Practical Pigment Mixing for Digital Painting

ŠÁRKA SOCHOROVÁ and ONDŘEJ JAMRIŠKA,
Czech Technical University in Prague, Faculty of Electrical Engineering, Czech Republic and Secret Weapons, Czech Republic

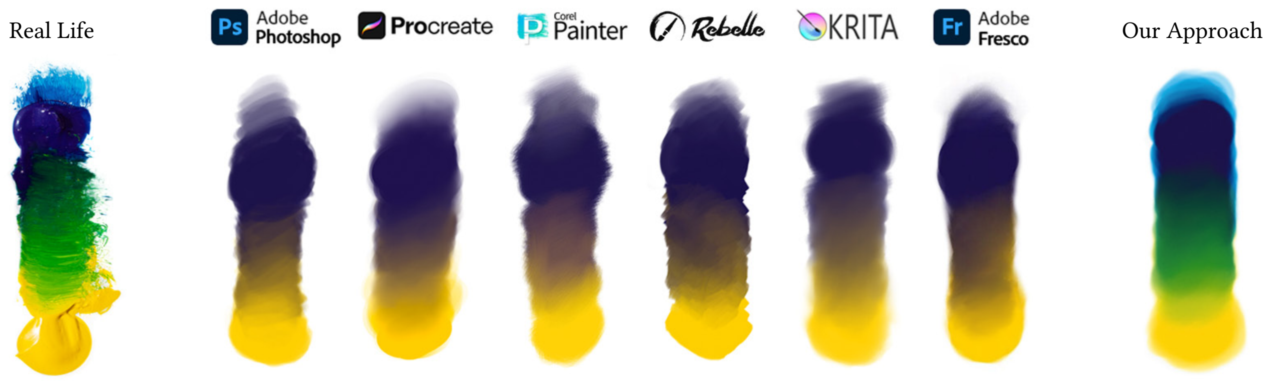

Fig. 1. In real life, blue and yellow make green. Yet, in today's painting software, they mix toward gray. Our mixing approach rectifies this by leveraging the Kubelka–Munk model while operating in RGB-In / RGB-Out fashion, which makes it practical to integrate into any existing painting software.

There is a significant flaw in today's painting software: the colors do not mix like actual paints. E.g., blue and yellow make gray instead of green. This is because the software is built around the RGB representation, which models the mixing of colored lights. Paints, however, get their color from pigments, whose mixing behavior is predicted by the Kubelka–Munk model (K–M). Although it was introduced to computer graphics almost 30 years ago, the K–M model has never been adopted by painting software in practice as it would require giving up the RGB representation, growing the number of per-pixel channels substantially, and depriving the users of painting with arbitrary RGB colors. In this paper, we introduce a practical approach that enables mixing colors with K–M while keeping everything in RGB. We achieve this by establishing a latent color space, where RGB colors are represented as mixtures of primary pigments together with additive residuals. The latents can be manipulated with linear operations, leading to expected, plausible results. We describe the conversion between RGB and our latent representation, and show how to implement it efficiently.

CCS Concepts: • **Computing methodologies** → **Computer Graphics**; • **Software and its engineering** → **Software functional properties**.

Additional Key Words and Phrases: pigment mixing, digital painting

ACM Reference Format:
Šárka Sochorová and Ondřej Jamriška. 2021. Practical Pigment Mixing for Digital Painting. *ACM Trans. Graph.* 40, 6, Article 234 (December 2021), 11 pages. https://doi.org/10.1145/3478513.3480549

Author's addresses: Šárka Sochorová, sochorova@scrtwpns.com; Ondřej Jamriška, jamriska@scrtwpns.com, Czech Technical University in Prague, Faculty of Electrical Engineering, Karlovo náměstí 13, Praha 2, 121 35, Czech Republic.

1 INTRODUCTION

What happens when we mix blue and yellow? Any 5-year-old knows we get green. We gain that intuition from our first experiences with painting as kids. One would expect the digital painting software to follow that intuition. Surprisingly, they do not. In today's painting software, blue and yellow mix into gray (see Fig. 1). This happens because the software represents the painting in RGB, which corresponds to additive mixing of colored lights. Indeed, when we shine blue and yellow light at the same spot, their spectra linearly sum together, and we get gray. Paints, however, mix in a very different way. They get their color from the mass of pigment particles which absorb and scatter the light in a complex fashion. That makes the outcome of pigment mixture highly non-linear (see Fig. 3). Blue and yellow making green is the most striking consequence. However, there are other interesting effects taking place. Unlike RGB, where colors tend to lose saturation when mixed with white, the saturation of actual paints momentarily increases, revealing the true nature of the pigment. On top of that, the hue can also change in a dramatic way. Phthalo Blue, for instance, shifts from purple to turquoise as we keep adding white (see Fig. 4).

These phenomena are reproduced with remarkable accuracy by the model of Kubelka and Munk [1931], who were the first to predict the behavior of pigment mixtures. Their work was introduced to the computer graphics community by Haase and Meyer [1992], who readily applied the Kubelka–Munk (K–M) model in the digital painting scenario. This launched an exciting line of research, where the authors of [Curtis et al. 1997; Baxter et al. 2004; Chu and Tai 2005; Haevre et al. 2007] achieved spectacular results applying K–M to realistic simulations of watercolors, oil paints, inks and pastels. Strangely enough, the K–M model has never been picked up by graphics software outside research. Even three decades since the publication of [Haase and Meyer 1992], blue and yellow make gray in today's painting software. Why?

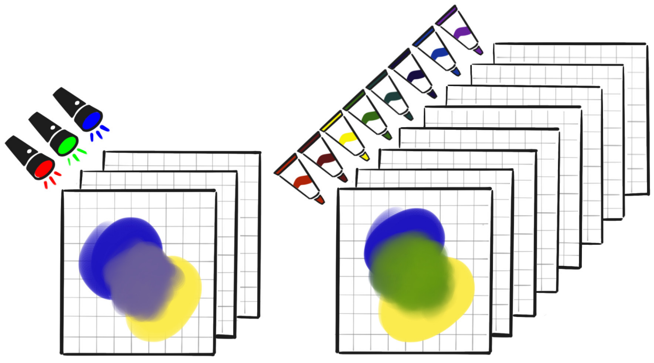

Fig. 2. Implementing K–M requires increasing the number of per-pixel channels to track pigment concentrations (right). This is in contrast to the 3-channel RGB representation (left), which is hardwired in today's software.

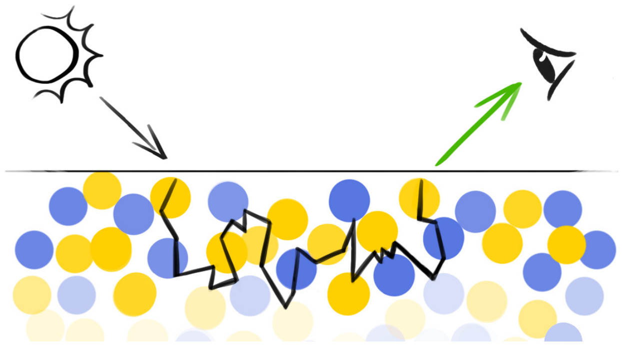

Fig. 3. The reason we get green when mixing blue and yellow paints is that their pigment particles repeatedly absorb and scatter the incoming light, leaving only the green wavelengths to come out of the mixture.

An explanation that came up from our discussions with the authors of *Rebelle* [Blaškovič 2016] is that the developers do not find the K–M model practical to implement. This is because K–M requires tracking the respective concentrations of all available pigments at each pixel [Baxter et al. 2004]. Another option is to track the per-wavelength absorption and scattering coefficients instead [Haase and Meyer 1992]. However, in both cases, the number of per-pixel channels grows substantially. In comparison, the contemporary painting software is usually built around the 3-channel RGB representation, and implementing K–M would require changing the software structure down to the core (see Fig. 2). In addition to that, displaying a painting on screen would require evaluating the visible-spectrum integrals per pixel, which has a considerable performance cost. Another drawback is that pigment mixtures do not cover the whole RGB gamut, which complicates the exchange with the outside world. For example, the user may want to paste a piece of artwork containings RGB colors that cannot be represented with pigments. It also prevents the user from picking an arbitrary RGB color to paint with, as they are used to from current software. It is for these practical reasons why the K–M model resisted adoption for so long in the graphics software industry.

Based on our discussion with Blaškovič et al., we put forward the requirements that any color-mixing model needs to meet in order to be practical for a real–world painting software:

- work directly on RGB without requiring additional channels,
- be fast enough to compute to avoid latencies during painting,
- handle all RGB colors without causing clipping or distortion.

