Vectorizing Cartoon Animations

Songhai Zhang[1] Tao Chen[1] Yi-Fei Zhang[1] Shi-Min Hu[1] Ralph R. Martin[2]
[1]Tsinghua University [2]Cardiff University

***Abstract*—We present a system for vectorizing 2D raster format carton animations. The output animations are visually flicker free, smaller in file size, and easy to edit. We identify decorative lines separately from coloured regions. We use an accurate and semantically meaningful image decomposition algorithm which supports an arbitrary color model for each region. To ensure temporal coherence in the output cartoon, we reconstruct a universal background for all frames, and separately extract foreground regions. Simple user-assistance is required to complete the background. Each region and decorative line is vectorized and stored together with their motions from frame to frame.**

The significant novel contributions of this paper are: (i) the new trapped ball segmentation method, which is fast, supports non-uniformly colored regions, and allows robust region segmentation even in the presence of imperfectly linked region edges, (ii) the separate handling of decorative lines as special objects during image decomposition, which avoids results containing multiple short, thin over-segmented regions, and (iii) extraction of a single patch-based background over all frames, which provides a basis for consistent, flicker-free animations.

***Index Terms*—Cartoon vectorization, Trapped-ball method, Image decomposition, Foreground extraction**

Fig. 1. left column: Original videos; right column: Vectorization results showing region boundaries and decorative lines

I. Introduction

Cartoon animation, i.e. cel animation, has a long history, resulting in a large body of artistic work. Children and adults enjoy the stylized characters, the drawing and animation styles, plots, and soundtracks. Currently, such cultural heritage is preserved and disseminated as digital video after conversion from film using telecine. Such video is in raster format.

However, animated 2D cartoons intrinsically have a vectorized nature. Cartoon making has a relatively fixed composition procedure, including an unchanging but moving background, and an animated foreground. In comparison to real-world videos, 2D cartoon animations have simpler, more artificial contents, composed of regularly (but not necessarily uniformly) colored regions, and wide decorative lines, as shown for example in Fig.2.

This particular nature of 2D cartoons means that there are potential advantages in converting them to vector format:

- a vector format will generally allow higher compression ratios for storage and transmission than raster format, because of the small number of regions which can be described by simple colour models,
- the vector format is resolution independent, readily allowing display on devices with differing capabilities,
- a meaningful vector representation allows the cartoon to be much more easily edited, e.g. to change the location of a character in a scene, or to add new objects in front of some existing objects and behind others,
- a meaningful vector representation is much more useful for multimedia information retrieval, allowing a search to be made for objects of a particular shape,
- undesirable artifacts caused by lossy raster compression will be removed: instead regions will be strongly coloured, with strong edges, which better fits the artistic style of cartoons. This in turn allows better quality of video to be transmitted if limited bandwidth is available, and also has potential applications to cartoon restoration.

In recent years, various approaches have been proposed for vectorizing rasterized images [1], [2], and commercial packages exist, e.g. Stanford VectorMagic [3]. However, these do not take into account the particular features of cartoon

Fig. 2. Cartoon gallery

animations, and in particular tend to suffer from oversegmentation. Furthermore, none of them provides temporal coherence. Sýkora [4]–[6] made significant advances in vectorizing cartoons, but his approach relies on a particular drawing style in which meaningful regions are enclosed by thick bounding lines, and as a result, is inapplicable to many modern cartoon styles.

We note that specific challenges exist when performing 2D cartoon vectorization. The simplicity of the cartoon's contents can be at times a disadvantage as well as an advantage. Small amounts of flickering are more noticeable in simple cartoon scenes than in real-world videos, and strongly decrease the visual quality. Thus, accurate boundaries and locations of segmented regions are necessary to achieve visual temporal coherence. Segmentation should be semantically meaningful in terms of the color regions and decorative lines output. We take into account the particular nature of cartoons to solve these problems.

Edge information provides a strong hint for decomposition of cartoon animations, but extracted edges often contain gaps and can not always be reliably joined. We overcome this problem by using a novel trapped-ball method to stably segment each frame into regions, guided by the edges, but which can cope with gaps. The color in each region need not be uniform: any desired color model may in principle be used (in practice here we use a quadratic model). We separately reconstruct the static background and extract moving foreground objects to provide a layered representation in which the regions and decorative lines are vectorized, and we record their motions. Simple user-assistance is required to complete the background.

The significant novel contributions of this paper are: (i) a new trapped ball segmentation method, which is fast, supports non-uniformly colored regions, and allows robust region segmentation even in the presence of imperfectly linked region edges, (ii) the separate handling of decorative lines as special objects during image decomposition, which avoids results containing multiple short, thin over-segmented regions, and (iii) extraction of a single patch-based background over all frames, which provides a basis for consistent, flicker-free animations.

