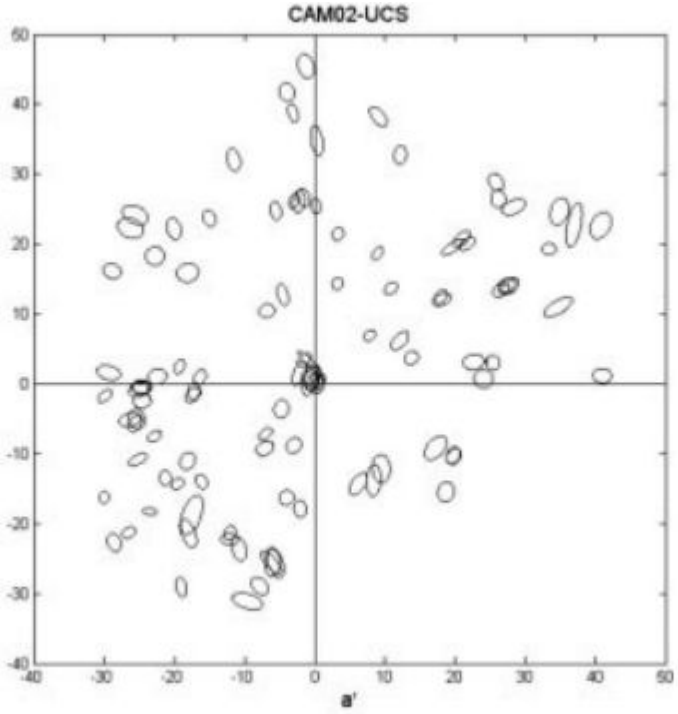


(f) CAM02-UCS

Qualitative Performance



(a) CIELAB



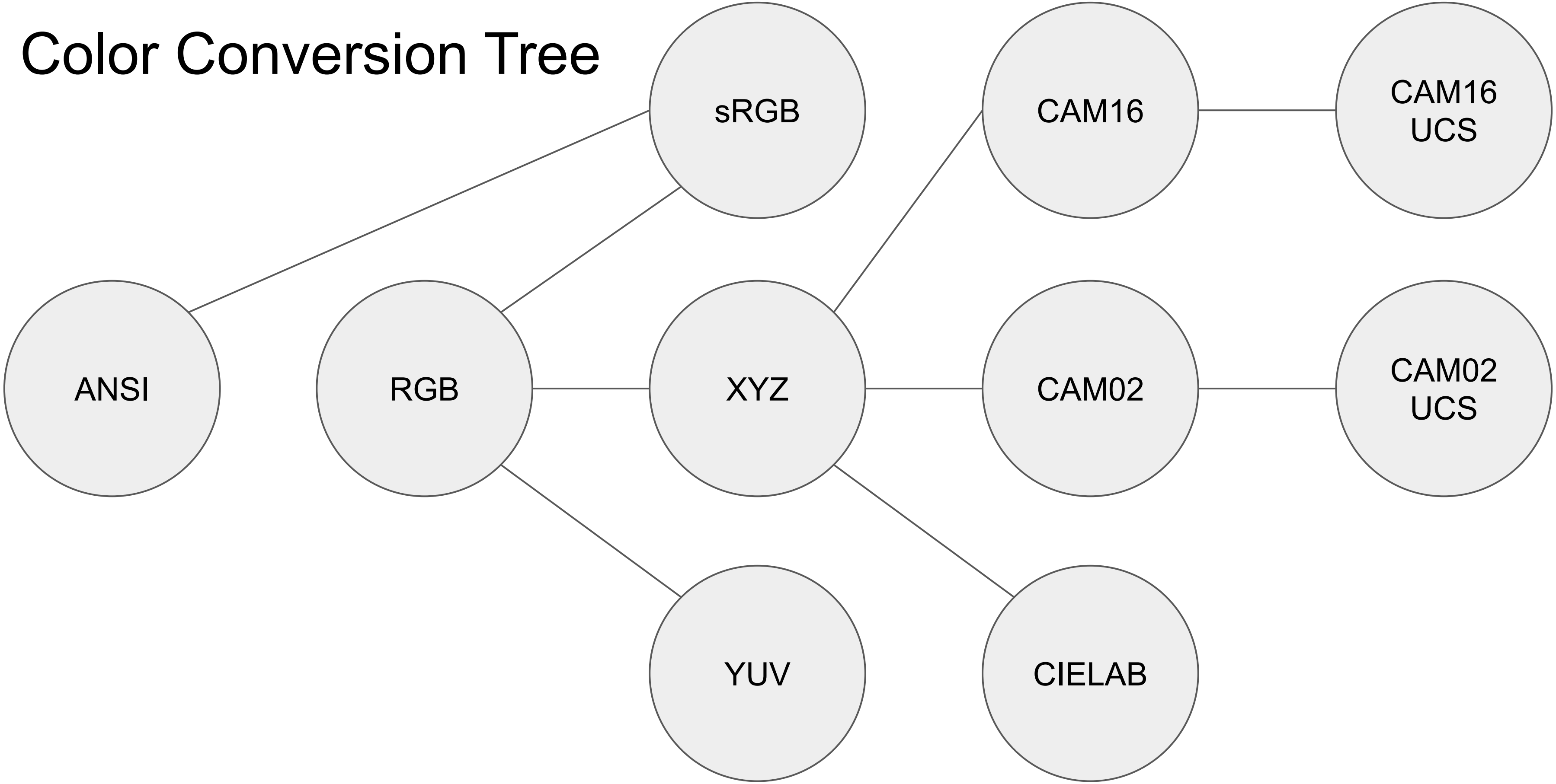
(f) CAM02-UCS

Miscellaneous

One Weird Trick Color Scientists DON'T Want You to Know: Lower Your *STRESS* By 17%!

- Steven's power law: $S = a\, I^b$
  - S: subjective magnitude, I: physical stimulus
- $\Delta E' = a\, \Delta E^b$
- Kind of but not really
- *STRESS*: measure of color difference performance, lower is better
- Lowers *STRESS* by an average of 17%
- Why does this work?
  - $b < 1$: compresses the space (small values -> larger, larger values -> smaller)
  - Raters have difficulty with small values
