Interactive Region Matching for 2D Animation Coloring Based on Feature’s Variation

Pablo GARCIA TRIGO†, Henry JOHAN††, Takashi IMAGIRE†††, Nonmembers, and Tomoyuki NISHITA†††, Member

SUMMARY We propose an interactive method for assisting the coloring process of 2D hand-drawn animated cartoons. It segments input frames (each hand-drawn drawing of the cartoon) into regions (areas surrounded by closed lines. E.g. the head, the hands) extracts their features, and then matches the regions between frames, allowing the user to fix coloring mistakes interactively. Its main contribution consists in storing matched regions in lists called “chains” for tracking how the region features vary along the animation. Consequently, the matching rate is improved and the matching mistakes are reduced, thus reducing the total effort needed until having a correctly colored cartoon.

key words: 2D animation, coloring, matching

1. Introduction

Traditional 2D animated cartoons are created by drawing and coloring each frame manually as follows [1]:

1. The main artists draw the concept storyboard.
2. The senior artists draw the key frames (main hand-drawn drawings of the cartoon, all made of lines only).
3. Secondary artists draw the inbetweenings (the frames between the key frames, lines only).
4. Other artists color all the drawn frames.
5. Background, music and dialogs are added.

 Usually steps 3 and 4 consume 60% of the whole process [2], even when using commercial software like Retas or Animo, and previous research has concentrated on automating them using a matching process:

1. Segment each input frame into its regions (areas surrounded by closed lines. E.g. the head, the hands).
2. Analyze each region in each frame and extract its features for using them in the next step.
3. Matching: For all frames, identify what regions in corresponding frames with lines only. There is no grey information. Thus, this solution is not applicable.

4. Correct manually wrongly colored regions.

This paper introduces a classification-based interactive method for assisting the coloring process. Each time a region is matched to another, they are classified as belonging to the same group of regions. We call these groups “chains” of regions. During the matching, we use not only the features of the region currently being matched but also the features of the previous regions in the chain. Having the chains, we know how regions’ features vary during the animation, and that helps us reduce the number of wrongly matched regions in the matching step. Consequently, more regions are automatically correctly colored, reducing the workload.

2. Related Work

Previous methods have concentrated in automating completely the coloring process with no user intervention at all. Methods like [3] specialize in the colorization of old grayscale cartoons. The levels of gray serve as a hint for the colorization, but nowadays animation pipeline produces frames with lines only. There is no grey information. Thus, this solution is not applicable.

Having only lines and the possible lack of coherence between frames can make the matching very difficult. For addressing that, methods like [2], [4], [5] increase the matching accuracy by using master frames as a reference, building a hierarchy of regions or inserting skeletons.

While the above approaches can color successfully certain kinds of animations, they may achieve bad results in others. That is especially true in [2]–[6] if the topology of the regions suffers many changes. This leaves the artist with the task of correcting many wrongly colored regions manually in the end. This is due to the fact that coloring mistakes in one frame propagate to the rest of the frames. Furthermore, those mistakes prevent other regions from being correctly colored, creating more mistakes.

This paper proposes a different approach. Instead of aiming for the complete automation of the coloring process, it recognizes that matching mistakes are very difficult to avoid in all cases and introduces user interactivity for fixing matching mistakes as soon as they appear. On top of this, it automatically builds chains of regions along the animation. This helps to know how regions change and increases the accuracy of the matching. Also, the fact that it does not need any additional step like manually registering a skeleton as in [4], [6] compensates for the work required in the interactive part. We work directly with raster images (bitmaps) as in [7] and [8]. Taking the bitmaps and vectorizing them is done in
3. Proposed Method

3.1 General Overview

Our algorithm consists of the following steps:

1. Scan the hand-drawn frames.
2. Apply filters to reduce noise and holes (Sect. 3.2).
3. Segment each input frame into regions (Sect. 3.2).
4. For each region, extract its shape and topological features (Sect. 3.3).
5. Interactive matching (Sect. 3.6).
6. During the interactive matching, group the regions into chains to improve the matching accuracy (Sects. 3.4 and 3.5).