II. Related work

Sýkora's [4]–[6] work on vectorization of cartoon animations is the most closely related previous work to ours. In his case, cartoons must have a foreground layer consisting of dynamic regions each enclosed by clearly visible outlines. An edge detector similar to a Laplacian of Gaussian filter is used for outline detection, and is extensively discussed in his paper. Foreground and background are extracted by the strong hints of outlines in each frame, and then a graph-based region matching process is used to find the region relations and transformations between frames. His approach relies heavily on correct and *complete* detection of the enclosing outlines. Due to these strong restrictions, his approach fails on many cartoons in practice, such as the one in Fig.2. Our method can handle more complicated cartoons, with non-uniform shading, and weaker edges. Importantly, we are able to compute a high-quality segmentation without perfect edge detection.

High-quality vectorization of cartoons requires accurate segmentation of relatively few meaningful regions. There is a vast literature on image segmentation. Many sophisticated color-based methods, such as mean-shift segmentation [7], typically generate an over-segmented result with too many regions of meaningless shape and flat shading when applied to cartoons. Commercial software, such as Adobe Live Trace, CorelTrace, and AutoTrace, also typically produces regions with flat shading. Ardeco [1] can find regions of quadratically varying color. However, as this method initially labels the pixels randomly and refines the labeling, it too often produces many small regions, and hence is unsuitable for our purpose. We generate larger initial regions based on edge information, and then refine these regions using color compatibility to find precise region boundaries. Because we use edge information to find initial regions, the final regions are larger and more meaningful. However, we only label each pixel once, so our method is much faster than many other segmentation methods, taking just a few seconds per frame.

Sun also presents a semi-automatic *image* vectorization method [2]. Complex images must be manually decomposed into several semantic parts. Each part is then modeled using an optimized gradient mesh Coons patch allowing for smooth color variation. Use of a gradient mesh means that relatively few regions are needed to represent an object. As the author notes, his method has problems with images containing complicated topologies, very thin structures, or many small holes.

Clearly, simply applying segmentation and vectorization on a frame by frame basis will not produce good results, especially in the presence of raster compression artifacts and occlusion. Segmentation which is not coherent between frames will cause flickering in the vectorized output. Previous approaches to temporally coherent video segmentation [8]–[10] have tried to avoid such problems via optimization. Although such methods may be appropriate for real-world video, they do not produce precise enough results for cartoons, as even minute inappropriate regions and outlines are clearly visible, especially when present in the background, due to the smooth shading found in cartoons. We assume that the cartoon has an unchanging background over a sequence of frames, allowing us to achieve temporal coherence by extracting a unified background before detection of foreground objects and their motions.

III. Overview

Temporal video segmentation, which can segment a whole video into independent video sequences, each with a different (possibly moving) background shot, is a well-studied problem [11], [12]. We assume that such segmentation has already been performed. We focus on vectorization of a single sequence, which comprises a static background, possibly with camera motion relative to it, and foreground moving objects. Such static backgrounds are widely used in cartoon making.

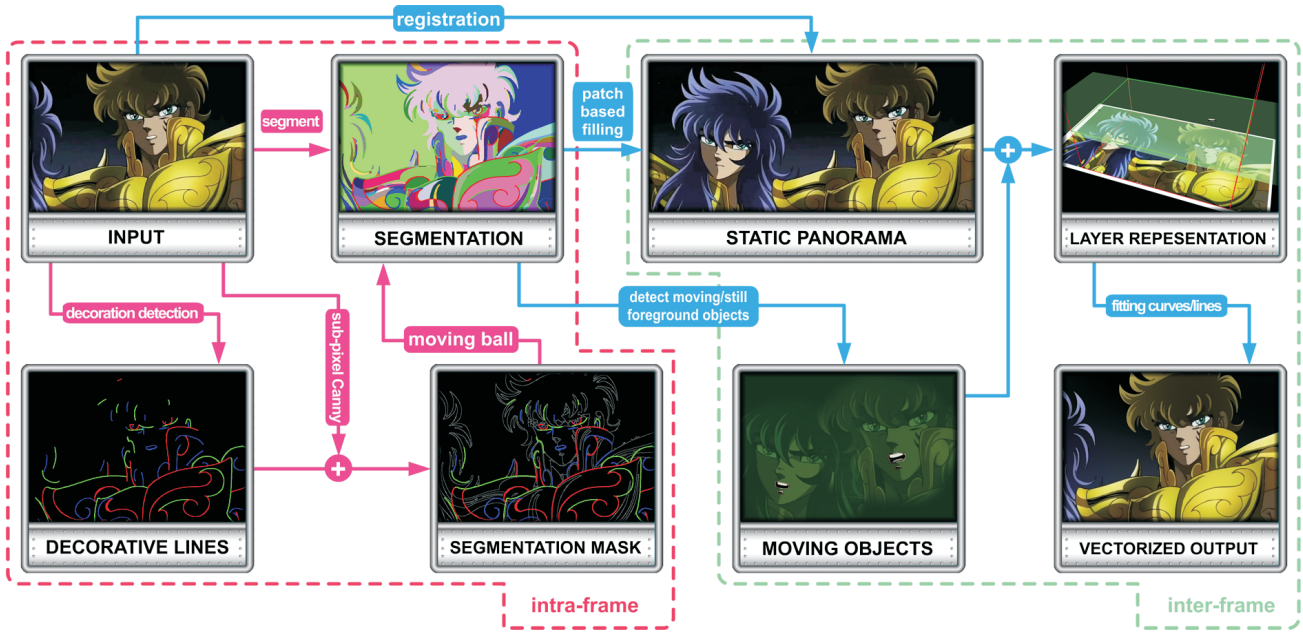


Fig. 3. Vectorization framework. The input at top left is the original cartoon video, and the output at bottom right the vectorized animation.