  - Also large values asymptote

Color Conversion Tree



Color Conversion API

- Could hand-design
  - 10 choose 2 = 45
- Could have “central” color, e.g. XYZ
  - CAM16 -> CAM16UCS
  - CAM16 -> XYZ -> CAM16 -> CAM16UCS
- BFS from one space to the other
  - Minimize # of functions -> faster, less numerical error

Color Difference

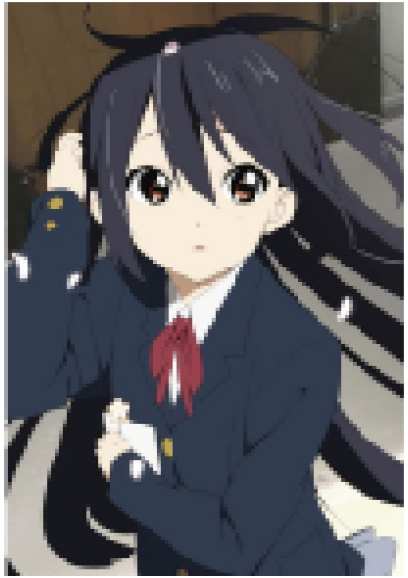
- Distance parameterized by (space, metric)
- e.g. (RGB, Euclidean) or (CIELAB, CIEDE2000)
- 8 spaces + 3 metrics = 11 possible!
- Power function for 6 -> 17 possible
- Can technically apply custom metrics on non-CIELAB
    - Surprisingly, this kinda… works?