Taking these considerations into account, we propose a new, practical approach to K–M-based pigment mixing which operates purely on RGB information. Our main contribution is a new color representation that enables mixing RGB colors as if they were made of actual pigments. Our latent representation can be manipulated with linear operations in the usual way, yielding expected, plausible results: taking an average of blue and yellow produces green. Importantly, our representation treats all colors homogeneously, including those RGB colors that cannot be mixed from pigments. We describe the pair of transforms that map between RGB colors and our latent representation, and we show they can be implemented with minimal overhead. Our contributions make it possible to integrate Kubelka–Munk into any painting software, solving the long-lasting issue of colors mixing wrong in digital painting programs. We make the implementation of our method available online: https://github.com/scrtwpns/pigment-mixing

2 BACKGROUND AND RELATED WORK

In order for color mixing to behave naturally in digital painting software, the mixing process should follow what happens in real life. The color sensation we experience when looking at the paint that comes out of the tube is caused by the interaction between incident light and pigment particles inside the paint. The pigment particles selectively absorb and scatter the light by a different amount at each wavelength. It is the difference in absorption and scattering properties that causes pigments to have distinct colors.

When we mix blue and yellow paint, the two kinds of pigments get dispersed together. When the light travels inside such a mixture, it is repeatedly absorbed and scattered by both the blue and the yellow particles along the way (see Fig. 3). Finally, some of the remaining light finds its way out of the mixture and reaches the eye. This is a process we are familiar with in computer graphics as subsurface scattering. It is this subsurface scattering phenomenon that causes the mixture of blue and yellow paint to appear green.

Over the years, different models were proposed to predict the outcome of this phenomenon [Klein 2010]. Perhaps the most successful one is the model of Kubelka and Munk [1931]. In order to predict the color of a pigment mixture with K–M, we need to know the absorption and scattering coefficients $K(\lambda)$ and $S(\lambda)$ of each pigment. These are functions of wavelength λ, and they need to be measured beforehand. Given the concentrations $\mathbf{c} = [c_1, \cdots, c_N]$ of the set of N pigments $\mathcal{P} = \{(K_i(\lambda), S_i(\lambda))\}_{i=1}^{N}$ entering the mix, the absorption and scattering of the resulting mixture is determined by a linear combination of its constituents [Duncan 1940]:

$$K_{mix}(\mathbf{c}, \lambda) = \sum_{i=1}^{N} c_i K_i(\lambda), \quad S_{mix}(\mathbf{c}, \lambda) = \sum_{i=1}^{N} c_i S_i(\lambda), \tag{1}$$

where the concentrations c_i are non-negative and sum to one.

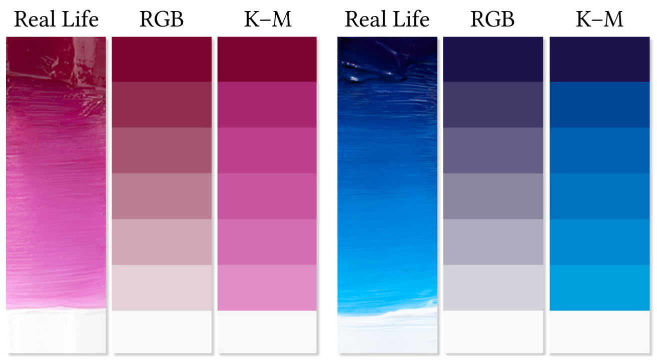

Fig. 4. Real paints gain saturation and shift in hue when mixed with white, as exhibited by Quinacridone Magenta (left) and Phthalo Blue (right). These effects are reproduced faithfully by the K–M model, unlike RGB, where colors tend to lose saturation when mixed with white.

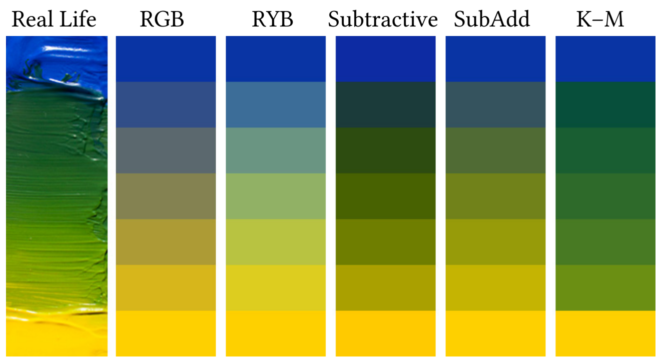

Fig. 5. Mixing Cobalt Blue and Hansa Yellow in real life vs. in RGB, RYB [Gossett and Chen 2004], Subtractive, Subtractive-Additive [Simonot and Hébert 2014] and K–M model. The ad-hoc approaches are unable to match the K–M's ability to predict the outcome of two pigments mixing together.

From here we use the K–M equation to predict the reflectance spectrum $R_{mix}(\mathbf{c}, \lambda)$ of the mixture. We assume a layer of paint so thick it completely hides the underlying substrate:

$$R_{mix}(\mathbf{c}, \lambda) = 1 + \frac{K_{mix}(\mathbf{c}, \lambda)}{S_{mix}(\mathbf{c}, \lambda)} - \sqrt{\left(\frac{K_{mix}(\mathbf{c}, \lambda)}{S_{mix}(\mathbf{c}, \lambda)}\right)^2 + 2\frac{K_{mix}(\mathbf{c}, \lambda)}{S_{mix}(\mathbf{c}, \lambda)}}. \quad (2)$$

To display R_{mix} on screen, we illuminate the mixture with a suitable light source and integrate the reflected spectrum with the CIE standard observer functions over the visible range of wavelengths $\lambda \in [380, 750]$. Here we use the D_{65} illuminant, which corresponds to average daylight and coincides with the sRGB white point:

$$X(\mathbf{c}) = \int_\lambda \bar{x}(\lambda) D_{65}(\lambda) R'_{mix}(\mathbf{c}, \lambda)\, d\lambda, \quad (3)$$

$$Y(\mathbf{c}) = \int_\lambda \bar{y}(\lambda) D_{65}(\lambda) R'_{mix}(\mathbf{c}, \lambda)\, d\lambda, \quad (4)$$

$$Z(\mathbf{c}) = \int_\lambda \bar{z}(\lambda) D_{65}(\lambda) R'_{mix}(\mathbf{c}, \lambda)\, d\lambda. \quad (5)$$

To account for surface reflection, we use the modified reflectance spectrum R'_{mix} obtained from Saunderson's [1942] equation:

$$R'_{mix}(\mathbf{c}, \lambda) = \frac{(1 - k_1)(1 - k_2) R_{mix}(\mathbf{c}, \lambda)}{1 - k_2 R_{mix}(\mathbf{c}, \lambda)}, \quad (6)$$

where k_1 and k_2 are the measured reflectance constants [Okumura 2005]. Finally, to obtain an sRGB color of the pigment mixture, we multiply the XYZ tristimulus values with the matrix of sRGB chromaticities and normalize by $Y_{D_{65}} = \int_\lambda \bar{y}(\lambda) D_{65}(\lambda)\, d\lambda$:

$$\underset{\mathcal{P}}{mix}(\mathbf{c}) = \begin{bmatrix} R(\mathbf{c}) \\ G(\mathbf{c}) \\ B(\mathbf{c}) \end{bmatrix} = \frac{1}{Y_{D_{65}}} \begin{bmatrix} +3.2406 & -1.5372 & -0.4986 \\ -0.9689 & +1.8758 & +0.0415 \\ +0.0557 & -0.2040 & +1.0570 \end{bmatrix} \begin{bmatrix} X(\mathbf{c}) \\ Y(\mathbf{c}) \\ Z(\mathbf{c}) \end{bmatrix}. \quad (7)$$

In summary, given a set of pigments $\mathcal{P}$, the function $mix(\mathbf{c})$ takes the input concentrations $\mathbf{c}$ and returns the predicted RGB color of the mixture. When displayed on a screen, this RGB color will evoke the same sensation in the eye as if we were looking at the pigment mixture in real life.

The computational machinery outlined in equations (1) – (7) was introduced to the computer graphics community by Haase and Meyer [1992] almost 30 years ago. Surprisingly, none of the widely used painting software actually implements it. Instead, they mix colors additively in RGB, which mimics the behavior of colored lights. That is why painting software lacks phenomena typical for pigments, such as hue shifts and saturation gains (see Fig. 4).

The most striking difference is that yellow and blue mix into gray instead of green. There are some exceptions where the developers recognized the issue and tried to mitigate it with some ad-hoc workaround. An example of such a workaround is the RYB mixing devised by Gossett and Chen [2004] and followed by [Chen et al. 2015; Sugita and Takahashi 2017]. They proposed an alternate color space, designed specifically to make the average of blue and yellow turn out green. They achieve this by interpolating eight hand-picked colors placed at the vertices of a 3D cube. As seen in Fig. 5, this approach produces colors that are far from the behavior of real pigments.