Fig.3 shows the framework of our system. We assume that the input is a raster 2D animated cartoon, which may have been regenerated from an original compressed using a lossy compression method. We vectorize a raster cartoon sequence as follows:

- In each frame, decorative lines are detected first, and these together with edge detection results are used to build a mask. The image is then segmented into regions using a trapped-ball method, controlled by this mask.
- To achieve inter-frame coherence, the frames in the sequence are registered by finding the homography following the approach in [13]. A static panoramic background image is reconstructed by first initializing it with unchanging areas, and refined by adding regions identified as belonging to the background. The moving objects are extracted as a foreground layer, together with their between-frame motions.
- The background and foreground key objects are vectorized: their boundaries are represented as curves and their interiors filled using quadratic (or potentially any other) color models, and the vectorized animation is output.

We now consider particular aspects in detail.

IV. Single Frame Decomposition

An important requirement for improving visual coherence is to decompose the cartoon image into relatively few meaningful objects. Typically, cartoon images contain two types of objects: colored regions, and decorative lines—see Fig.2. Colored regions need not have a uniform color, but can be described using some simple model for color as a function of position, e.g. a quadratic model.

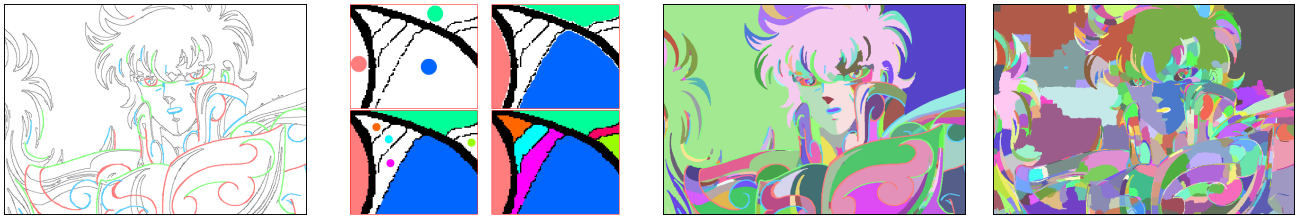
Due to the typically large differences in shading between neighboring regions, edge information provides a strong hint for meaningful region segmentation. We thus use a Canny edge detector [14] to extract edge information as a guide to image decomposition.

However, directly using edge information to find region boundaries and decorative lines has various challenges:

- especially when processing compressed video, whatever parameter settings are used, use of any simple edge detector typically results in various edges being missing, others containing gaps, and spurious noisy edges,
- wide decorative lines produce two parallel edges, which can lead to many thin, and short, over-segmented regions.

We take these into account as follows. First, the decorative lines are separately extracted in each frame. The pixels covered by these lines are then combined with other pixels belonging to edges detected in the frame, using a standard Canny edge detector, to give a mask to control image segmentation.

Segmentation of each frame is performed using a novel *trapped-ball model*, allowing us to overcome imperfections in the edge detection results. The idea is that a suitably large ball moves around inside a region delimited by the edge mask. Even if the edge mask locally has a gap, the ball is too big to go through it, and so will be limited to the initial region. In practice, we use morphological operations from image processing to implement this conceptual framework, as detailed later. Unlike many segmentation methods, which require regions to be of a uniform shade, we use a model allowing spatial variation of color within each region. This is by default a quadratic model, but any type of model could be used, depending on user requirements on computation time and image quality.



(a) Segmentation mask (b) Trapped-ball segmentation (c) Our segmentation result (d) Mean shift segmentation result

Fig. 4. Single cartoon frame segmentation

We now give further details of decorative line detection and trapped ball segmentation.

