LoCoPalettes: Local Control for Palette-based Image Editing

Cheng-Kang Ted Chao\textsuperscript{1}, Jason Klein\textsuperscript{2}, Jianchao Tan\textsuperscript{3}, Jose Echevarria\textsuperscript{4}, Yotam Gingold\textsuperscript{1}

\textsuperscript{1}George Mason University, USA  
\textsuperscript{2}Cornell University, USA  
\textsuperscript{3}Kuaishou Technology, China  
\textsuperscript{4}Adobe Research, USA

Figure 1: LoCoPalettes augments palette-based image editing with image-space constraints and semantic hierarchies. Left: The user places a spatial constraint on the input image to make the leaves brown. LoCoPalettes optimizes for a palette with respect to users’ image-space constraints, shown as small white circles (Sec. 4.2). Center: The optimization updates the global palette. Right: The user places a second spatial constraint on the grass. If constraints cannot be satisfied with a single global palette, LoCoPalettes semantically segments the image and activates a palette hierarchy $H$ to achieve the color constraints using local palettes (Sec. 4.4). The grass constraint’s semantic segmentation mask is shown in the lower-right.

Abstract

Palette-based image editing takes advantage of the fact that color palettes are intuitive abstractions of images. They allow users to make global edits to an image by adjusting a small set of colors. Many algorithms have been proposed to compute color palettes and corresponding mixing weights. However, in many cases, especially in complex scenes, a single global palette may not adequately represent all potential objects of interest. Edits made using a single palette cannot be localized to specific semantic regions. We introduce an adaptive solution to the usability problem based on optimizing RGB palette colors to achieve arbitrary image-space constraints and automatically splitting the image into semantic sub-regions with more representative local palettes when the constraints cannot be satisfied. Our algorithm automatically decomposes a given image into a semantic hierarchy of soft segments. Difficult-to-achieve edits become straightforward with our method. Our results show the flexibility, control, and generality of our method.

CCS Concepts

- Computing methodologies → Image processing;

1. Introduction

In palette-based image editing approaches (\cite{CFL*15} and follow-up work), a representative palette is extracted from an image. Users manipulate the colors of the palette to edit the image. This allows users to perform fast and simple global edits. However, to recolor a specific object of interest, users must often iteratively adjust multiple palette colors, since the object itself may not be directly represented in the palette. In many scenarios, recoloring only a specific
region with a palette-based editing approach is impossible, since colors alone do not reflect semantic information (Figures 1–3).

In this paper, we focus on extending the usability of geometric palettes (e.g., [TLG16]). Leveraging recent previous work [CKT*23], LoCoPalettes solves for an as-sparse-as-possible RGB palette change that respects both image-space color constraints and palette constraints in real-time. Our key contribution is that, when the user’s constraints cannot be satisfied, LoCoPalettes automatically splits the image into semantic sub-regions with local palettes (Fig. 1). Given an input image, LoCoPalettes automatically computes a hierarchical semantic soft segmentation, extracts local palettes and weights for each node, and computes optimal palette transformations to propagate changes from parent to child nodes. We demonstrate LoCoPalettes with a variety of examples impossible to achieve with purely palette-based editing. Code for this work can be found at https://github.com/tedchao/LoCoPalettes.

2. Related Work

Image recoloring is a common task performed by digital artists. There are many approaches to image recoloring, including methods based on examples, scribbles, and palettes. Example-based recoloring methods transfer style and color characteristics from one image to another. The pioneering work of [RAGS01] performs statistical analysis in LAB-space to transfer colors between images. [TTT05] uses local color transfer between regions of pair of images by Gaussian mixture models with augmented spatial smoothness and color consistency. Recently, using semantic features from deep neural networks [LLY*17, HLC*19] has also shown high-quality color transfer. Unlike example-based recoloring, scribble-based recoloring [LLW04, AP08, LJJH10] does not require a reference image; rather, users edit the image directly by making rough color scribbles, which define edits that are propagated to pixels with similar intensities or colors.