Instead of picking the colors by hand, Lu et al. [2014] took a data-driven approach, where they interpolate samples of real pigment mixtures taken from a provided example. However, their approach only works with a predetermined mixing ratio (e.g., 50:50), which is not flexible enough for the use in painting software, where colors need to be mixed in arbitrary ratios. In a related line of work Aharoni-Mack et al. [2017] and Tan et al. [2019] use the K–M model to extract a pigment palette from a supplied RGB image and decompose it into a collection of pigment channels. Shugrina et al. [2020] propose a reduced-order parametric model to fit the distribution of image colors. Although they achieve impressive results on image recolorization, these approaches are tailored to manipulating an existing artwork, which does not make them directly applicable to the digital painting scenario.

Another way to work around the yellow–blue issue is to perform subtractive color mixing, which models the behavior of overlaid color filters that selectively absorb light. In print, the CMYK color model uses subtractive mixing to predict the light absorption by the successive layers of cyan, magenta, and yellow ink. Although the

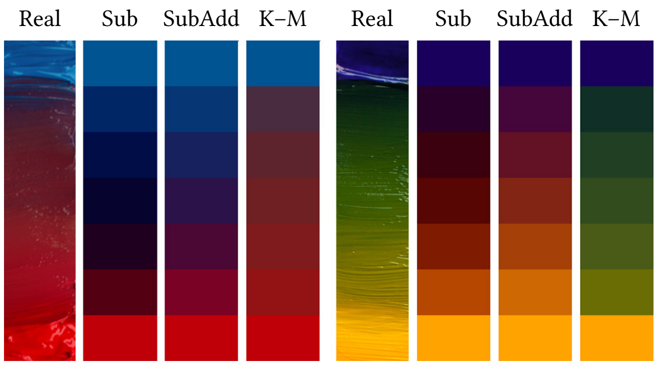

Fig. 6. Compared to the K–M model, subtractive mixing produces results that are either too dark (left) or have a wrong color (right). Subtractive–additive mixing improves the lightness but shares the improper color.

subtractive mixing does have the ability to produce green out of yellow and blue, the resulting color lacks the saturation achieved when mixing real paints (see Fig. 5). This is because subtractive mixing accounts only for the absorption of light, which is different from how the light interacts with actual pigment mixtures, where both absorption and scattering take place. The consequence is that the subtractive mixture turns out darker and has a different hue compared to real life (see Fig. 6). This is in contrast with the K–M model, whose predictions are in great agreement with reality. The shortcomings of subtractive mixing were recognized by Simonot and Hébert [2014] who derived hybrid subtractive–additive mixing formulas. While their formulation yields appreciably less dark results than purely subtractive mixing (see Fig. 5), it mispredicts the color in a similar way due to the shared subtractive component (see Fig. 6). Overall, none of the workarounds can match the K–M model in its ability to accurately predict the outcome of two paints mixing together.

When looking at today's painting software, it is evident that the developers went to great lengths to implement realistic brushes and believable media. Some software even include sophisticated solvers for simulating the dynamics of oils and watercolors. Yet, when it comes down to mixing paints, they all get the color wrong. This is in disregard of the body of work done by the research community, which clearly recognized the importance of using K–M to achieve realistic mixing of colors ([Curtis et al. 1997; Baxter et al. 2004; Chu and Tai 2005; Haevre et al. 2007]). Baxter et al. [2004] in particular made a clear case for using K–M in painting software and described the steps to implement it. Still, no painting software seems to implement K–M outside research. This is because the existing approaches require giving up the 3-channel RGB representation and resort to per-pixel tracking of pigment concentrations (like Baxter et al.) or store the full absorption and scattering spectra at each pixel (like Hasse and Meyer). Moving away from RGB is not practical from the developers' standpoint, as contemporary software is built around it, and the users expect certain features, like the ability to load an existing RGB image or to pick an arbitrary RGB color. This prompted us to devise a practically oriented color mixing solution that enables to use K–M in a real-world setting.

3 OUR APPROACH

To make the K–M pigment mixing practical, we need to make it work directly with RGB colors. This is the representation virtually every painting software works in. This means the work-in-progress painting is stored as per-pixel RGB triplets in memory. No other information is usually available (except for alpha). The most basic mixing operation used in today's painting software is linear interpolation, i.e., the "*lerp*":

$$\mathit{lerp}(\mathbf{RGB_1}, \mathbf{RGB_2}, t) = (1-t)\mathbf{RGB_1} + t\mathbf{RGB_2}. \tag{8}$$

The *lerp* takes two colors and a mixing parameter $t \in [0, 1]$, and produces the blended color as a linear combination of inputs. Most operations that involve color mixing, like blending, smudging, brush stamping, and compositing, can be reduced to a sequence of one or more applications of the *lerp* with suitably chosen t parameters.

Our aim now is to build an alternate version of *lerp*, which takes the same RGB inputs but produces a color that would result if we mixed actual paints, according to what the K–M model predicts. We call it the $\mathit{kmerp}(\mathbf{RGB_1}, \mathbf{RGB_2}, t)$.

The important point is that encapsulating the K–M mixing in this way makes it practical to integrate into existing software. The developers can keep the in-memory painting representation as-is, and only need to replace every occurrence of *lerp* with *kmerp* in their color mixing code. Crucially, they are not forced to upgrade from the 3-channel RGB representation to N-channels of pigment concentrations or per-wavelength tracking of absorption and scattering coefficients.

In order for *kmerp* to determine the outcome of mixing two RGB colors, it needs to know what pigments they are made of. Different combinations of pigments can be used to mix the same RGB color, each leading to a different outcome when mixed with another color. To resolve this, we need to decide on a set of primary pigments $\mathcal{P}^* = \{(K_i^*(\lambda), S_i^*(\lambda))\}_{i=1}^N$. This is the palette of pigments all the other colors will be mixed from.

The set of primary pigments is decided a priori, and the concrete choice is up to the *kmerp* user. For the general purpose, we use four modern organic pigments that provide a wide color gamut:

Phthalo Blue, Quinacridone Magenta, Hansa Yellow, and Titanium White, as suggested by Briggs [2007].

The main idea behind *kmerp* is to first decompose each RGB color into a mixture of these primary pigments, i.e., to *unmix* the input color into a vector of pigment concentrations $\mathbf{c}$. This amounts to inverting the K–M mixing procedure outlined in equations (1) – (7). We pose this inversion task as a least-squares optimization problem and solve for the unknown concentrations $\mathbf{c}$:

$$\underset{\mathcal{P}^*}{\mathit{unmix}}(\mathbf{RGB}) = \underset{\mathbf{c}}{\arg\min} \; ||\underset{\mathcal{P}^*}{\mathit{mix}}(\mathbf{c}) - \mathbf{RGB}||^2 \quad \text{s.t.} \;\; c_i \geq 0 \wedge \textstyle\sum_{i=1}^N c_i = 1. \tag{9}$$

Even though the *mix* is non-linear in $\mathbf{c}$, it is smooth and differentiable, which allows us to use a Newton-type solver to obtain the solution of the optimization problem (9).

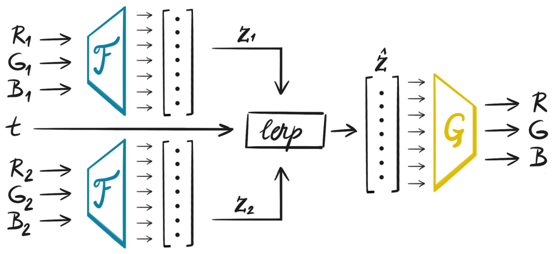

Fig. 7. To mix the two RGB colors, we first encode both colors into our latent representation using $\mathcal{F}$. Next, we mix the latent vectors using regular linear interpolation (*lerp*) and decode the result back to RGB using $\mathcal{G}$.

Equipped with the ability to *unmix* an RGB color, we can now put the rest of *kmerp* together. Given the two input colors $\mathbf{RGB}_1, \mathbf{RGB}_2$, we take a linear combination of their unmixed concentrations $\mathbf{c}_1, \mathbf{c}_2$ and *mix* the combined concentrations $\hat{\mathbf{c}}$ back to obtain the final RGB color:

$$\begin{aligned} \mathbf{c}_1 &= unmix(\mathbf{RGB}_1), \\ \mathbf{c}_2 &= unmix(\mathbf{RGB}_2), \\ \hat{\mathbf{c}} &= (1-t)\mathbf{c}_1 + t\mathbf{c}_2, \\ kmerp(\mathbf{RGB}_1, \mathbf{RGB}_2, t) &= mix(\hat{\mathbf{c}}). \end{aligned}$$

The *kmerp* meets our goal of having a color mixing device which takes RGB colors as input and produces the same answer as what the K–M model predicts. However, compared to *lerp*, it has a slight deficiency: it isn't guaranteed to reproduce the input colors when $t = 0$ or $t = 1$. This deficiency stems from the fact that there are RGB colors that cannot be mixed from the primary pigments. In fact, some RGB colors cannot be mixed out of any currently known pigments [Pointer 1980]. Such colors can only be made by shining light. For these RGB colors, the inverse of the *mix* function (7) does not exist. While the least-squares formulation (9) handles this gracefully by projecting the unattainable RGB color onto its closest mixable counterpart, this does have some negative implications for using *kmerp* in painting software.