A. Decorative line detection

Unlike real video, which contains just *edges* between regions of different colour, cartoons also often contain drawn *decorative lines*, with small but finite width. The former correspond to a high clour gradient, while the latter are specifically drawn by artists to convey structure, motion, and other artistic meaning. Some decorative lines are located within regions, while others may be drawn at region boundaries for emphasis. As we noted previously, edges and decorative lines give strong hints for cartoon frame decomposition, but normal edge detectors like the Canny edge detector are not suited to decorative line detection. Using such a detector would find two edges, one either side of the decorative line, leading to the decorative line being considered to be a narrow region. Furthermore, edge detectors typically lead to results with gaps, so to find these narrow regions, we would need either an edge-linking method, or a flood-filling method which can ignore gaps of a certain size, to produce connected decorative lines. To avoid these problems of producing narrow regions, and the difficulty of connecting them, we detect decorative lines separately from edges use a specialised approach. Sýkora's method [4]–[6] also detects decorative lines forming outlines of regions, but they rely on an artistic style which produces *closed* decorative lines. Unfortunately, decorative lines do not take this form in most recent cartoon art styles: some regions are enclosed by decorative lines, but others are only delimited by edges in the image (i.e. places with a high gradient in colour). Sýkora also assumes that decorative lines are the darkest regions in a cartoon, which again does not always hold. We use a more general approach to decorative line detection which overcomes these problems, as well as being faster and giving more accurate results. We first find the points on the centreline of each decorative lines (which we call *line points*, to distinguish them from edge points), and then link them. This accurately locates the line. After that, we estimate the width of the line using the repsonses of a Canny edge detector on either side of the lines. We now explain in detail:

- First, we find line points. We convolve the image with a *second* derivative of a Gaussian. Local maxima of the result correspond to line points, and are found by using non-maxima suppression as in the original Canny edge detector [14]. This also gives a local line orientation for each point.
- We next wish to link the line points to get contiguous lines which are as long as possible. Steger's linking algorithm works by repeatedly adding points to the current line provided that their orientation is consistent with that of the line. Steger's algorithm can break down due to noise in the orientation estimates. Suppose the orientations at successive points along the line are $\ldots, \alpha_{k-2}, , \alpha_{k-1}, \alpha_k$. Steger uses α_k as the orientation of the line to find the next point which will be added to the line. We overcome noise by replacing α_k by a smoothed version α'_k, using a weighted average:

$$\alpha'_k = \sum_i \omega^{k-i} \alpha_i / \sum_i \omega^{k-i}$$

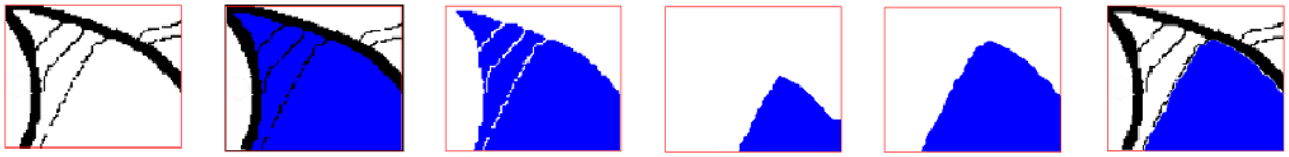
  when deciding whether to add the next point to the line. Closer points to the current point k are thus given greater weight. The tuning parameter ω should be in the range $(0,1)$; we have used $\omega = 0.3$ in all our examples.
- A traditional Canny edge detector is used to find both edges of each line, from which we can estimate its width. For robustness, we check for and discard outliers, assuming the width changes smoothly, and also discard lines of widths larger than 10 pixels.

Sample output is shown in Fig. 4(a). Each colored line is one decorative line. Detecting such decorative lines separately and representing them explicitly as lines with a given width greatly improves the quality of the final vectorized result, as otherwise, many short, thin, regions are found instead, leading to an oversegmented result. Region boundaries where there is a sudden change of colour, i.e. at conventional edges, are the black edge pixels in Fig. 4(a).

B. Trapped-ball region segmentation

Next, we find the regions. Our algorithm is based on the desire to find large regions, each represented by a single consistent color model. We must also take into account while that the segmentation mask found in the previous step gives guidance as to where region boundaries should be, in practice the mask will not form contiguous, 'watertight' boundaries, precluding simple floodfilling.

One possibility is to try to perform edge-linking [15] on the information in the mask. Unfortunately, such methods cannot



(a) (b) (c) (d) (e) (f)

Fig. 5. Trapped-ball segmentation achieved by using morphological operations

give guarantee to close all gaps, and even a single failure may result in two large regions being inappropriately fused.

Instead, to meet our above requirements, we have designed a novel *trapped-ball* algorithm particularly suited to cartoon segmentation. The basic idea behind our segmentation method is to move a ball around, limited by the mask; each separate area in which the ball may move determines a region of the image. Fig. 4(b) illustrates *trapped-ball segmentation*. A *best first* rule is used: at the start of the algorithm, we use a large ball, and then iterate with balls of smaller and smaller radius. Choice of ball sizes is discussed later. In practice, the same effect as moving a ball can be achieved by using morphological operations [16] from image processing, as we shortly explain.