Palette-based recoloring was first introduced by [CFL*15], who suggested extracting palettes through color clustering on a given image and performing recoloring using radial basis function-weighted color transformations. One set of follow-up work focuses on geometric palettes in RGB and LAB-space. [TLG16, TEG18a, TEG18b, WLX19, CKT*23] extract geometric palette in RGB- and LAB-space via convex hull simplification. [TLG16] suggested decomposing the image into layers for over compositing via non-linear optimization. Other approaches [TEG18a, WLX19] target additive mixing weights, which they compute in a spatially coherent manner. [TEG18a] proposed an efficient and direct algorithm making use of RGBX- and Y-space. We adopt it in our weights computation. [WLX19] proposed a post-process to make geometric palettes more compact, representative, and less sensitive to outliers. We could make use of their approach to improve our palettes. One recent approach suggested to extract palettes in a 2D color space (LAB’s AB dimensions) [CKT*23] and provide separate control over lightness. Notably, they solve for palette colors that satisfy image-space color constraints. We improve these approaches with sparser weights (Sec. 4.1) in the more commonly used RGB-space, though our approach could be used with any technique for weight computation from a geometric palette. We also adapt the palette optimization from [CKT*23], extending it to our hierarchical palettes in RGB-space. [TDLG19] and [AMSL17] proposed to decompose RGB images using the Kubelka-Munk [BM11] physical pigment mixing model, which may be more intuitive for artists most familiar with physical pigments, but adds significant overhead to the compositing process. [GS20] divided RGB-space into multiple regions, employing various geometric techniques to separate pixel colors depending on their position within the RGB-space. [ZNZ*21] formulated an optimization solving for palette colors and mixing weights simultaneously by considering color separation priors. [JYS19] create palettes in a hierarchical, bottom-up manner. These are quite different from the image-space and palette-space hierarchies we use to support local editing.

Approaches for unmixing colors, such as those proposed in [AAPS16, AASP17], involve minimizing an energy to identify a small number of sparse layers with nearly homogeneous colors. More recently, [AZIA20, HAS*22] solve the unmixing problem using neural networks to achieve fast performance. While the soft color layers generated by unmixing approaches allow users to perform various fast global edits, such as compositing, recoloring is more challenging without homogeneous layer colors. Unmixing approaches are a form of color-based soft segmentation or image matting [LLW07]. [AOP18] solves the soft matting problem by optimizing an energy using high-level semantic features with color and texture to decompose an image into semantically meaningful segments. We evaluated these approaches, but found that adding softness (Sec. 4.3.2) to a recent panoptic (hierarchical) segmentation algorithm [CMS*20] was more robust.

Though palette-based recoloring methods are simple to use and practical, they only allow for additive mixing weights and do not reflect semantic information. This makes them limited in their ability to achieve more complex recoloring results. Our approach addresses these limitations by providing a more intuitive way to manipulate image colors while respecting both image-space and palette-space constraints. This leads to more natural and realistic recoloring results, which can be applied to a wide range of digital art applications. We believe that our approach will be a valuable tool for digital artists and image editors, providing them with a more effective way to manipulate image colors.
fast to compute, they lack the ability to directly recolor specific objects or areas of interest with a target color. Two approaches [ZYL21, CZYL22] solve this problem with learning-based region selection followed by a recoloring step to perform natural color adjustments. [CKT*23] formulate a sparsity loss to solve for palette changes and lightness curves that satisfy users’ constraints on pixel colors. However, the approach does not allow recoloring a specific semantic object of interest. A local recoloring algorithm [XPZ21] based on a modified GrabCut algorithm also demonstrates the ability to improve local recoloring and color leakage. In addition, approaches for constrained editing [MVH*17, NPCB17] support global palette adjustment. Our work extends palette-based recoloring to support direct and localized image-space edits.