Our next idea is to correct this deficiency by introducing an additive residual term. The residual represents the missing part of red, green, and blue light that needs to be supplemented to the light reflected off of the pigment mixture in order to exactly match the original RGB color. The residual is the difference between the RGB color and its projected counterpart: $\mathbf{RGB} - mix(unmix(\mathbf{RGB}))$. Since they essentially represent light, we can take the residuals $\mathbf{r}_1, \mathbf{r}_2$ of the input RGB colors and add their linear combination $\hat{\mathbf{r}}$ back to the result of $mix(\hat{\mathbf{c}})$. With this correction in mind, we revise the *kmerp* in a following way:

$$\mathbf{c}_1 = unmix(\mathbf{RGB}_1),\ \ \mathbf{r}_1 = \mathbf{RGB}_1 - mix(\mathbf{c}_1),$$

$$\mathbf{c}_2 = unmix(\mathbf{RGB}_2),\ \ \mathbf{r}_2 = \mathbf{RGB}_2 - mix(\mathbf{c}_2),$$

$$\hat{\mathbf{c}} = (1-t)\mathbf{c}_1 + t\mathbf{c}_2, \tag{10}$$

$$\hat{\mathbf{r}} = (1-t)\mathbf{r}_1 + t\mathbf{r}_2, \tag{11}$$

$$kmerp(\mathbf{RGB}_1, \mathbf{RGB}_2, t) = mix(\hat{\mathbf{c}}) + \hat{\mathbf{r}}.$$

The important thing to notice is that when the two input colors get combined together (Eqns. 10 & 11), the *kmerp* treats both the concentrations and the residuals in the same way. This leads us to the idea of establishing a homogeneous color representation by concatenating the pigment concentrations $\mathbf{c}$ and the RGB residuals $\mathbf{r}$ into a single latent vector $\mathbf{z} = \left[c_1\, c_2\, c_3\, c_4\, r_R\, r_G\, r_B\right]^\top$. We use $\mathcal{F}$ to encode an RGB color into the latent representation and $\mathcal{G}$ to decode it back:

$$\mathcal{F}(\mathbf{RGB}) = \begin{bmatrix}\mathbf{c}\\ \mathbf{r}\end{bmatrix} = \begin{bmatrix} unmix(\mathbf{RGB}) \\ \mathbf{RGB} - mix(\mathbf{c})\end{bmatrix} = \mathbf{z}, \tag{12}$$

$$\mathcal{G}(\mathbf{z}) = \mathcal{G}\left(\begin{bmatrix}\mathbf{c}\\ \mathbf{r}\end{bmatrix}\right) = mix(\mathbf{c}) + \mathbf{r}. \tag{13}$$

Borrowing from machine learning terminology, we call $\mathcal{F}$ the *encoder* and $\mathcal{G}$ the *decoder*. Our latent representation has two desirable properties:

- it represents all RGB colors in a homogeneous way, including those that cannot be mixed out of pigments $\mathcal{P}^*$,
- linear operations on latent vectors give expected, plausible results.

Note that unlike machine learning, where the latent representation is usually learned from data, here it is induced directly by the K–M model and the set of pigments $\mathcal{P}^*$. Also, while the latent space usually has less dimensions compared to input, here it is the other way: the latent space is 7D (4 concentrations + 3 residuals), while the input is the 3D RGB space.

With $\mathcal{F}$ and $\mathcal{G}$ at hand, the K–M-based pigment mixing can be easily plugged into any painting software. Each time two or more RGB colors need to be mixed in the host software, one simply encodes each participating color into our representation, applies the mixing operation on the corresponding latent vectors, and decodes the result back to RGB. Crucially, the whole operation is performed in an RGB-In / RGB-Out fashion (see Fig. 7), which makes it compatible with software that has the RGB representation hardwired.

With this scheme in mind, we can formulate the *kmerp* using $\mathcal{F}$ and $\mathcal{G}$ in the following way:

$$kmerp(\mathbf{RGB}_1, \mathbf{RGB}_2, t) = \mathcal{G}((1-t)\mathcal{F}(\mathbf{RGB}_1) + t\mathcal{F}(\mathbf{RGB}_2)). \tag{14}$$

The scheme extends analogously to situations where more than two colors mix together, like in bilinear interpolation, when applying convolution, or taking a weighted average of n colors in general:

$$\mathcal{G}\left(\frac{\sum_{i=1}^{n} w_i \mathcal{F}(\mathbf{RGB}_i)}{\sum_{i=1}^{n} w_i}\right).$$

Our latent representation makes it possible to mix RGB colors as if we were mixing real paints. Thanks to the residual part, our mixing approach generalizes even to RGB colors that cannot be mixed out of pigments. This makes it a viable drop-in replacement for the ordinary RGB mixing in digital painting software.

3.1 Surrogate Pigments

A practical issue with the pigment-based latent representation described above is that some pigment mixtures produce colors outside the sRGB gamut. For example, when mixing Phtahlo Blue and Titanium White in real life, at a certain point, we get a turquoise so saturated that no ordinary sRGB monitor is able to display it. In software, this issue manifests itself by the RGB values turning negative or greater than one in Eqn. 7.

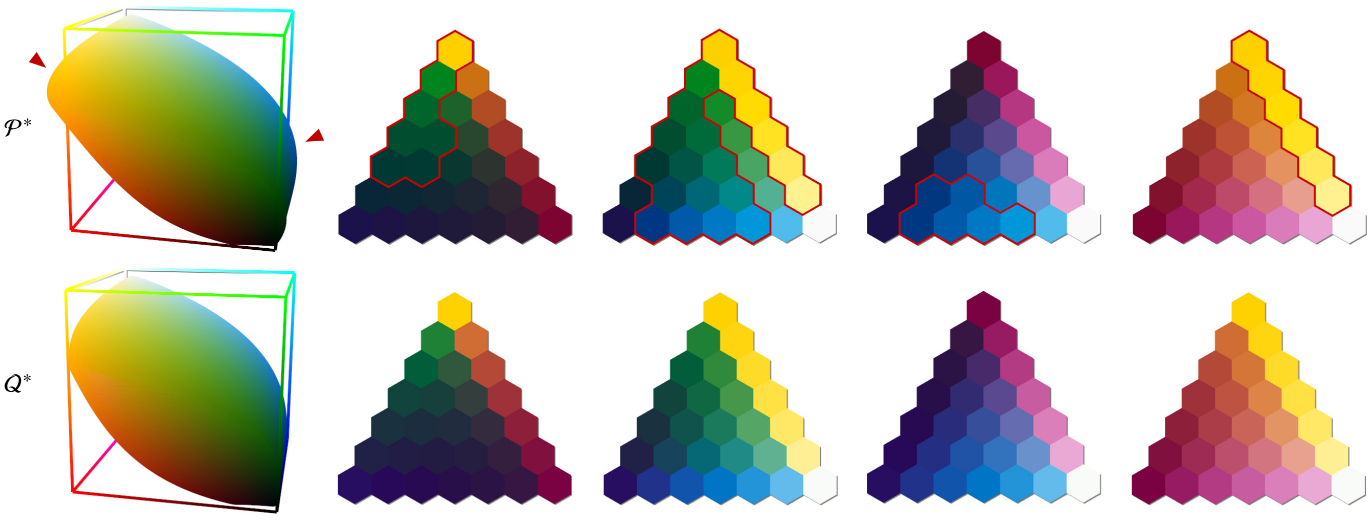

Fig. 8. The mixtures of $\mathcal{P}^*$ exceed the gamut of sRGB colors (top row). We mitigate this by optimizing a set of surrogate pigments Q^* which are confined to stay inside the RGB cube while being as close as possible to $\mathcal{P}^*$ (bottom row). The pyramids show 3-way mixtures between pigments placed at the vertices, with the outside-gamut colors highlighted in red. While the colors mixed from Q^* differ slightly from those mixed from $\mathcal{P}^*$, the change is mostly unnoticeable.

We may clamp the out-of-range values for a quick remedy, but that causes the mixing process to no longer be invertible. There are multiple combinations of the primary pigments that, after mixing and clamping, result in the same RGB color. Since RGB is the persistent representation of color that ultimately gets stored in memory, the clamping causes information to be lost.