In detail, the following steps are iterated:

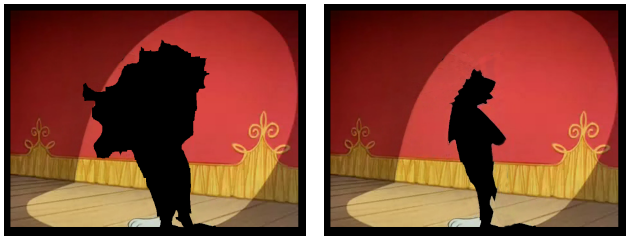
1) **Trapped-ball Filling** A ball-shaped morphological structuring element of radius R is used. Fig. 5 illustrates the morphological operations in use on a local area of the whole image. Given the original mask (a), we first use flood-filling, giving (b), or without the mask, (c). We then use a circular structuring element having the size of the trapped ball, and perform first an erosion operation on (c), giving (d), then a dilation operation on (d) to give (e), which is the result of the first pass of our trapped ball filling; we again show this result with the mask for reference in (f). (Note that the blue region is presumed to extend outside the red box, explaining why erosion does not shrink the blue region at the edges of the red box).
2) **Color Modeling** After using the above process on the whole image, multiple large regions result. We assume that a single colour model is an adequate fit to each such region, an assumption which works well in practice, in terms of producing visually acceptable results, even if counterexamples are not hard to construct—certainly, any sudden changes in color would have produced edges. By default we use a quadratic color model, in HSV color space, as does Ardeco [1]. For each channel, the color of the i^{th} region at pixel (x,y) is modeled by the function $f_i(x,y) = a_{i0} + a_{i1}x + a_{i2}y + a_{i3}x^2 + a_{i4}xy + a_{i5}y^2$. The parameters a_{ij} are determined by solving a linear system equation for all pixels belonging to region i. As each region is larger than the ball, we can be certain there is adequate data to fit a color model to it.
3) **Region Growing** We next grow the regions generated by trapped-ball filling. This is necessary because the ball cannot penetrate into narrow pockets belonging to regions, or pass through narrow necks separating parts of the same region. As we do so, we use two constraints. Firstly, edge pixels may not be added to a region. Secondly, pixels added to a region should agree in color with the prediction given by the color model for the region, within the limits of human perception. In the implementation, we label each pixel to indicate which region it belongs to, null indicating not yet assigned to any region.
Let us define the reconstruction error of a labeled pixel to be the difference between its colour and the color predicted by the color model for that region and pixel position. To perform region growing, we put all region pixels at the outline of the region into a priority queue, based on their reconstruction error given that they are already labelled; the queue is sorted by reconstruction error. Then, we repeatedly pop the pixel with minimum reconstruction error from the queue. We next check if the same label could be used for each unlabeled neighboring pixel, with a sufficiently small reconstruction error (in practice, we require the difference in intensity to be below 20). If so, this pixel's label is updated, and it is added to the priority queue. We repeat these steps until the queue is empty, or the reconstruction error is too large.

As the above steps are performed, all pixels newly given labels are added to the mask, so that subsequent iterations do not consider pixels which are already labelled.

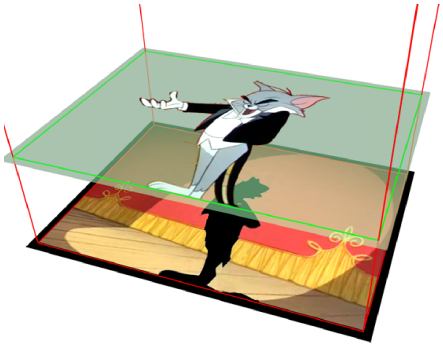
Some pixels may remain unlabeled, so we reduce the ball radius by 1, and iterate, labeling and growing smaller regions, and so on, until the ball radius reaches 1. By now, all pixels must have been labeled, and the image fully segmented. Fig.4(c) illustrates our segmentation result for our earlier example, and compares it with the *mean shift* segmentation result (Figs.4(d), 4(a)). Clearly, our result has fewer regions, and the regions we find are more meaningful.

Theoretically, the initial radius used should be equal to the maximum distance of any pixel from some edge pixel; but in practice, to make the algorithm faster, we use a value of 8 pixels, as in our experience, any gaps in the mask are almost always smaller than this. Clearly, if highest quality results are desired, the user can experiment with different initial ball radius settings.

As a final step, we wish to get rid of regions which are too small. A user chosen 'fine detail' parameter in the range 0 to 200 pixels decides the minimum permissible size of a region.



(a) Stable background pixels (b) Final filled background



(c) Foreground over background

Fig. 6. Illustration of background filling and layer representation

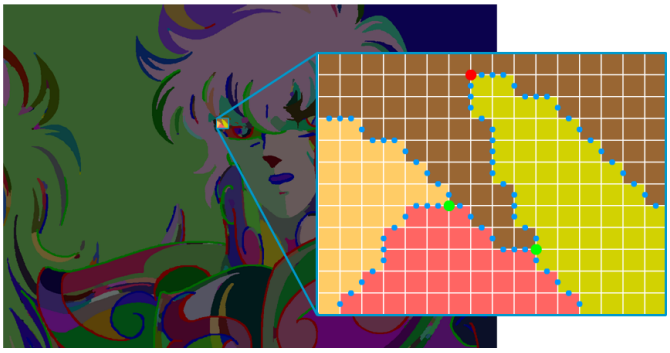


Fig. 7. Vectorization of segmentation results

We merge any regions smaller than this with the neighboring region with the most similar color model. This parameter helps to provide a balance between file size and quality in the final result.