3.2 Segmentation

The frames are preprocessed by a filter for increasing the contrast and hue of the drawn lines. The lines can be of up to three colors: black, red and blue (See Sect. 3.3). Due to the scan process there can be artifacts: noise and holes. These artifacts can influence negatively the segmentation, making false small regions (made up of very few pixels). For fixing them, we remove regions whose size is less than a certain threshold (we use a size of 5 pixels).

3.3 Region Features

For each region we extract the following features:

1. Shape features:
   a) Area. b) Frontier. c) Dominant Points.
2. Topological features:
   a) Position. b) Neighbor regions.
   1.a) The area is the number of pixels that make up that region. The background is usually the region with the biggest area, although not necessarily.
   1.b) Regions are enclosed by the strokes that the artist drew. Those strokes are usually black, but can be also blue or red, indicating the presence of a highlight or a shadow. The frontier of a region is the boundary of the region. They are contiguous to the strokes and have their color: black, blue or red.
   1.c) Dominant Points are points in a boundary that have a high curvature. For finding them, we use the k-cosine as in [11]. Once we have found the angle at each pixel of the frontier of a region, we determine that those pixels with an angle less than 90 degrees or more than 270 degrees are Dominant Points (see Fig. 1).
   2.a) The position of a given region \( r \) is reported as its centroid, where the centroid is the arithmetic mean of all pixels of \( r \) (see Fig. 2).
   2.b) For each region we register its neighbor regions.

They can be found expanding the borders of the regions outwards up to a certain distance. This can be tuned according to the thickness of the lines.

3.4 The Matching Algorithm

We can think of the matching process as a process of making chains of regions. It starts with a given region \( r_1 \), let us call it source region, and tries to find its corresponding region \( r_2 \) in the next frame (see Fig. 3). If found, \( r_1 \) and \( r_2 \) are linked, \( r_2 \) becomes the new source region and the process is repeated. When it cannot find its corresponding region in the next frame the chain ends. We call the above process "to match a given region \( r \)", which is the process of finding all the regions that correspond to \( r \) in the following frames. The result is a chain of regions started by \( r \) (Fig. 4). We have implemented a forward matching algorithm. We start in the first frame and we try to match all its regions. When done, we look at the next frame, and do the same with all the regions that are still unmatched. We continue until we reach the last frame.

3.5 The Comparison Function

For finding the correspondence of one region in the following frame we use a comparison function \( \text{comp} \). It accepts two regions as input and returns a score indicating how probable is that those two regions correspond to each other. A score is always a number between 0 and 1, 0 in the case of total dissimilarity and 1 when the features are identical.

For a given region \( r_1 \) in a frame (see Fig. 3), we run \( \text{comp} \) against \( r_1 \) and all the regions in the next frame that have not been matched yet. Then, we pick up the region with the highest score as \( r_1 \)'s correspondence. Sometimes
we cannot find a correspondence because all the regions in the next frame have already been matched.

comp is made of five sub functions that compare each of the features in Sect. 3.3 and return a score (again between zero and one). The score of each sub function is multiplied by a weight \( w \) that increases or reduces the importance of that feature. Finally, all the weighted scores are summed to make up the final score that comp will return.

\[
comp(r_1, r_2) = \left( \frac{1}{5} \right) \left( \begin{array}{l}
comp_{\text{Area}}(r_1, r_2) \cdot w_{\text{Area}} + \\
comp_{\text{Frontiers}}(r_1, r_2) \cdot w_{\text{Frontiers}} + \\
comp_{\text{DomP}}(r_1, r_2) \cdot w_{\text{DomP}} + \\
comp_{\text{Pos}}(r_1, r_2) \cdot w_{\text{Pos}} + \\
comp_{\text{Neighs}}(r_1, r_2) \cdot w_{\text{Neighs}}
\end{array} \right)
\]

(1)

Concretely, the functions that compare each feature are as follows:

- \( \text{comp}_{\text{Area}} \) compares the area of two regions calculating their ratio. The areas are the amount of pixels in the region. Dividing the smaller area by the bigger we obtain the score between zero and one.

\[
\text{comp}_{\text{Area}}(r_1, r_2) = \frac{\text{Min}(\text{Area}(r_1), \text{Area}(r_2))}{\text{Max}(\text{Area}(r_1), \text{Area}(r_2))}.
\]

(2)

In the above formula, \( r_1 \) and \( r_2 \) are the input regions. Functions Min() and Max() return the region with the smallest and biggest area among \( r_1 \) and \( r_2 \) respectively.

- \( \text{comp}_{\text{Frontiers}} \) compares the length of the frontiers distinguishing by color (\( \text{col} \) in Eqs. (3) and (4)). Colors that do not exist in both regions are ignored.

\[
\text{comp}_{\text{Frontiers}}(r_1, r_2) = \sum_{\text{color} = \{\text{black}, \text{blue}, \text{red}\}} \frac{\text{comp}_{\text{FrontierLength}}(r_1, r_2, \text{col})}{N_{\text{colors}}},
\]

(3)

\[
\text{comp}_{\text{FrontierLength}}(r_1, r_2, \text{col}) = \frac{\text{Min}(F_1(\text{col}, r_1), F_2(\text{col}, r_2))}{\text{Max}(F_1(\text{col}, r_1), F_2(\text{col}, r_2))}
\]

(4)

- \( \text{comp}_{\text{DomP}} \) compares the dominant points of two regions \( r_1 \) and \( r_2 \) taking the angles of the dominant points, subtracting them, and calculating the mean of the differences. There is a problem: since the frontiers are circular, there is no way to determine a robust starting point that will work for all the regions. Thus, we decided to consider all the possibilities: Given two lists of angles, we compare them, store their score and then shift one position one of the lists, and perform the comparison again. We repeat shifting and comparing until we go back to the original position and keep the highest score. This way, we solve the problem of the starting point and make this function compare well even when the regions are rotated. While it is a brute force approach, the average number of dominant points in a region, in all our sample animations, was around ten points at most, making \( \text{comp}_{\text{DomP}} \) fast enough.

\[
\text{comp}_{\text{DomP}}(r_1, r_2) = \text{Max}(\text{AllShifts}(\text{ALComp}(r_1, r_2))).
\]

(5)

\[
\text{ALComp}(r_1, r_2) = 1 - \text{Mean}([\text{dist}_{\text{DomP}}(r_1, r_2)]) \left( \frac{1}{360} \right).
\]

(6)

Equation (6), \( \text{ALComp} \), takes the angle lists and compares them. \( \text{dist}_{\text{DomP}} \) and \( \text{dist}_{\text{DomP}} \) are the lists of angles of the dominant points. They are made the same length by down-sampling the longest one, then the angles are all subtracted and the mean is calculated. The angles are in degrees. For normalizing to [0, 1] we multiply by (1/360) and subtract to 1. In Eq. (5) \( \text{AllShifts} \) is the function that calculates \( \text{ALComp} \) shifting \( r_2 \)'s list of angles until a whole rotation is done. \( \text{AllShifts} \) returns all the scores and \( \text{Max} \) returns up the highest score.

- \( \text{comp}_{\text{Pos}} \) compares the position calculating the distance between their centroids. The result is normalized between zero and one dividing by the distance to the furthest region in the frame.

\[
\text{comp}_{\text{Pos}}(r_1, r_2) = 1 - \frac{\|\text{centroid}(r_1) - \text{centroid}(r_2)\|}{\text{Max}_{i}(\|\text{centroid}(r_1) - \text{centroid}(r_i)\|)}.
\]

(7)

In Eq. (7), \( \text{centroid}(r) \) returns the position vector of region \( r \). In the denominator we calculate the distance from \( r_1 \)
to the furthest region in the frame of \( r_2 \). We do it calculating the distance from \( r_1 \) to all the regions \( r_i \) in the frame of \( r_2 \) and choosing the maximum.