One way to alleviate this issue is to perform gamut compression instead of clamping. The idea is to deform the in-gamut colors to make room for the outside-gamut colors and squeeze them in. Importantly, gamut compression is bijective, which preserves the invertibility and avoids the loss of information. Unfortunately, after experimenting with many gamut compression techniques [Morovič 2008], we came to the conclusion that they all cause too much distortion. The gamut compression manages to fit all mixed colors inside the RGB cube but causes the gradients between colors to no longer follow the trajectory predicted by K–M.

This prompted us to devise a different strategy: instead of compressing the colors after they have been already mixed, we tweak the primary pigments themselves to make their mixtures stay inside the RGB gamut in the first place. We call these modified pigments *surrogate*, denote them Q^*, and use them in place of the original pigments $\mathcal{P}^*$ in the encoder $\mathcal{F}$ and decoder $\mathcal{G}$. We formulate the task of tweaking Q^* as a non-linear optimization problem:

$$\begin{aligned} \underset{Q}{\arg\min} \quad & E_{push}(Q) + \alpha E_{pull}(Q, \mathcal{P}^*) \\ \text{s.t.} \quad & K(\lambda) > 0 \wedge S(\lambda) > 0 \quad \forall (K, S) \in Q, \end{aligned} \tag{15}$$

where the variables are the collected absorption and scattering coefficients of the four surrogate pigments $Q = \{(K_i^\dagger(\lambda), S_i^\dagger(\lambda))\}_{i=1}^4$. The mixtures of Q are spanning the surrogate color gamut Ω. The objective is to push Ω completely inside the RGB cube, while diverging as little as possible from the gamut of the original pigments $\mathcal{P}^*$. To that end, the term E_{push} penalizes the gamut boundary $\partial\Omega$ protruding outside the RGB cube:

$$E_{push}(Q) = \int_{\partial\Omega} \max(0, \phi(\underset{Q}{mix}(\mathbf{c})))^2 \, ds, \tag{16}$$

where ds is the element of the surrogate gamut boundary, and $\phi(\mathbf{p})$ is the signed distance between $\mathbf{p}$ and its closest point on a surface of the unit cube $[0, 1]^3$, having negative sign when $\mathbf{p}$ is inside the cube, otherwise being positive.

The second term pulls the boundary towards the gamut of the original pigments $\mathcal{P}^*$ by penalizing the deviations between colors mixed from Q and those mixed from $\mathcal{P}^*$:

$$E_{pull}(Q, \mathcal{P}^*) = \int_{\partial\Omega} ||\psi(\underset{Q}{mix}(\mathbf{c})) - \psi(\underset{\mathcal{P}^*}{mix}(\mathbf{c}))||^2 \, ds. \tag{17}$$

To obtain Q^* that differs from $\mathcal{P}^*$ in the least perceptible way, we take the color difference in the Oklab space [Ottosson 2020], which is a recent alternative to CIELAB with better perceptual properties. The function ψ takes care of converting the mixed RGB color to its Oklab counterpart.

The parameter α controls the relative strength of the pushing and pulling forces. To obtain the best Q^*, we start with $\alpha = 10^5$ and repeatedly solve the optimization problem (15), reducing α by half after each step. We use $Q^0 = \mathcal{P}^*$ as an initial guess and terminate as soon as Ω completely fits inside the RGB cube. We evaluate the integrals (16) and (17) numerically by sampling the gamut boundary $\partial\Omega$ at a number of quadrature points corresponding to equally spaced concentrations. We obtain the gradient of the objective with automatic differentiation [Griewank and Walther 2008] and use the L-BFGS-B algorithm [Byrd et al. 1995] to solve (15) in the loop.

This procedure yields surrogate pigments Q^* that are as close as possible to $\mathcal{P}^*$, but whose mixtures are never clipped as they do not exceed the RGB gamut. This guarantees that the round trip from RGB to latents and back is always invertible, which is what makes

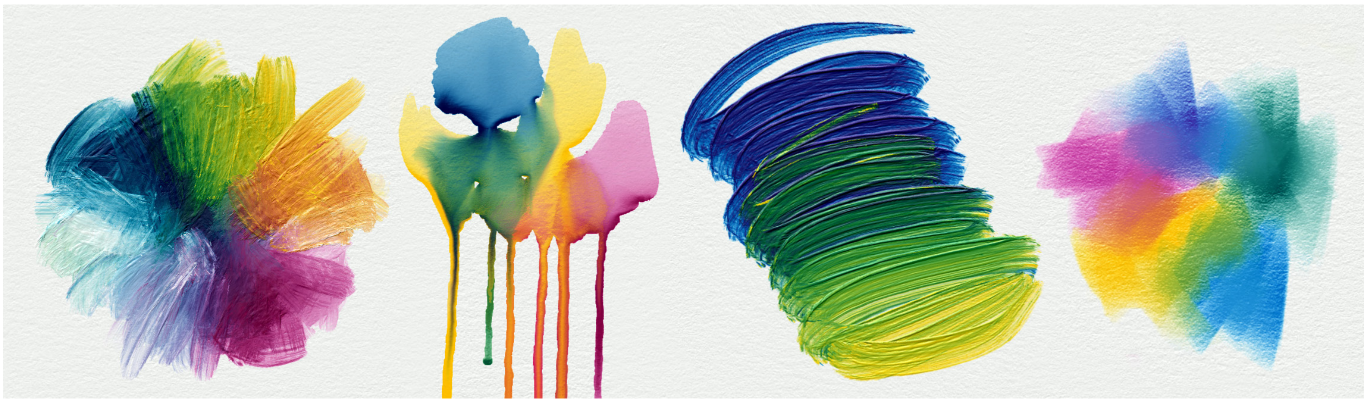

Fig. 9. This digital painting was done using our pigment-based color mixing approach. It was painted on an 8-bit RGB canvas with colors picked from a regular HSV wheel. Thanks to our mixing method, all simulated media such as oil paints, watercolor, or pastels look realistic and natural. The paints blend intuitively and produce vibrant secondary colors: orange, violet, and most critically, green.

our approach work. Figure 8 shows the comparison between the mixing behavior of the original pigments $\mathcal{P}^*$ and their surrogates $\mathcal{Q}^*$. Although some colors do change slightly, this does not pose an issue in practice since real pigments also show some variance between different batches and manufacturers. The important point is that each surrogate pigment preserves its characteristic behavior when mixed together with another pigment, as seen on the triangular gradients in Fig. 8. The bias introduced with surrogate pigments is barely noticeable, which we confirm later in Section 4.3 by looking at the statistics of color differences. Thanks to our surrogate-based approach, we manage to achieve the invertibility without sacrificing the K–M mixing behavior.

3.2 Implementation

To implement pigment-based color mixing into painting software, one has to integrate our encoder $\mathcal{F}$ and decoder $\mathcal{G}$ (Eqns. 12 & 13). Each time colors need to be mixed in the host software, a sequence of three steps is performed:

(1) encode each participating RGB color to its latent vector $\mathbf{z}$
(2) take a weighted average of the latent vectors
(3) decode the resulting latent back to obtain the final RGB color

We showed how this is done in the case of *kmerp* (14), and the procedure translates analogously to other situations.

In order for the painting experience to feel natural in software, it is important to minimize any latency between the user's action and the result appearing on screen. Since a large number of color-mixing operations needs to be performed at any given time, it is necessary for the evaluation of $\mathcal{F}$ and $\mathcal{G}$ to be fast. To that end, we use lookup tables and precompute the expensive operations as a preprocess, which makes $\mathcal{F}$ and $\mathcal{G}$ very fast at runtime.

The preprocess starts by choosing a set of primary pigments $\mathcal{P}^*$ and supplying their absorption and scattering coefficients $K(\lambda)$, $S(\lambda)$. These are either obtained by measuring the paint samples with a spectrophotometer or taken from a pre-existing database [Okumura 2005; Berns 2016]. For our purposes, we use four Golden Artist acrylic paints PB15:4, PY73, PR122, and PW6, whose coefficients we take from the Artist Paint Spectral Database [Berns 2016], together with the reflectance constants k_1, k_2 of the Saunderson's correction (6). The coefficients are sampled at wavelengths from 380 − 750 nm in 10 nm increments. Since the mixtures of $\mathcal{P}^*$ lie outside the RGB gamut, we find their surrogate pigments $\mathcal{Q}^*$ first and use $\mathcal{Q}^*$ in place of $\mathcal{P}^*$ in all the subsequent steps. Since the coefficients are sampled at 36 wavelengths, this amounts to solving the optimization problem (15) in 288 variables.

The most expensive operation associated with our latent representation is the evaluation of the *unmix* function (9) inside the encoder $\mathcal{F}$, which involves running a quasi-Newton solver for each input RGB color. Here we use automatic differentiation together with L-BFGS-B to obtain the 4 unmixed concentrations, starting with $\mathbf{c}^0 = (0.25, 0.25, 0.25, 0.25)$ as an initial guess. Unmixing an RGB color this way can take up to 100 milliseconds to converge. We speed this up by precomputing the *unmix* function for all 8-bit RGB colors and store the results in a 3D look-up table. We quantize the unmixed concentrations to 8-bits and further reduce the table footprint by storing only 3 concentrations instead of 4. Since the concentrations sum to one, the fourth concentration is implicitly $c_4 = 1 - (c_1 + c_2 + c_3)$. The final 256^3 lookup table occupies 48 MB of memory.