V. Inter-frame Coherence

To achieve a high-quality vectorized result, we must avoid even small amounts of flickering due to lack of temporal coherence. This is especially important in the background, as it is static, making flicker more apparent. Most methods for achieving temporal coherence in video are based on global optimization, and do not consider the particular requirements for background coherence.

Registering the images in a sequence allows us to locate each frame I_i in a global coordinate system by transforming it to $\tilde{I}_i$ via a homography H_i. To do so, we make various assumptions: (i) the foreground area is not too large compared to the whole image, and (ii) the background motion comprises a translation and scale. While these may sound restrictive, they generally hold for many cartoon animations. Under these assumptions, registration methods such as SIFT [13] matching or even brute force search work well. Using RANSAC, we can detect outliers and overcome the jitter between frames. We obtain visual coherence by separately reconstructing the background as a single static panoramic image and extracting the foreground objects in the global coordinate system.

We now discuss how background and foreground objects are treated separately.

A. Background filling

Generally, a cartoon comprises several scenes. Each scene is a sequence of frames with a constant background, on top of which moving foreground objects are placed. A camera motion (translation or zooming) is then often adding to produce the final cartoon scene. We remove the camera motion by registration, resulting in a background which should be entirely static. We now discuss how we find the background of such a scene. We use our per frame decomposition results as input, and assume that each region found will either belongs to the background or the foreground. Furthermore, parts of the background may be occluded by foreground regions in each frame. We first use a background initialization method based on estimating the probability that each region belongs to the background, by considering its temporal consistency, and then use a patch based filling method which selects background regions while dynamically changing the probability map.

Let B be the global background image, and M be a probability map of the same size which estimate the probability for each pixel that it belongs to background. Initially M_q is set to 0 for every pixel q, meaning that B_q is unknown.

We now perform background image initialization, and patch-based filling, as follows:

1) **Background image initialization**. Taking each image $\tilde{I}$ in turn, we find the set S_q of pixels corresponding to each pixel B_q, by extracting the corresponding pixel P_{iq} in each $\tilde{I}_i$, if such a pixel exists. B_q is set to the median color of S_q, and M_q is set as follows:

$$M_q = \frac{|\{S_{iq} : |S_{iq} - B_q| < T_c, \forall S_{iq} \in S_q\}|}{|S_q|},$$

where T_c is the limit of visual ability to distinguish color differences, which we take to be 20 units. If M_q is sufficient large, e.g. larger than 90%, we regard B_q as a 'stable' background pixel, in that its colour is more or less unchanging over time. Fig.6(a) shows the 'stable' pixels belonging to the background for a particular video.

2) **Patch-based filling**. The background image is next augmented with per-frame segmentation results. This step recovers those background are that are occluded by foreground regions at various times. We examine all regions, and compute the probability that they belong to the background, where P_{ij} is the probability for region

Fig. 8. Vectorization results. Columns 1, 3: original video frames; columns 2, 4: corresponding vectorization results. (a), (b) Two clips from Tom & Jerry; (c) A clip from Naruto, with a panning camera.