\( \text{comp}_{\text{neigh}} \) Given two regions \( r_1 \) and \( r_2 \) for comparing their neighbors we can use the above functions: We compare the neighbors’ features and return a score that indicates their similarity. The only feature that we do not compare is, precisely, the neighbor’s neighbors to avoid entering an infinite loop. Also, given the set of neighbors of \( r_1 \) and the set of neighbors of \( r_2 \), there are many possibilities for matching neighbors among them. Again, we calculate all the possibilities and return the best score. It is important to note that many times it is not necessary to actually perform the comparison: if two neighbors have been matched previously, we already know that they correspond. We can know it by looking at the chains. Similarly, we can know that two neighbors do not correspond when one of the neighbors (or both) have been matched to other regions.

\[
\text{comp}_{\text{neigh}}(r_1, r_2) = \max\left(\text{AllCompN}(\text{CompN}(r_1, r_j)), \forall r_j \in \text{Neighborhood}(r_1)\right), \quad (8)
\]

\[
\text{compN}(n_1, n_2) = \begin{cases} 
1.0 & \text{correspond}(n_1, n_2) \\
0 & \text{notCorrespond}(n_1, n_2) \\
\text{comp}'(n_1, n_2) & \text{notMatchedYet}(n_1, n_2)
\end{cases}, \quad (9)
\]

\( \text{compN} \) (Eq. (9)) checks how similar are a given neighbor \( n_1 \) of the region \( r_1 \) and a given neighbor \( n_2 \) of the region \( r_2 \). Looking at the chains, \( \text{correspond}(n_1, n_2) \) returns true if and only if both regions \( n_1 \) and \( n_2 \) have been matched to each other. In this case we return directly the highest score. \( \text{notCorrespond}(n_1, n_2) \) returns true when regions \( n_1 \) and \( n_2 \) have been matched not to each other, but to other regions (\( n_1 \) and \( n_2 \) are not in the same chain). In this case we return directly the lowest score. Finally, \( \text{notMatchedYet}(n_1, n_2) \) returns true if we have not matched yet neither \( n_1 \) nor \( n_2 \). In this case, we do not have any information about any of them and need to use \( \text{comp}' \) to find a score. \( \text{comp}'(n_1, n_2) \) is like \( \text{comp} \) (Eq. (1)) but without the neighbors feature (the last addend). In Eq. (8), \( \text{AllCompN} \) uses \( \text{compN} \) for doing all combinations of comparisons of the neighbors of \( r_1 \) against all the neighbors of \( r_2 \) and returns the mean of the scores in each combination. Max returns the highest score. This makes this function robust even in the case where neighbors change positions. In the case that the number of neighbors of \( r_1 \) and \( r_2 \) are different, the extra regions receive a score of zero. This comparison function, as the rest, is symmetric, returns the same value for inputs \((r_1, r_2)\) and \((r_2, r_1)\).

Regarding the weights used in Eq. (1), each chain has its own set of weights and they are recalculated as the chain grows. Each weight \( w_i \) is calculated according to the regions in the chain to reflect the variance of the features along it. Specifically, the variance of a feature \( \text{feat} \) in a chain \( ch \) is:

\[
\text{Variance}(ch, \text{feat}) = \text{Score}_{\text{Max}}(ch, \text{feat}) - \text{Score}_{\text{Min}}(ch, \text{feat}), \quad (10)
\]

where, \( \text{Score}_{\text{Max}}(ch, \text{feat}) \) and \( \text{Score}_{\text{Min}}(ch, \text{feat}) \) return, respectively, the maximum and the minimum score of feature \( \text{feat} \) in chain \( ch \). The variance is a value between 0 and 1, where 0 means no variation and 1 means that varies completely. The weights are calculated as the opposite of the variance:

\[
\text{Weight}(ch, \text{feat}) = 1 - \text{Variance}(ch, \text{feat}), \quad (11)
\]