We further speed up $\mathcal{F}$ and $\mathcal{G}$ by using a second lookup table to precompute the results of the *mix* function (7) in a similar fashion. Thanks to the fact the fourth concentration is implicit, we can use a 3D table instead of 4D, which makes the approach feasible. The expensive part of the *mix* function are the tristimulus integrals (3) – (5) which we compute from the 36 spectrum samples using the trapezoidal rule. We quantize the concentrations to 8-bits and store the mixed RGB colors in another 48 MB 256^3 table.

The two precomputed tables occupy 96 MB in total. To reduce their on-disk storage, we compress each table in a lossless way by tiling its slices into a 4096×4096 PNG image. The two PNG images have 7 MB in total, which is lightweight enough to bundle them together with the software. With the precomputed tables at hand, the evaluation of $\mathcal{G}$ reduces to performing a trilinear table lookup followed by vector addition. The evaluation of $\mathcal{F}$ reduces to two

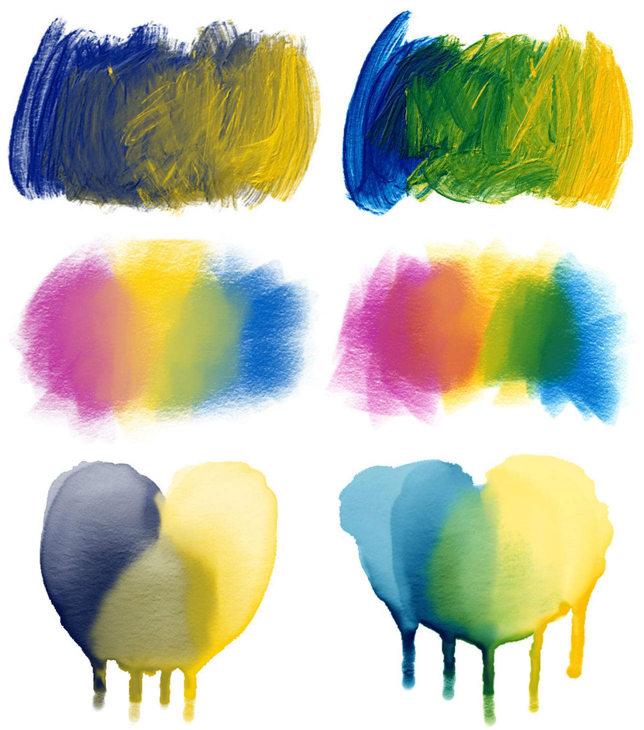

Fig. 10. The effect pigment-based color mixing has on traditional media simulation. When mixed in RGB, paints blend into grayish, dull colors, while our method yields intuitive, natural shades and brings the simulation of oil paints, pastels, watercolor, and other media to the next level.

trilinear lookups followed by a subtraction. This poses a reasonable overhead compared to the usual RGB mixing, making the approach practical for current painting software.

3.3 Choice of Primary Pigments

The set of primary pigments $\mathcal{P}^*$ has a central role in our color mixing approach. Because we interpret all RGB colors in terms of $\mathcal{P}^*$, the specific choice of pigments affects the resulting mixing behavior. In Section 3, we picked Phthalo Blue, Quinacridone Magenta, Hansa Yellow, and Titanium White as primary pigments, and we recommend those as a good default. However, our method is not restricted to this particular choice, and one can use different pigments instead. This is important from the artist's standpoint since the choice of pigments is an intimate part of their approach and an expression of how they see the world. Ideally, the host software should allow the user to choose their preferred set of pigments in advance, pre-computing the necessary look-up tables before starting the painting.

The primary pigments are usually selected from the yellow, red, and blue pigment categories. Within each category, the pigments come in various warm and cool hues, each having its specific color bias. In our approach, the set of primary pigments is limited to four entries, with one slot reserved for white. The main consequence is that we can only choose either the warm or the cool version of a pigment from each category. I.e., we cannot afford to have both

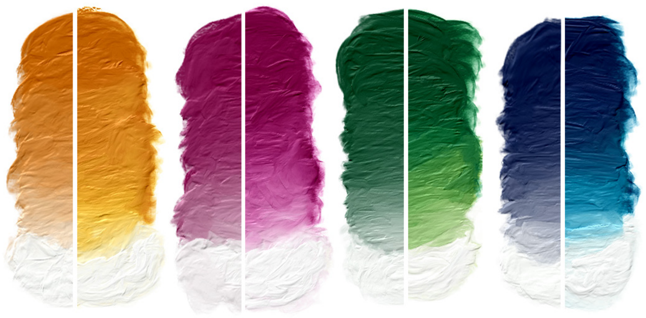

Fig. 11. When mixing paints with white in RGB, their colors get desaturated and dimmed (left). In contrast, our method mimics saturated hue shifts from real life and keeps the gradients radiant (right).

warm and cool blue, as that would take up two of the three available slots. Therefore, the final behavior of our mixing method depends on the particular kind of pigment we choose for each primary. Their characteristic properties will get baked in, and interactions between RGB colors will behave accordingly. For example, if we pick a warm primary blue, such as Ultramarine, most blue-hued RGB colors will inherit its warm nature and mix with other RGB colors in a similar fashion.

Besides mixing behavior, the choice of pigments also bakes in a specific appearance of the medium. That depends on the kind of paints used to acquire the K & S coefficients. In our case, we used the coefficients measured from acrylic paints, which are smooth and glossy. The gloss provides the colors with a rich and saturated appearance, and our method bakes in this characteristic. That can cause it to overestimate saturation when applied in the context of a different medium. For example, even though watercolor and acrylic paints share the same pigments, dried watercolors should look comparatively duller due to the absence of gloss. Therefore, when applied to a simulated watercolor, our method can produce colors that look more saturated than they would in real life. Although this can be remedied by applying a desaturating color correction in a post-process, we have not found this necessary in practice.

4 RESULTS

To prove our approach viable, we collaborated with Escape Motions who integrated our mixing method into their painting software *Rebelle*. In Fig. 9 you can see an example painting made using our pigment-based color mixing approach. We painted this image on an 8-bit RGB canvas with colors picked from a regular HSV wheel. Notice how all paints blend naturally and stay vivid even after being repeatedly mixed. This painting looks realistic partially thanks to the convincing media simulation, but natural color mixing brings it to the next level.

To appreciate the difference a natural color mixing makes in media simulation, we compare paints mixed additively in RGB to the same paints mixed using our approach in Fig. 10. We show mainly the critical combination of blue mixing with yellow on three different techniques – oil paints, pastels, and watercolor. The outcome from the additive mix is always grayish and dull while our approach gives

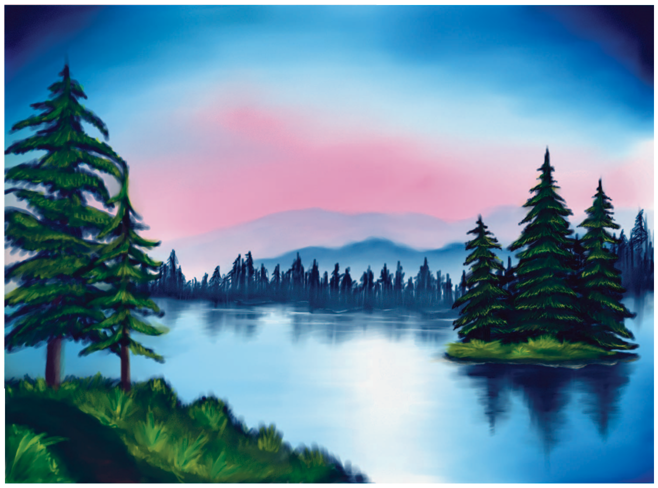

Fig. 12. Landscape painting made for the purpose of profiling the performance of our color-mixing approach. The whole image was done using four colors – dark blue, magenta, yellow, and white, all picked from a regular HSV wheel.