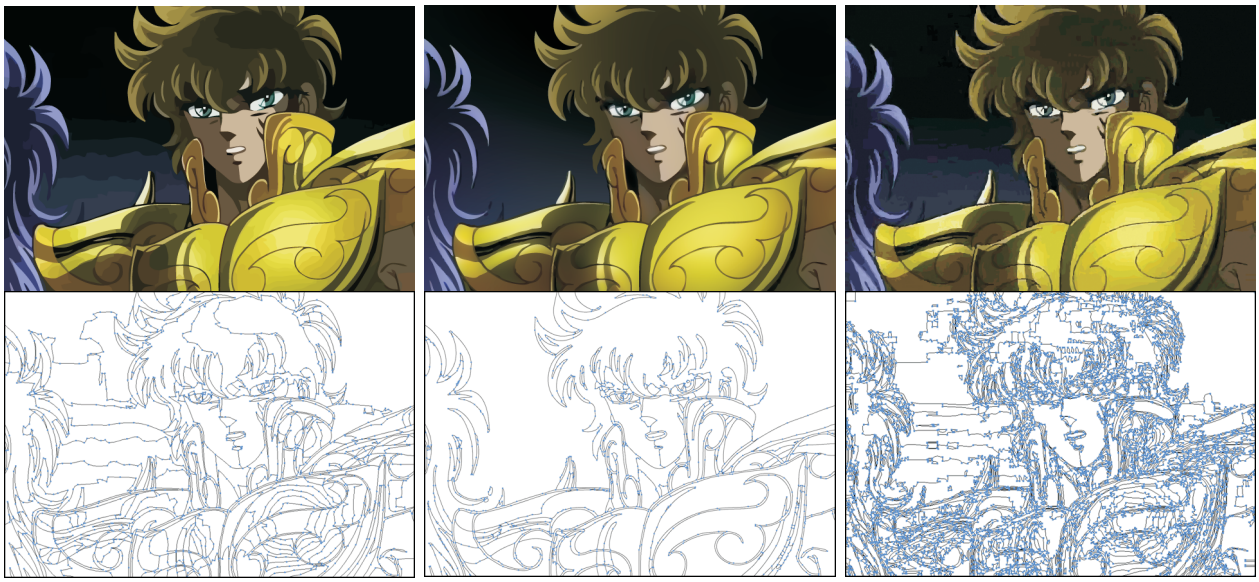


Fig. 9. Top: vectorization results using VectorMagic, our method and Adobe Illustrator Live Trace. Bottom: region boundaries.

(a) Composition of two clips (b) Object Editing

Fig. 10. Editing: (a) taking a character from one scene and putting him in another, (b) changing the shape of the character's ears

R_{ij}, the j^{th} region in frame i found by averaging the probabilities for its pixels:

$$P_{ij} = \sum_{|P_{iq}-B_q|<T_c, \forall P_{iq} \in R_{ij}} M_q / |R_{ij}|.$$

Higher P_{ij} means region has a larger probability that belongs to background.

All regions in all frames are next sorted in descending order of probability. Pixels belonging to the region with highest probability are added to the background, and M is updated accordingly. The background probability of each region remaining in the list is then updated and the list resorted. We then consider the remaining region with highest background probability, and so on. This process is quick as only a few regions need to be updated and resorted each time. This process terminates when either the background image has been completed, or when the highest probability of any remaining region being background is below some threshold value, at which point remaining objects in the list are considered to be foreground objects. We usually use 0.3–0.4 as this threshold, dependent on the quality of the input video. Too low a value may incorrectly fuse background and foreground areas, while too high a value means that the detected background will be smaller than it could be, with certain background areas being treated as foreground regions at higher cost. Generally, the lower the threshold that can be used, the better, as having a larger background area will provide better temporal coherence in the results.

At the end of this process, the background image may still contain holes where foreground objects *always* occlude the background. Clearly, such holes are irrelevant. Fig.6(b) shows the results after augmenting the background in this way.

While automatic background image filling usually works well, certain background areas may be uncovered in only a few frames, and thus cannot be detected automatically. Simple user interaction is used to add such regions to the background. This interaction needs only a few mouse clicks, combined with scanning through the video, and does not take long. Also, the user may remove regions which have been inappropriately assigned to the background image (e.g. the stationary feet of a cartoon character, the rest of whose body is moving). Both processes more meaningfully separate the cartoon into foreground and background.

B. Foreground extraction

Due to the fact that cartoons are hand drawn, the shapes of objects can be slightly different between neighboring frames, even if unchanged semantically. This issue hinders provision of temporal coherence. It is important to identify corresponding regions whenever possible in successive frames, both for high-level purposes such as editing, and compression of the final video.

The motion of large foreground objects between adjacent frames can easily be detected by tracking or matching, as the motions are typically small. Small regions are more tricky to track, but tend to move along with neighboring large regions.

We follow the approach in [10], which groups pixels according to their motion, with the difference that we use large foreground objects as seeds for region tracking. We compute their homographies and place them, together with neighboring regions with similar transformations, into groups, each in a different initial layer. This may result in allocation of certain regions to more than one group, so we use a graph-cut algorithm [10] to decide the final grouping of regions, as well as occlusions.

Regions which are successfully tracked are replaced in subsequent frames by the original region plus the corresponding transformation: the segmentation results found in subsequent frames are discarded. The groups found typically have semantic significance, which is useful for further editing and other high-level processes.

C. Vectorization

We now have an intermediate representation with regions assigned to foreground layers or the background. To achieve a seamless and accurate vectorization result, the *mutual* boundaries of adjacent regions in the *same* layer (and hence which are moving together) must be determined, as well as other boundaries. Boundary curves are cut at endpoints, which are points where more than two regions meet (green points in Fig.7) or boundary points with a local maximum of curvature (red points in Fig.7). We fit one or more cubic Bézier curves to each boundary point sequence between adjacent endpoints, to achieve a given tolerance. We use tolerances between 4 and 20 pixels, allowing a trade-off between quality and file size.

Solid regions are defined by their boundary curves and color model parameters. Decorative lines are represented by a polyline with accompanying width and color at each vertex. Regions in the background layer only need to be vectorized once, while for key objects in the foreground, its position and transformation in each frame are stored. It is straightforward to convert our vectorized output into Adobe Flash swf format for use in the Flash vector animation editor.

VI. Experimental Results

Fig.1 and Fig. 8 show various cartoon vectorization results. In the classical cartoons (Figs. 8(a,b)), a flat color model is sufficient for modeling region colors; in modern 2D cartoons (Figs.1 and 8(c)), a quadratic color model is more typically required.