3. Workflow

We describe LoCoPalettes’s workflow with the scenario illustrated in Fig. 2. A user begins by loading an image of two women sitting in a forest. The user wishes to edit the ground to appear less red. The user first explores editing global palette colors, but finds that the red in the palette is used for both the ground and skin. The user finds a nice appearance for the ground, but the woman’s skin on the right has changed undesirably (Fig. 2, second image). To fix this, the user places an image-space constraint (Sec. 4.2) on the woman’s forehead to keep her skin colors from changing and another on the ground to directly change its color. LoCoPalettes first tries to find a global palette that satisfies all of the user’s constraints simultaneously. It then determines that this can’t be done, so it splits the image into semantic sub-regions with independent local palettes (Sec. 4.4). Constraints are elastic and order-independent. Once the colors on the skin and ground are satisfactory, the user “bakes” their changes. This updates the rest state of the palettes and removes the current constraints (Sec. 4.2), creating a check-point. The user turns their attention to the woman on the left’s shirt. The user uses a new image-space constraint to change the turquoise color to dark purple. However, the user notices that the forest has also become purple, so they press “undo” to reverse the edit and place another image-space constraint on the forest to keep it from changing. These constraints can only be satisfied by splitting the image again. In the final image, the two women and the ground each have their own local palettes.

4. Method

The geometric palette-based editing formulation computes image colors \( I \) by applying per-pixel mixing weights \( W \) to the palette \( P \), i.e., \( I = W \cdot P \), where \( W \in \mathbb{R}^{(N \times M) \times #P} \subseteq [0,1] \) and \( P \in \mathbb{R}^{#P \times 3} \subseteq gamut \), and \( gamut \) is the unit cube in RGB-space where our algorithm is implemented. Users simply change the colors in \( P \) to globally recolor the image; the mixing weights remain fixed [CFL*15, TLG16, TEG18a, WLX19]. While this formulation provides fast and simple global recoloring, users may struggle to achieve a desired color change in the resulting image. For example, given a color \( c \in \mathbb{R}^3 \) with its corresponding mixing weights \( w \in \mathbb{R}^{#P} \), we can express \( c = w \cdot P \). If users wish to change the original color \( c \) into a different color \( c’ \), users need to anticipate the effects of multiple changes to the palette \( P \) on changes in \( c’ \). This is a difficult task even if users are given the exact weight values corresponding to the palette colors. In addition, if multiple image-space constraints are placed on an image, a single palette may not be able to satisfy all of them (Fig. 3). We tackle these usability problems and achieve users’ constraints with constrained optimization, automatic hierarchical semantic segmentation, and local palettes.

4.1. Sparser Weights

Our goal is to support real-time optimization on color constraints. Therefore, we adopt [TEG18a]’s linear palette-based formulation for efficiency. The linear formulation allows for efficient optimization when satisfying user constraints (Eq. 1) and propagating palettes throughout the hierarchy (Eq. 4). Previous approaches such as [CFL*15] or [AASP17] do not satisfy the linearity requirement. We follow [TEG18a]’s RGBXY approach for palette extraction and weight computation. However, we propose a modification to the weight computation to achieve sparser results. Given an arbitrary image \( I \) with size \( N \times M \), we follow [TLG16] and decompose \( I \) into a palette \( P \) with \(#P\) colors by simplifying the convex hull of the image’s colors in RGB-space. For mixing weights, to ensure spatial coherence, we follow [TEG18a]’s RGBXY approach and compute the convex hull vertices in RGBXY-space for colors and spatial locations in \( I \):

\[
V_{\text{RGBXY}} = \text{ConvexHull}\{([R_i, G_i, B_i, X_i, Y_i]) | i = 1, 2, \ldots, N \times M\}
\]

where \( V_{\text{RGBXY}} \) are convex hull vertices in RGBXY-space and \( i \) enumerates each pixel in \( I \). Note that in [TEG18a], they com-
Figure 4: Using K-means to find interior sparsifying points improves sparsity versus [TEG18a]. This example took 160 seconds to compute with K-means, 14 seconds with our proposed PCA projection, and 7 seconds for the unmodified algorithm [TEG18a]. Based on [AASP17]’s metric, our weights perform the best, with sparsity cost 1.224, versus K-means sparsity 1.28, and [TEG18a] sparsity 1.574. Weights computed in pure RGB-space (bottom row) have maximum sparsity (1.117 in this example), but lack spatial coherence, which manifests as speckles. Our approach exhibits a balanced outcome, delivering both efficiency and a satisfactory level of spatial coherence. Edited results under a single global palette are shown in the rightmost column.