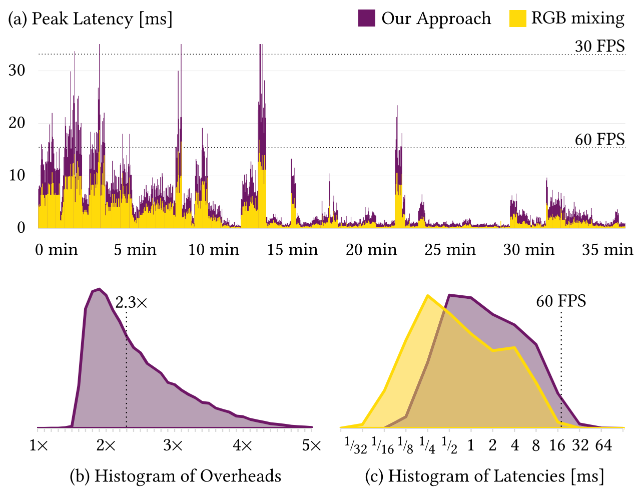

Fig. 13. The latency and overhead of our color-mixing approach compared to ordinary RGB mixing.

a natural shade of green color. Another combination we show is magenta and yellow, painted with pastels in the middle row. The orange color they produce with additive mixing is desaturated and weak, while our approach yields a radiant, fiery orange, as we would expect in real life.

Another effect that contributes to the realistic feel is the pronounced hue shifts and increased saturation when colors mix with white. Notice in Fig. 11 how especially the lower halves of the gradients differ. Additive mixing causes the colors close to white to look grayish and dim, while our approach keeps colors lively throughout the whole gradient.

4.1 Performance

To evaluate the performance of our method in a real-world setting, we record a sequence of brush strokes made over the course of a painting session. This allows us to replay them afterward with different color-mixing implementations and compare the timings. Our primary interest is to quantify the amount of lag incurred with our pigment-based mixing, i.e., the time it takes for the painted stroke to catch up with the motion of the stylus. With compute-heavy approaches, the lag can easily build up when the user is making fast brush strokes, which can negatively impact the painting experience.

To gather the recorded brush strokes, we asked an artist to paint a landscape following the wet-on-wet technique of Bob Ross, which relies on the paints mixing together on canvas (see Fig. 12). To better isolate the overhead of color mixing, the simulation of media dynamics was turned off, and the painting was deliberately made using only two simple tools: a soft round brush and an area-averaging smudge brush. We replay the recorded brush strokes with two concurrent implementations: the usual RGB mixing and our K–M-based pigment mixing, both implemented in a single-threaded plain C code. The strokes are replayed in real-time, and we measure the latency $t_2 \quad t_1$ between the time t_1 when the latest stylus position was registered and t_2 when the painted stroke first reached up to this position. We plot the peak latency encountered during each second of the painting in Fig. 13a. The painting session lasted 38 minutes, during which the artist placed 2915 brush strokes of varying diameter, starting with large brushes for the background and transitioning to smaller brushes when adding details. The painting was made in a 2340 × 1654 resolution, and the performance was measured on a laptop with Intel Xeon E-2276M CPU clocked at 2.8 GHz.

The plot of peak latencies shows that the performance of our approach follows that of RGB mixing with about 2× overhead, spiking above 20 ms on a few occasions when the artist used a large smudging brush. Overall, painting the brush strokes with our method is 2× – 3× slower compared to RGB mixing, with the median overhead of 2.3× (Fig. 13b). Crucially, when looking at the histogram of all latencies (Fig. 13c), we see that both methods manage to catch up with the stylus in less than 16 milliseconds most of the time. This is quicker than the time it takes the 60 Hz display to refresh, which makes our overhead unnoticeable in practice. Therefore, mixing the colors with our approach does not introduce any appreciable lag that would negatively affect the artist.

4.2 Effect of Residuals

While our method achieves perfect results for RGB colors that can be mixed out of primary pigments, the colors outside the pigment gamut need to be supplemented with a non-zero residual. To assess the impact of the residual term, we examine the gradients between extremal RGB colors, which are farthest away from the pigment gamut. As can be seen in Fig. 14, our method shows a reasonable behavior even in the presence of significant residuals, yielding plausible mixing results. The gradients follow smooth trajectories without going through unexpected colors.

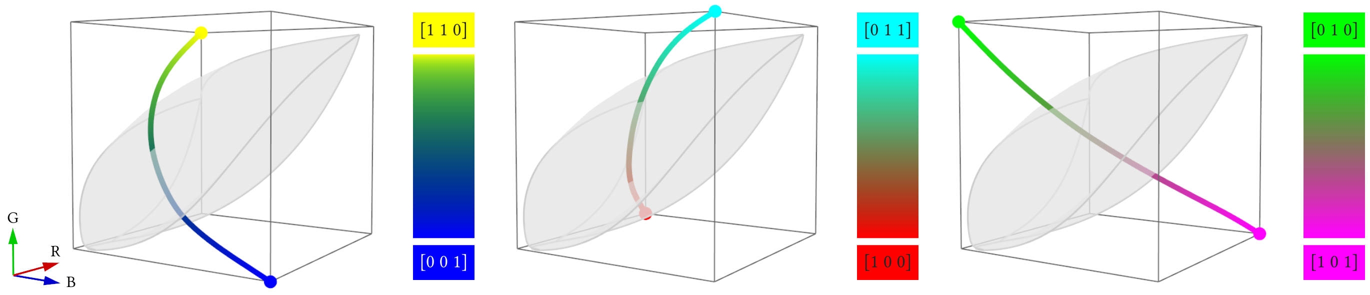

Fig. 14. Mixing behavior in the presence of residuals. The colored curves show the RGB trajectories that result when mixing between extremal RGB colors. The endpoint colors are away from the pigment gamut (shown in gray), which necessitates the participation of the residual term. Despite the presence of significant residuals, the resulting gradients follow smooth, plausible trajectories without causing unexpected colors to appear. Unlike linear RGB mixing, which follows straight-line paths in the RGB space, our approach produces curved trajectories due to the non-linearity of the underlying K–M model.

4.3 Bias of Surrogate Pigments

Since the primary pigments $\mathcal{P}^*$ protrude outside the RGB gamut, we need to replace them with surrogate pigments Q^* in order to achieve invertible conversion between RGB colors and our latent representation. The mixtures of Q^* produce slightly different colors compared to $\mathcal{P}^*$. In order to quantify the amount of bias caused by replacing $\mathcal{P}^*$ with Q^*, we sample 10^6 concentrations $\mathbf{c}$ and gather the histogram of perceptual differences between colors mixed from $\mathcal{P}^*$ and those mixed from Q^*:

$$\Delta E_{00}(\underset{\mathcal{P}^*}{mix}(\mathbf{c}), \underset{Q^*}{mix}(\mathbf{c})),$$

where ΔE_{00} is the CIEDE2000 color difference formula [CIE 2001]. The resulting histogram is plotted in Fig. 15. The histogram peaks around ΔE level of 1 which corresponds to just noticeable difference (JND), and falls rapidly after that. This confirms that the surrogate pigments do not introduce considerable bias. In fact, the difference between colors mixed from $\mathcal{P}^*$ and Q^* is typically close to the threshold of being unnoticeable.

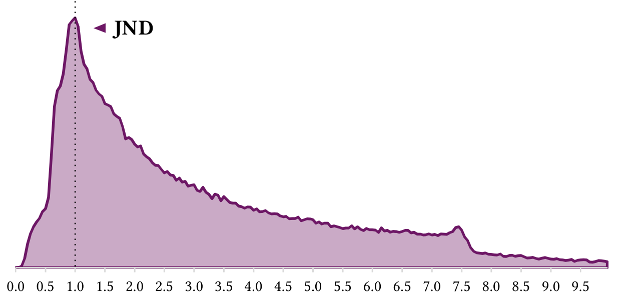

Fig. 15. Histogram of perceptual color differences between colors mixed from $\mathcal{P}^*$ and Q^*.

5 LIMITATIONS AND FUTURE WORK

The main restriction of our approach is that the number of primary pigments is limited to four. A practical reason is that we need to use look-up tables for quick conversion between latent vectors and RGB values. If there were more than four pigments, the use of the 3D table would no longer be possible, and the real-time performance would suffer. Another reason is that a fifth pigment would cause ambiguities. There would be more ways to mix the primary pigments into the same RGB color. For example, if we add Viridian Green as the fifth pigment, we can now obtain certain green hues in two possible ways: either by mixing blue and yellow, or by using the green pigment in its pure form. This ambiguity would cause the conversion between pigment concentrations and RGB values to no longer be invertible and our method would cease to work. The four-pigment-limitation can be alleviated to some extent by precomputing multiple versions of the lookup tables for different sets of primary pigments and switching between them on the fly. However, the question how to properly generalize our approach to a case of more than four pigments is an interesting avenue for future work.

Another limitation is that our approach only considers homogeneous mixing of opaque paints. When compositing two RGB layers using alpha-blending, our approach treats both layers as if they consisted of thick wet paint. A possible extension left for future work is to add support for the Kubelka–Munk layer-compositing model, which would enable more faithful simulation of translucent layers and handle effects like watercolor glazing.