The weight is also a value between 0 and 1. When the variance is high, it means that that feature changes a lot and is not reliable, thus the weight becomes small. Conversely, if there is almost no variance, the feature is reliable and the weight becomes high. \( \text{Weight}(ch, \text{feat}) \) is used in Eq. (1) to compute each of the weights \( w_{\text{Area}}, \ w_{\text{Frontiers}}, \ w_{\text{DomP}}, \ w_{\text{Pos}} \) and \( w_{\text{Neigh}} \). In each case the feature \( \text{feat} \) is \( \text{Area}, \ \text{Frontiers}, \ \text{DominantPoints}, \ \text{Position} \) and \( \text{Neighbors} \). The chain \( ch \) is the chain of region \( r_1 \).

Taking into account the previous regions in the chain enhances the accuracy, although it may still make wrong matchings. For those cases, the user can correct interactively the matchings while they are being done.

### 3.6 Interactive Matching

In our system, we have opted to make the matching process interactive and ask the user to supervise the matchings as they are done. Although this means more work for the user, it means that matching mistakes are corrected as early as they appear and, in the end, it is less work than fixing all the mistakes of all the frames after an automatic matching with mistakes. Usually, we will open some panels, e.g. showing the first five frames, and do the matching in these five frames only. Once all regions in the working set have been matched, it proceeds to the next five ones, and so on until the last frame. As long as matches are found, chains grow along.

The smallest working set would be two frames: A frame and its following frame. Our example animation has five frames, so we can see them all at once while performing the matching (see Fig. 5).

![Fig 5](image-url) The user interface consists of a set of panels that display the frames to be matched. Here with five panels opened, showing five frames. We can open as many panels as we want.
the same color. A random color is used for distinguishing each chain. Figures 7, 8 and 9 show the first two steps of the matching process.

For fixing matching mistakes the toolbox (the left dialog in the user interface, Fig. 6) determines the action to take when the user clicks on a region in a frame. The actions are: fix a chain, indicate that a chain ends and indicate that a chain starts.

The action “Fix the chain” takes two regions clicked by the user and links them. Note that it is possible to link regions that are in non-contiguous frames. This is especially useful for cases where a region becomes non-visible during some frames but later reappears. When this function is used in regions \( r_1 \) and \( r_2 \), the part of the chain that starts at \( r_1 \) (let us call it “tail”) is deleted (all those regions become non-matched), \( r_1 \) and \( r_2 \) are linked and \( r_2 \) is matched. Thus, the tail starting at \( r_1 \) is effectively recalculated. Because this function is often used, there is a shortcut with only one click: When doing “Right Click” on a region, the system will infer that the region to link is the in the previous frame and will take it from the last built chain. The action “Chain ends” removes the tail from the clicked region. Useful when a region becomes hidden completely in the rest of the animation (only one click). The action “Chain starts” launches the matching process at the clicked region as a new chain. Any tail starting at \( r \) or any link from another chain to \( r \) is deleted first (one click).

4. Results and Discussion

We implemented our system in Java. We ran the tests on an Intel Core2 Quad CPU Q6700 @ 2.66 GHz, with 2 GB of RAM. The current implementation is single-threaded and does all its calculations on CPU. Segmenting and extracting the region features of one frame takes around 300 milliseconds. After the segmentation and the extraction of features, all the interaction is in real-time. We used four animations to test the efficiency of each of the five features separately and compared the results when using all of them together.

In Table 1, we can see how different features perform better or worse depending on the animation. Using all the features makes the algorithm more efficient.

In another experiment (see Table 2), we performed a user test counting the number of mouse clicks and time. Different subjects took a different number of clicks because of using different strategies when coloring or fixing mistakes they made themselves. Underscores show the lowest number of clicks and best times.

Table 3 shows another experiment on the Face animation with user D.

As Table 3 reflects, the user-guided matching method allowed us to do the coloring with less clicks than doing the work manually or with a non-interactive approach. The coloring process of our method is shown in Fig. 10, with the first two steps and the final correct result with correspond-

![Fig. 6](image) Close-up of the user interface.