compute spatial weights $W_{\text{RGBXY}}$ with respect to $V_{\text{RGBXY}}$ for each RGBXY point and project $V_{\text{RGBXY}}$ to RGB-space for computing color mixing weights $W_{\text{RGB}}$ with respect to the RGB-space geometric palette. Here, we introduce a simple and fast modification to achieve sparser weights. Instead of computing spatial weights by using $V_{\text{RGBXY}}$, we compute our spatial weights $W_{\text{F RGBXY}}$ by using augmented internal vertices $V_{\text{A}}$ along with $V_{\text{RGBXY}}$. The motivation is that internal vertices $V_{\text{A}}$ can be added properly such that RGBXY points are closer to $V_{\text{A}}$ than to $V_{\text{RGBXY}}$, and therefore, $W_{\text{F RGBXY}}$ can be sparser. Naively, one can compute spatial weights with respect to all RGBXY points. This makes $W_{\text{RGBXY}}$ an identity matrix, which is the sparsest set of weights. However, this naive approach is equivalent to computing weights in only RGB-space, which lacks spatial coherence.

Given a set of RGBXY data $I_{\text{RGBXY}}$, to compute $V_{\text{A}} \subset I_{\text{RGBXY}}$, our goal is to find some-but-not-too-many internal vertices that can reasonably separate the given RGBXY data distribution into pieces when tessellating $V_{\text{A}} \cup V_{\text{RGBXY}}$. In other words, $V_{\text{A}}$ are reasonably distant from each other under $I_{\text{RGBXY}}$. Note that we do not want to find as-many-as-possible internal vertices since they might penalize spatial coherence. We considered K-means clustering. Although we could obtain slightly improved sparsity with small $K$ (Fig. 4), the time complexity of K-means is dependent on the cluster count $K$ and is too slow for useful values of $K$.

Therefore, we introduce an approach to find $V_{\text{A}} \subset I_{\text{RGBXY}}$ using semantic feature vectors from [AOP*18]’s feature extractor. We denote a feature vector at pixel location $i$ as $F^i \in \mathbb{R}^{128}$. Then, we concatenate $F^i$ with RGBXY data $I_{\text{RGBXY}}$ at each pixel $i$ and we denote the concatenated per-pixel vector as $I_{\text{RGBFEAXY}}^i \in \mathbb{R}^{133}$. Since convex hull vertices capture the geometric structure of data, we wish to compute convex hull vertices:

$$V_{\text{RGBFEAXY}} = \text{ConvexHull}(\{I_{\text{RGBFEAXY}}^i | i = 1, 2, \ldots, NxM \})$$
the resulting palette to be the baseline used in future optimizations and releasing all active constraints. The user can then continue editing the new baseline palette.

Consider an image decomposed into a pre-computed global palette $P \in \mathbb{R}^{n \times 3}$ with global mixing weights $W \in \mathbb{R}^{N \times \#p}$ where $N$ is the number of pixels in the image and $\#p$ is the number of palette colors. Given a constraint $c_i$ placed at image location $x$ with desired color $c \in \mathbb{R}^3$, we define $w_k$ to be the pixel weights obtained from $W$ at $x$. In practice, because users can’t click at the precision of an individual pixel and to avoid pixel-level noise or outliers, we use the average weights in a small $3 \times 3$ window around $x$ for $w_k$. Our goal is to optimize for the minimum palette change $\Delta P$ such that the new palette $(P + \Delta P)$ satisfies the desired color constraint. To avoid modifying the user’s direct palette edits, we define a palette constraint as $(P[j], c_P)$, meaning that user wants to change the $j$th palette color of palette $P$ to $c_P$, i.e. $P[j] = c_P$. Considering both image-space constraints and palette constraints together, we solve for the new palette as