6 CONCLUSION

We introduced a practical method to achieve natural color mixing that follows the behavior of real paints. Our K–M-based mixing works in RGB-In / RGB-Out fashion, which makes it easy to integrate in current software that has the RGB representation hardwired. Thanks to being compatible with all RGB colors, our method does not deprive the users of the ability to work with arbitrary RGB images and allows them to keep using their favorite color pickers. Since our color mixing can be performed with just a few table lookups, it does not introduce any noticeable lag during painting compared to ordinary RGB mixing. The practical side of our approach was confirmed by a painting software vendor, who integrated it into their product within a few weeks. Thanks to that, the first few artists got the opportunity to try it out and share their thoughts with us.

According to them, having a natural color mixing available in painting software is highly desirable because it allows one to carry the intuitions gained in real life over to the digital domain. This is

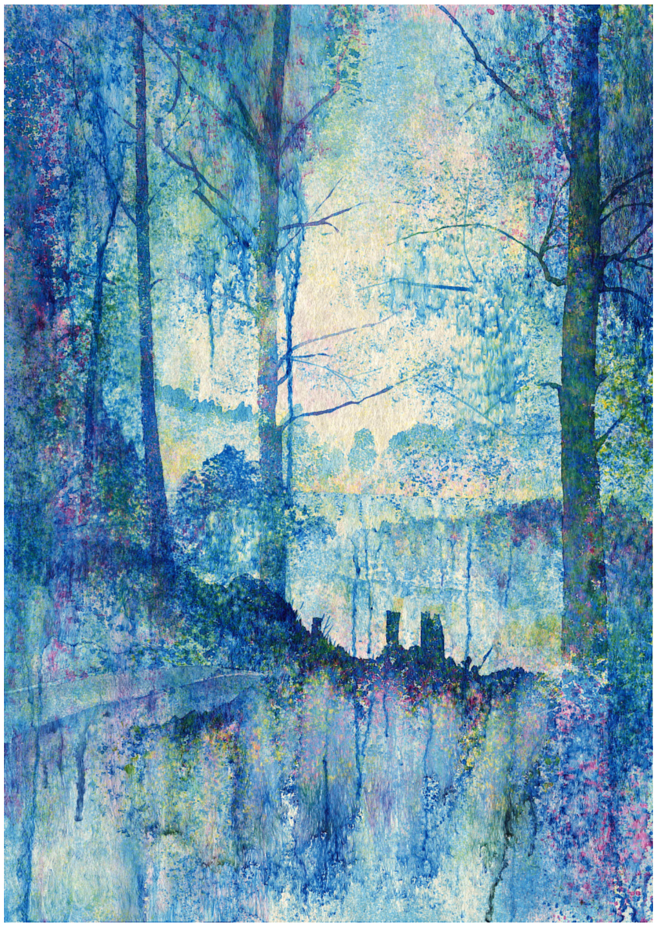

Fig. 16. Watercolor painting made in Rebelle using our color mixing method. Thanks to the faithful pigment behavior, the waterdrops dissolved into vivid gradients and blended naturally. As a result, the painting process was intuitive, enjoyable, and felt realistic. ©Ľubomír Zabadal, Constantine the Philosopher University in Nitra, Slovakia.

especially important for classically trained artists, who often find the color behavior in painting software unintuitive. Thanks to our K–M-based mixing, the digital tools suddenly become accessible to artists who are proficient in traditional media (see Fig. 16). It allows them to extend their craft to a digital setting while enjoying some of its benefits, like the ability to undo and redo. Digitally trained painters, on the other hand, are already used to the RGB mixing behavior. However, they need to consciously work against it to avoid the regression toward the gray, which takes away from their creative process. Unlike RGB where colors mix in linear fashion, pigment-based mixing has the advantage of non-linear behavior, which causes the mixed colors to naturally bend away from gray, producing secondary hues while preserving saturation. This makes our K–M-based color mixing desirable even to the artist who is already skilled in RGB painting.

We believe this paper will finally put an end to the long-standing issue of colors mixing wrong in digital painting software.

ACKNOWLEDGMENTS

The authors wish to thank: Peter Blaškovič and the Escape Motions team for inspiring discussions and for bringing the problem to our attention; Pavol Obuch and Michal Fapšo for their effort integrating our method into Rebelle; Ľubomír Zabadal, Jakub Javora, and Markéta Kolářová for sharing their opinions and artistic skills; Daniel Sýkora and the anynomous reviewers for their insightful comments and suggestions. This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS19/179/OHK3/3T/13 (Research of Modern Computer Graphics Methods), and by the Research Center for Informatics, grant No. CZ.02.1.01/0.0/0.0/16_019/0000765.

REFERENCES

Elad Aharoni-Mack, Yakov Shambik, and Dani Lischinski. 2017. Pigment-Based Recoloring of Watercolor Paintings. In *Proceedings of NPAR '17*. Article 1.

William Baxter, Jeremy Wendt, and Ming C. Lin. 2004. IMPaSTo: A Realistic, Interactive Model for Paint. In *Proceedings of NPAR '04*. 45–56.

Roy S. Berns. 2016. Artist Paint Spectral Database. In *Proceedings of CIC24*.

Peter Blaškovič. 2016. Rebelle: Real Watercolor and Acrylic Painting Software. In *Proceedings of SIGGRAPH '16: ACM SIGGRAPH 2016 Appy Hour*. Article 3.

David Briggs. 2007. *The Dimensions of Colour*. http://www.huevaluechroma.com

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing* 16, 5 (1995), 1190–1208.

Zhili Chen, Byungmoon Kim, Daichi Ito, and Huamin Wang. 2015. Wetbrush: GPU-based 3D Painting Simulation at the Bristle Level. *ACM Transactions on Graphics* 34, 6, Article 200 (2015).

Nelson S.-H. Chu and Chiew-Lan Tai. 2005. MoXi: Real-Time Ink Dispersion in Absorbent Paper. *ACM Transactions on Graphics* 24, 3 (2005), 504–511.

CIE. 2001. *Improvement to industrial colour-difference evaluation.* Central Bureau of the CIE, Vienna, Austria.

Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. 1997. Computer-Generated Watercolor. In *Proceedings of SIGGRAPH '97*. 421–430.

D. R. Duncan. 1940. The colour of pigment mixtures. *Proceedings of the Physical Society* 52 (1940), 390–400.

Nathan Gossett and Baoquan Chen. 2004. Paint Inspired Color Mixing and Compositing for Visualization. In *Proceedings of InfoVis '04*. 113–118.

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, PA, USA.

Chet S. Haase and Gary W. Meyer. 1992. Modeling Pigmented Materials for Realistic Image Synthesis. *ACM Transactions on Graphics* 11, 4 (1992), 305–335.

William Van Haevre, Tom Van Laerhoven, Fabian Di Fiore, and Frank Van Reeth. 2007. From Dust Till Drawn. *The Visual Computer* 23, 9–11 (2007), 925–934.

Georg A. Klein. 2010. *Industrial Color Physics.* Springer, New York, NY, USA.

Paul Kubelka and Franz Munk. 1931. Ein Beitrag zur Optik der Farbanstriche. *Zeitschrift für Technishen Physik* 12 (1931), 593–601.

Jingwan Lu, Stephen DiVerdi, Willa Chen, Connelly Barnes, and Adam Finkelstein. 2014. RealPigment: Paint Compositing by Example. In *Proceedings of NPAR*. 21–30.

Ján Morovič. 2008. *Color Gamut Mapping.* Wiley, Chichester, West Sussex, UK.

Yoshio Okumura. 2005. *Developing a Spectral and Colorimetric Database of Artist Paint Materials.* Master's thesis. Rochester Institute of Technology.

Björn Ottosson. 2020. *A perceptual color space for image processing.* https://bottosson.github.io/posts/oklab

Michael R. Pointer. 1980. The Gamut of Real Surface Colours. *Color Research and Application* 5, 3 (1980), 145–155.

J. L. Saunderson. 1942. Calculation of the color of pigmented plastics. *Journal of the Optical Society of America* 32 (1942), 727–736.

Maria Shugrina, Amlan Kar, Sanja Fidler, and Karan Singh. 2020. Nonlinear Color Triads for Approximation, Learning and Direct Manipulation of Color Distributions. *ACM Transactions on Graphics* 39, 4, Article 97 (2020).

Lionel Simonot and Methieu Hébert. 2014. Between additive and subtractive color mixings: intermediate mixing models. *Journal of the Optical Society of America A* 31, 1 (2014), 58–66.

Junichi Sugita and Tokiichiro Takahashi. 2017. Computational RYB Color Model and its Applications. *IIEEJ Transactions on Image Electronics and Visual Computing* 5, 2 (2017), 110–122.

Jianchao Tan, Stephen DiVerdi, Jingwan Lu, and Yotam Gingold. 2019. Pigmento: Pigment-Based Image Analysis and Editing. *IEEE Transactions on Visualization and Computer Graphics* 25, 9 (2019).