Experimental Results

ANSI Color Codes

- Image -> text
- catimg
- Some terminals support 3-byte “true color”
- Neither Vim’s AnsiEsc nor Vim’s Colorizer can render these
- Thus, have to use 255 color mode

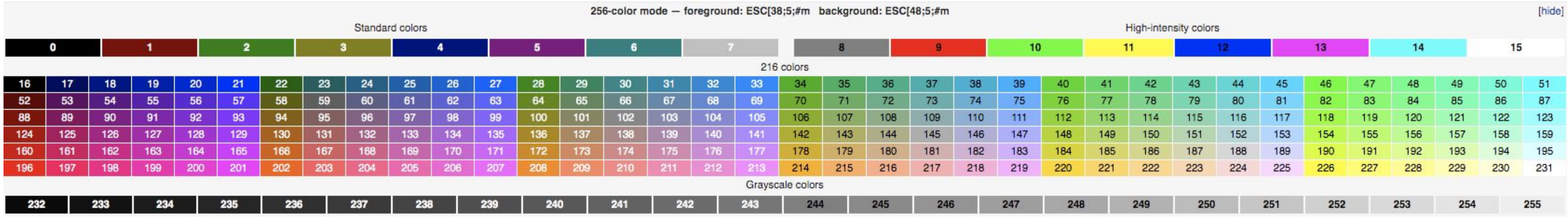
a. original image



b. `catimg` 24-bit “true color”

c. `catimg` 8-bit

The Algorithm



- Downscale image somehow (averaging window, bicubic, bilinear, etc.)
- Convert color to nearest ANSI color
  - Find closest color!
  - `catimg` uses YUV color space + Euclidean which explains the poor quality

Downscaling

- Use ffmpeg
- Good
  - Area (used by catimg)
  - Experimental
- Okay
  - Bilinear
  - Gauss
- Bad
  - Bicubic
  - Neighbor
  - Bicublin
  - Sinc
  - Lanczos
  - Spline

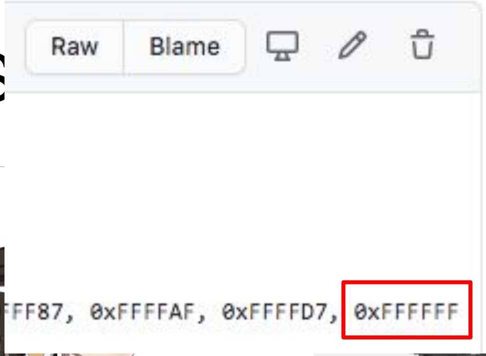
Downscaling

- `full_chroma_int` and `full_chroma_inp` don’t seem to do anything
- Could use uniform color space for the averaging
- Makes very little difference

Dataset

- Album cover of *Nisemonogatari Gekihanongakushu (Original Soundtrack)*
  - Album covers are nice and square, low resolution (544x544)
- Colors reveal failure points

Getting S



not sure why `catimg` doesn't render white…?