$$
\min_{\Delta P} \|\Delta P\|_{2,1}
$$

subject to

$$
\begin{align*}
&\|\text{LAB}(w_k, (P + \Delta P)) - \text{LAB}(c_i)\|_2 \leq \text{JND} \\
&0 \leq P + \Delta P \leq 1 \\
&\text{(P + \Delta P)[j]} = c_P
\end{align*}
$$

where LAB is the operator that converts any color in RGB-space to LAB-space and JND is the Just Noticeable Difference threshold in LAB-space (2.3). The $L_{2,1}$-norm we use has desirable properties. Note that the $L_{2,1}$-norm for a matrix $X \in \mathbb{R}^{m \times n}$ can be written as:

$$
X = \sum_{j=1}^{n} \sum_{i=1}^{m} X_{ij}^2
$$

We want to change as few palette colors as possible, so the $L_{2,0}$ norm comes to mind. However, for any edit, it may be possible to achieve it by moving one vertex of the palette extremely far away, which would produce an $L_{2,0}$ norm of 1. This is not desirable, even though the gamut constraint will in general prevent that solution. There are also infinitely many solutions involving two vertices since the $L_{2,0}$ norm will consider all such solutions equal with value 2 without any way to distinguish them. Therefore, rather than being computationally intractable, the $L_{2,1}$ relaxation allows us to change as few palette colors as possible while also considering the total distance travelled by the palette colors. Since there are only $(3 \cdot \#p)$ degrees of freedom regardless of image size, Eq. 1 can be solved in real-time. We use SciPy’s Sequential Least Squares Programming (SLSQP) solver. Our formulation extends to constraining any number of palette colors and any $k \in \mathbb{Z}^+ \times 3$ pixel constraints by using $w_i \in \mathbb{R}^{k \times \#p}$ and $c_i \in \mathbb{R}^{k \times 3}$. Our optimization is elastic, meaning that adding or removing any constraint will trigger our optimizer to find the best palette to satisfy the current constraints.

### 4.3. Palette and Weight Hierarchy

The limitation of just using a global palette in palette-based editing is that changes to the global palette affect all colors of the image. Moreover, a palette for a subset of the image will necessarily be more representative than the global palette. This motivates us to...
propose an efficient data structure for hierarchical palettes that supports local edits when necessary or desired. Our hierarchy has the desirable property that pixels belonging to a node are reconstructed virtually and identically via that node’s palette or as the union of its children’s reconstructions, provided that the children’s palettes haven’t been independently edited. This guarantees that continuous changes to local palettes produce continuous changes to the reconstruction. In other words, no jarring or discontinuous change occurs to the image when an infinitesimally small change is initially made to a local palette.

4.3.1. Hierarchy Definition

We define a hierarchical segmentation tree \( H = (S, E) \), where \( S \) is a set of nodes and \( E \) is a set of edges describing connectivity between nodes in the tree. We denote \( s_j \in S \) as the \( j \)th node, with \( s_0 \) as the root node. Each node \( s_i \) in a given \( N \)-by-\( M \) image \( I \) has a corresponding sub-region mask \( r_i \), which is an \( N \times M \) matrix whose elements lie within \([0,1]\). (In practice, the \( r_i \) can be stored more efficiently, since they are 0 outside of the region’s bounding box.) We use real-valued masks instead of binary masks to maintain soft boundaries when compositing locally-recolored sub-regions. We then define the compositing operator \( \odot \) that performs element-wise multiplication using \( r_i \) over all three channels of the image \( I \). We set \( r_0 \) at the root node \( s_0 \) to be an all-ones matrix, which means that it covers the entire image \( I \) with full pixel weight, i.e. \( I = r_0 \odot I \).

4.3.2. Hierarchical Semantic Soft Segmentation

We build our hierarchy automatically using DETR [CMS20] for panoptic semantic image segmentation. DETR outputs labels for a hierarchical segmentation with 3 levels: root \( \rightarrow \) classes \( \rightarrow \) instances. However, DETR’s output has hard edges not always aligned well with the image contents. To create soft edges guided by the image contents, we first dilate each class-instance segment (with a 5 \( \times \) 5 kernel)—to guarantee that the entire image is covered by the union of segments—and then perform a guided filter [HS15].