![Fig. 7](image) The Flag animation: 13 frames of 207 × 163 pixels with 7, 7, 7, 7, 7, 7, 7, 7, 6 and 7 regions respectively.

![Fig. 8](image) The Bear animation: 6 frames of 178 × 309 pixels. Each frame has 20, 20, 16, 20, 20 and 20 regions respectively. (Total: 116 regions)

![Fig. 9](image) The Jump animation: 13 frames of 284 × 532 pixels. Each frame has 18, 16, 17, 16, 17, 18, 20, 20, 21, 21, 21 and 18 regions respectively. (Total: 240 regions)
User test. Our users were amateurs (users A, B, C and D) and after doing some training with the Flag animation (not in the table), they colored the Bear, Jump and Face animation. For each animation, we show the clicks (Clk. Column) and the time (in mm:ss format).

<table>
<thead>
<tr>
<th>User</th>
<th>Bear</th>
<th></th>
<th>Jump</th>
<th></th>
<th>Face</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clk.</td>
<td>Time</td>
<td>Clk.</td>
<td>Time</td>
<td>Clk.</td>
<td>Time</td>
</tr>
<tr>
<td>A</td>
<td>29</td>
<td>2:28</td>
<td>93</td>
<td>9:40</td>
<td>27</td>
<td>2:43</td>
</tr>
<tr>
<td>B</td>
<td>37</td>
<td>3:33</td>
<td>43</td>
<td>7:59</td>
<td>33</td>
<td>4:44</td>
</tr>
<tr>
<td>C</td>
<td>76</td>
<td>5:01</td>
<td>57</td>
<td>8:44</td>
<td>50</td>
<td>4:23</td>
</tr>
<tr>
<td>D</td>
<td>29</td>
<td>2:03</td>
<td>43</td>
<td>4:59</td>
<td>29</td>
<td>3:17</td>
</tr>
</tbody>
</table>

Table 3 Clicks comparison with user D in the Face animation. The manual method had the user specify the matchings between all regions, except the regions in the last frame, which do not have a next frame to be matched to (two clicks for each matching). In Our method the user clicked on the wrong regions for fixing them: 16 regions were fixed by the user and 10 regions were fixed automatically by the algorithm. The Non-interactive method required zero effort from the user initially, but had wrong regions. In order to fix them, 86 clicks were needed with the “fix” action of the Toolbox (two clicks each). The Right-Click option cannot be used in the Manual and Non-Interactive method as it uses the last built chain.

<table>
<thead>
<tr>
<th>Method</th>
<th>Clicks</th>
<th>Wrong regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>Our method</td>
<td>29</td>
<td>0 (16:user, 10: alg)</td>
</tr>
<tr>
<td>Non-interactive</td>
<td>0 (fixes:86)</td>
<td>43</td>
</tr>
</tbody>
</table>

Fig. 10 The Face animation: 5 frames, each of 507 × 446 pixels. Each frame has respectively: 15, 16, 17, 18 and 18 regions, 84 in total. We show the frames before being matched (lines only), the first two steps (hair and face) and the final correct matching.

Fig. 11 The frames after being wrongly matched automatically without user guidance. (Matching mistakes propagated)

In order to set the final colors, we only need to choose one color and apply it to one region. The color will expand automatically to the whole chain. Our method shows how, for difficult cases, an approach that takes into account the pre-
viously matched frames and involves the user can be more efficient that a manual or a non-interactive approach.

5. Conclusions and Future Work

As seen in the results, fixing the coloring mistakes as soon as they appear has allowed us to stop their propagation and the generation of other mistakes, while keeping the user intervention low. Also, classifying the regions into chains and seeing how regions change along them has allowed us to increase the matching accuracy, thus reducing the total number of mistakes and reducing the total effort needed to create a correctly colored cartoon. Our method can be applied to any animated cartoon, since it does not impose any restriction on the kind of character or scene to be colored. As for future work, we want to consider the case where there are regions occluding other regions or when regions break up into several ones or merge into a single one.

References