original

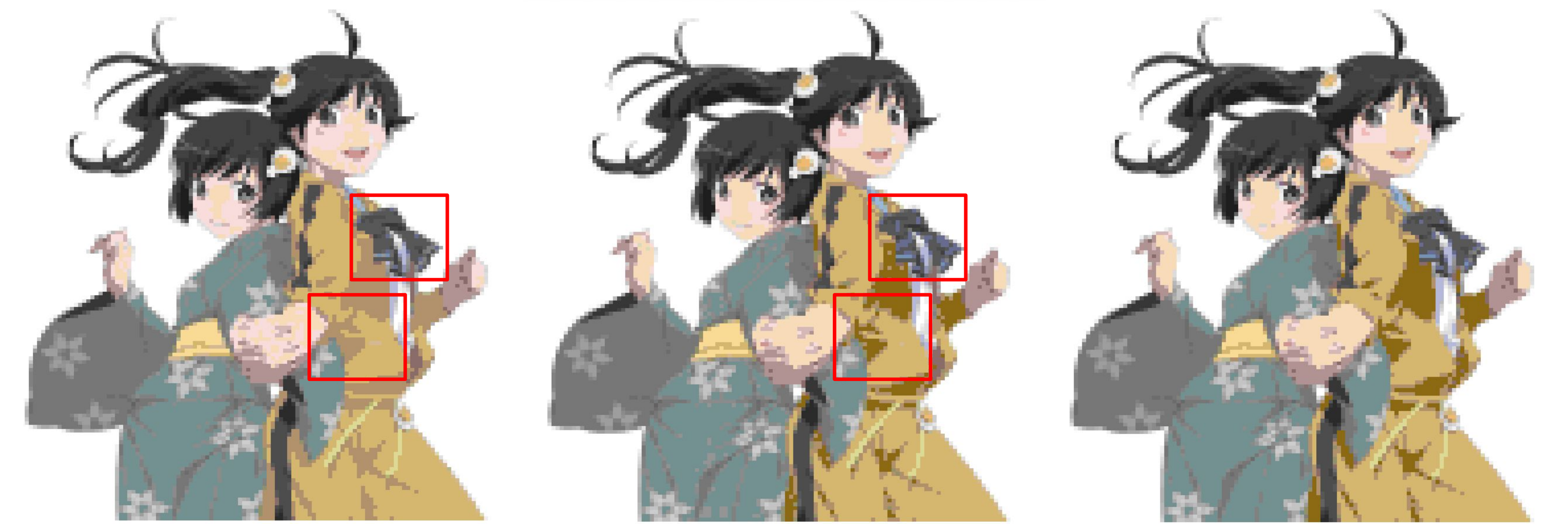
`catimg`

YUV

Basics

		
ansi256	RGB	XYZ

“Perceptually Uniform” Spaces



CIELAB	CIECAM02-UCS	CAM16-UCS

Distance Metrics

CIE94

CIE2000

CMC

k-means

Choosing *k*

- If *k* too big, differences minor -> color space doesn't matter
- If *k* too small, not much able to do -> also doesn't matter
- Need *k* to be just right
- In this case, *k* = 32
- Anime images are too easy, real life is better
    - 131,707 distinct pixels versus 28,641: 4.6x difference

Dataset #2

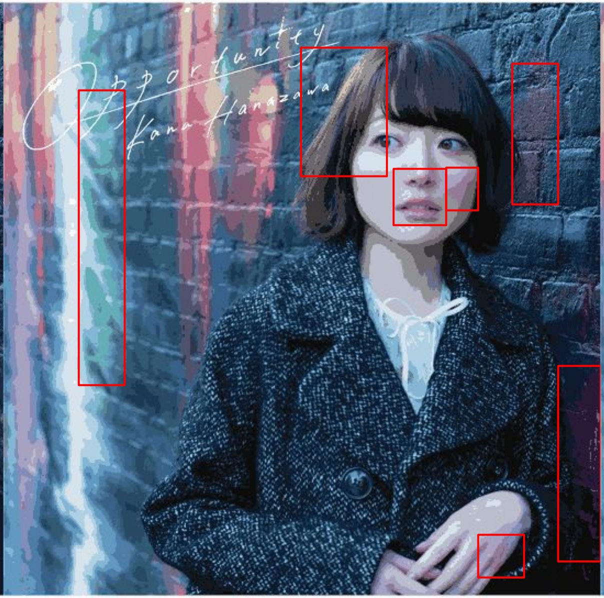
- Album cover of *Opportunity* by Kana Hanazawa

Basics

RGB	YUV	XYZ

Perceptually Uniform Spaces

CIELAB



CIECAM02-UCS

CAM16-UCS

	CIECAM02	original	XYZ	
✓				
✓				
✓				
✓				
				✓
Final score:	205,064	to	90,872	(CIEDE2000)


Opportunity
Kana Hanazawa


Opportunity
Kana Hanazawa

Conclusion

- Color perception != image quality
- Problems like this are really difficult
- Machine learning?

Sources

- My implementation(s)
- Previous lecture: Color Theory, Part 1: Color Difference Metrics
  - Check the sources in that presentation, not going to repeat!
- sRGB - Wikipedia
- Linear, Gamma and sRGB Color Spaces
- What's wrong with 8-bit linear RGB?
- YUV - Wikipedia
- catimg YUV implementation
  - scikit-image, tensorflow (tf.image)
- Delta E (CMC)
- Scaling - FFmpeg

Sources

- Power functions improving the performance of color-difference formulas
- The CIECAM02 color appearance model
- Uniform Colour Spaces Based on CIECAM02 Colour Appearance Model
- Comprehensive color solutions: CAM16, CAT16, and CAM16-UCS
- Algorithmic improvements for the CIECAM02 and CAM16 color appearance models