Figure 6: An example palette hierarchy \( \mathcal{H} \) with per-node activations \( a_i \in \mathbb{R} \) and palettes \( P_i \). LoCoPalettes uses DETR [CMS20] to automatically create a semantic hierarchy organized as root \( \rightarrow \) classes \( \rightarrow \) instances. We create soft region masks \( r_i \) with image-guided feathering via guided filtering [HS15].

using the original image data with a radius of 5 pixels. See Fig. 6 for a diagram illustrating our hierarchy data structure.

We also experimented with creating a trimap by dilating and eroding the output of DETR and inputting that to the KNN matting algorithm [CLT13] (Fig. 7). The resulting mattes have slightly better boundaries than our guided-filter approach, but it is very expensive to compute (~30 seconds per segment). Therefore, we trade off the higher-quality but expensive feathering of KNN matting for real-time, reasonable-quality soft boundaries using a guided filter. We evaluate the effectiveness of both methods, as well as a comparison to a superpixel-based approach (Fig. 9).

4.3.3. Palettes and Weights

For each node \( s_j \) in a given hierarchy \( \mathcal{H} \), we compute its geometric palette \( P_j \) using [TEG18a]’s convex hull simplification with a fixed number of palette colors \#\( p \) using the pixel colors in the node’s sub-region \( r_j \). We also compute corresponding weights \( W_i \) using our modified approach described in Sec. 4.1 with respect to \( P_j \).

4.3.4. Reconstruction

We reconstruct the edited image from the leaf nodes of \( \mathcal{H} \). Stated formally, the reconstruction process is as follows. Given a set of \( n \) leaf nodes \( x = \{s_1, s_2, \ldots, s_n\} \) with corresponding weights \( W_i \) and palettes \( P_i \) for \( i = 1, \ldots, n \), we compute:

\[
I \leftarrow (1 - r_i) \odot I + r_i \odot f(W_i \cdot P_i)
\]

where \( f \) is the reshaping operator. Since our reconstruction occurs via leaf nodes, any child palette \( P_i \) needs to reflect changes to its parent palette \( P_j \). For example, if a given \( \mathcal{H} \) only has two levels, any leaf node \( s_j \) needs to reconstruct the same colors as the root node does in sub-region \( r_j \), even when the root node’s palette \( P_0 \) has been edited. To do this, we perform palette propagation. We propagate a modified parent palette \( P_i \) to a child palette \( P_j \) by minimizing the color differences restricted to the child’s sub-region \( r_j \):

\[
\min_{P_i'} \| W_i \cdot P_i' - W_p[l_j \cdot P_p]\|_2^2
\qquad \text{subject to} \quad 0 \leq P_i' \leq 1
\]

where \( P_i', P_p \in \mathbb{R}^{p \times 3} \) and \( W_i, W_p[l_j \cdot P_p] \in \mathbb{R}^{K \times \#p} \) with \( K \) the number of pixels in sub-region \( r_j \). This can be expressed as a small, \( \#p \times \#p \) problem.

© 2023 The Authors. Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.
4.4. Sparse Editing with Hierarchy

Since a single palette may fail to satisfy all constraints, we use palette splitting rules with optimization under hierarchy to address this issue. Given a hierarchy $\mathcal{H}$ with $n$ nodes and an activation tree $\mathcal{A}$. Users add constraints by clicking on image points or palettes colors (Figs. 1–3). Every time a constraint is added, modified, or removed, LoCoPalettes re-runs the optimizer from scratch to satisfy all existing constraints. Consider $k$ image-space constraints $\mathcal{K} = \{c_{1}, c_{2}, \ldots, c_{k}\}$ and $l$ palette constraints $\mathcal{L} = \{(P_{1}[j], c_{P_{1}}), (P_{2}[j], c_{P_{2}}), \ldots, (P_{l}[j], c_{P_{l}})\}$, where $(P_{i}[j], c_{P_{i}})$ constrains the $j$-th color of node $s_{i}$’s palette $P_{i}$ to the color $c_{P_{i}}$.

4.4.1. Palette Splitting Rules

For each constraint $c_{j}$ in $\mathcal{K}$ for $j = 1, \ldots, k$, we push it to the deepest activated node $s_{3}$ that contains $c_{j}$, where containment is based on the sub-region mask $s_{3}$. We optimize the corresponding palette $P_{s_{3}}$ at $s_{3}$ to satisfy both $c_{j}$ and any palette constraints $(P_{s_{3}}[j], c_{P_{s_{3}}})$ at node $s_{3}$. We don’t push it deeper (i.e., to a descendant of $s_{3}$) unless optimization fails. We don’t push it shallower since $s_{3}$ is already activated and would occlude the effect of the image-space constraint.

When optimization fails, we push the most recently edited image-space constraint—the one that triggered the failure—to the shallowest inactive node that contains it (equivalently, to the child of the deepest activated node that contains it). This creates a *palette split*. Note that the *split* only occurs when Eq. 1 is over-constrained (Alg. 1, line 9). Users are allowed to *bake* in changes, i.e., the optimized palettes replace the unoptimized ones in $\mathcal{H}$ and the activation status is updated in $\mathcal{A}$.

4.4.2. Optimization under Hierarchy

LoCoPalettes optimizes the entire $\mathcal{H}$ by optimizing palettes to satisfy constraints using Eq. 1 along with *palette propagation* (Eq. 4) every time the constraints change. We also allow users to *localize* an image-space constraint, which directly associates the constraint with the deepest node that contains it, active or inactive. That is, users are able to directly edit colors in the level of object instances in our semantic hierarchy. To keep track of which local palettes should be used to optimize which constraints, each node in $\mathcal{H}$ stores the list of constraints to be used for optimizing its palette. Pseudocode can be seen in Algorithm 1.

```plaintext
Algorithm 1: Optimization under Hierarchy

Input: $\mathcal{H}$ and $\mathcal{A}$ with $k$ pixel constraints $\mathcal{K}$ and $l$ palette constraints $\mathcal{L}$ ($K_{i}$ implies $i^{th}$ constraint in $\mathcal{K}$)
Output: Edited image using new $\mathcal{H}$ and $\mathcal{A}$
1 for $i \leftarrow 1$ to $k$ do
    2 $s \leftarrow \text{findMatchedNode}(K_{i})$; /* Splitting rule in Section 4.4 */
    3 Add $K_{i}$ to node $s$ in $\mathcal{H}$;
    4 while True do
        5 OptimizeHierarchy($\mathcal{H}, \mathcal{A}$); /* Solve Eq. 1 */
        6 if each node in $\mathcal{H}$ has no errors then
            7 PaletteProp($\mathcal{H}, \mathcal{A}, s$); /* Solve Eq. 4 */
            8 break;
        else
            9 ModifyHierarchy($\mathcal{H}, \mathcal{A}$); /* Change node location for $K_{i}$ */
        end
    end
11 end
12 return ReconstructImage($\mathcal{H}$); /* Compute Eq. 3 */
```

5. Results and Evaluation

We show a variety of edited images made using LoCoPalettes in Figures 1–3 and a gallery (Fig. 11). These examples show how LoCoPalettes overcomes the limits of global palette-based editing. We show comparisons to [TEG18a]’s global, purely palette-based approach in Fig. 3 and 11. Fig. 11 also compares to a version of our approach with the hierarchy disabled. Without a hierarchy, it is impossible to avoid undesirable changes to local regions. Without image-space constraints, [TEG18a] is only able to approximate the editing intent with a large number of palette manipulations. This is because the mixture of palette colors on objects is not always obvious. Users must, in effect, manually iterate the steps of LoCoPalettes’ optimization.

The supplemental video shows a complete editing session. Hierarchical segmentations for our examples can be found in the supplemental materials.

5.1. Region Boundaries

Fig. 8 shows the benefit of applying a guided filter [HS15] to feather region boundaries. Without the guided filter, colors leak across the boundary when edited. KNN Matting (Fig. 7) further improves the quality of our region boundaries, but at great computational cost.
Input
Detectron + Guided Filter
Detectron output
Edits using only Detectron output
Edits using Detectron + Guided Filter

Figure 8: Directly using outputs from DETR [CMS∗20] causes color leakage between segment boundaries when reconstructing the edited image (red arrows). LoCoPalettes applies a guided filter [HS15] to feather abrupt color changes across boundaries. Photo courtesy of Ferdinand Studio.

Table 1: A numerical comparison of our weights’ sparsity versus [TEG18a]. We measure sparsity using two different metrics: Eq. 8 in [TLG16] (offset by 1 to a positive range) and the second term of Eq. 4 in [AASP17]. Images are from Figure 12. Our modified weights perform better (smaller cost) under both metrics.

<table>
<thead>
<tr>
<th>Sparsity Estimate:</th>
<th>Tan et al. [2016]</th>
<th>Aksoy et al. [2017]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weights:</td>
<td>Tan et al. [2018]</td>
<td>Ours</td>
</tr>
<tr>
<td>Mountain</td>
<td>0.2630</td>
<td>0.2586</td>
</tr>
<tr>
<td>Birds</td>
<td>0.2670</td>
<td>0.2614</td>
</tr>
<tr>
<td>Colorful</td>
<td>0.2549</td>
<td>0.2511</td>
</tr>
<tr>
<td>Boy</td>
<td>0.2676</td>
<td>0.2638</td>
</tr>
</tbody>
</table>

5.2. Sparser Weights

Examples comparing our sparser weights (Sec. 4.1) to the unmodified algorithm from [TEG18a] can be seen in Fig. 4 and Fig. 5, as well as Fig. 12. A numerical comparison for these examples can also be seen in Table 1. Our weights are sparser under two metrics from the literature [TLG16, AASP17]. We also experimented with K-means as an alternative to our PCA-based sparsity approach (Fig. 4). The K-means approach was less sparse and took an order of magnitude longer to compute.

5.3. Segmentation

Apart from using DETR, we also experimented with superpixel (SLIC [ASS∗12]) merging to create segments on-the-fly from the user’s image-space constraints (Fig. 9). We create a matrix storing pairwise superpixel distances using a combination of color information and [AOP∗18]’s per-pixel feature vectors. We interpret this matrix as a graph and use a binary search to find the largest threshold value which still separates the first two constraints after merging superpixels within each component of the cut graph. After merging, we assign all unassigned regions to the more general of the two constraints, and then repeat this process for each additional constraint \( c_i \), which is compared against the constraint currently governing the part of the image containing \( c_i \)’s pixel location.

6. Conclusion

LoCoPalettes addresses the primary shortcoming of existing palette-based editing frameworks. It allows for local, semantic changes when user edits cannot be achieved with a single global palette. We do this by integrating recent work [CKT∗23] into a palette hierarchy to provide a low-overhead, real-time optimization that achieves user constraints with sparse changes to the palette and hierarchy. We also proposed a new method for computing spatially smooth weights that improves sparsity over the state of the art.
6.1. Limitations and Future Work

Although we have proposed a fast, automatic algorithm for creating a soft semantic hierarchical segmentation, we are limited by the quality of the underlying hierarchical segmentation mode (Fig. 10). The limitations of the model we use, DETR [CMS∗20], are not always predictable, although we have found it to be more robust than [AOP∗18], which used an older network. We would also like to explore creating dynamic editing-aware segmentations on-the-fly based on user’s image-space constraints, as in our superpixel segmentation experiment.

Inspired by [KOWD21], we would like to extend LoCoPalettes to video editing. Instead of storing soft masks at each node, we plan to explore storing spatial-temporal segmentation along with geometric palettes computed from [DLX∗21]. In addition, we would also like to explore speeding up the computation of local palettes by using [WLX19] or variational approaches [LLTY21].

References

Figure 12: More examples of sparser weights (Sec. 4.1) versus [TEG18a]. Photos courtesy Trace Hudson, Jess Bailey Designs, Nibret Sanga.
LoCoPalettes: Local Control for Palette-based Image Editing

Chao, Klein, Tan, Echevarria and Gingold / LoCoPalettes: Local Control for Palette-based Image Editing