By taking account of the particular relatively simple nature of cartoons, our algorithms are efficient compared to other methods, especially with respect to segmentation and inter-frame region matching, the latter because we are able to stably find large regions. On 640×480 input, segmentation typically

Video clip	Frames	Divx	Vec1	Vec2
Fig.1: Saint	91	1049kB	75kB	41kB
Fig.8: Tom	80	1022kB	542kB	283kB
Fig.8: Jerry	54	610kB	388kB	201kB
Fig.8: Naruto	120	1782kB	1255kB	633kB
Fig.10(a): Jerry	82	619kB	192kB	98kB

TABLE I
FILE SIZE COMPARISON (SEE TEXT).

takes 2–3 seconds per frame, background and foreground extraction take 4–8 seconds, and vectorization takes less than 1 second; times vary with complexity of scene, but around 10 seconds per frame is typical, which means that an entire cartoon can be vectorized in an acceptable length of time. In contrast, Ardeco [1] takes 72–125 seconds per frame for 512×512 frames, the Stanford VectorMagic [3] image result in Fig. 9 took 2 minutes, and the method in [2] takes around 4–15 minutes per object.

We next compare our one-frame vectorization results to those produced by Stanford VectorMagic [3] and Adobe Illustrator Live Trace in Fig.9. Our result gives more a visually appealing segmentation with fewer regions, which also incorporates decorative lines.

Figs. 10(a), 10(b) show how our results can readily be edited using Adobe Flash. The left hand example shows how Tom (the cat) has been extracted from one clip (see Fig.8(a)), and inserted between the background and foreground (Jerry the mouse, in bed) of another clip, using simple drag-and-drop. The right hand example shows a simple edit to Jerry's ear which was done on just two frames, and continues as the region is propagated through the video (the original image sequence is illustrated in Fig.8(b)). (Editing was needed on two frames as Jerry's ear underwent a large change in shape part way through the original input sequence—note that the original was hand-drawn).

Table I compares the file sizes of our vectorized output to the original DivX compressed raster videos. The table shows DivX 6 compressed video size (DivX), high quality vectorized output with decomposition detail level 0 and curve fitting tolerance 4 (Vec1), and acceptable quality with decomposition detail level 200 and curve fitting tolerance 20 (Vec2). Summarizing these results, when producing high quality output, our vectorization method leads to an average file size of about half that required by Divx 6 compressed video. This is further reduced by another factor of 2 for lower but still acceptable quality output. Due to the complex content of the Naruto video clip, less region correspondences were found between frames, and thus a lower reduction in file size was achieved. We note that if our method is used to accelerate network transmission of cartoons, a desired bit rate can be adaptively achieved by adjusting the curve fitting and fine detail tolerances mentioned earlier. See Fig. 13.

VII. DISCUSSION

A. Comparison

We provide a side-by-side comparison with the methods in [4]–[6], [1] and [2] using images taken from their work (see Fig. 11). Sýkora's method [4]–[6] imposes very strong constraints on the cartoons processed, i.e. regions must have thick, closed black outlines and flat color. As a result, their method failed on most cartoons we tried, making a direct comparison difficult. Our trapped ball model for segmentation means that we can handle region boundaries which need not be closed or clear, greatly increases the applicability of our technique.

Compared to Ardeco [1], we can produce almost the same quality of results even for non-cartoon images, despite these not being our main goal. In Fig. 11(b), the mean pixel difference between Lecot's method and the original image (L_2 norm in RGB color space) is 9 units, whereas in our result, the mean error is 4 units. The resulting vectorized output from Ardeco looks more stylized (or *posterized*) than our result: see in particular the face and shoulder of the woman, which have more noticeable steps in shading.

Price [17] and Sun [2] give methods for vectorizing images that contain objects with complex but smoothly changing textures; they use meshes to describe those textures. Their work thus has very different motivation from ours: cartoons often contain untextured regions with generally well-defined boundaries. (Considering texture as a particular kind of colour model would be an interesting extension to our current work). Both of these methods need users to manually create initial meshes for each object in the image, so would be very hard to extend to video. As a trade-off, these methods can provide very low reconstruction errors, of usually less than 1 unit per pixel, which our method cannot.

We note that often, input video clips suffer from compression artifacts, and our output may actually have higher visual quality than the input, so least-squares reconstruction errors are not really a satisfactory metric for comparing algorithms for vectorizing such data.

B. Limitations

Our method targets cartoons which have artificial artistic content, including regularly (but not necessarily uniformly) colored regions and decorative lines. Our method produces suboptimal results when:

1) Adjacent regions have similar color: boundaries may be incorrectly placed or regions incorrectly fused. For example, in Fig. 11(c) the shadow of the pepper in our result is poorly segmented), while in Fig. 12(b) poor segmentation results where light crosses the character's hair.
2) Regions are not well fitted by the kind of color model used. This will result in large reconstruction errors, as

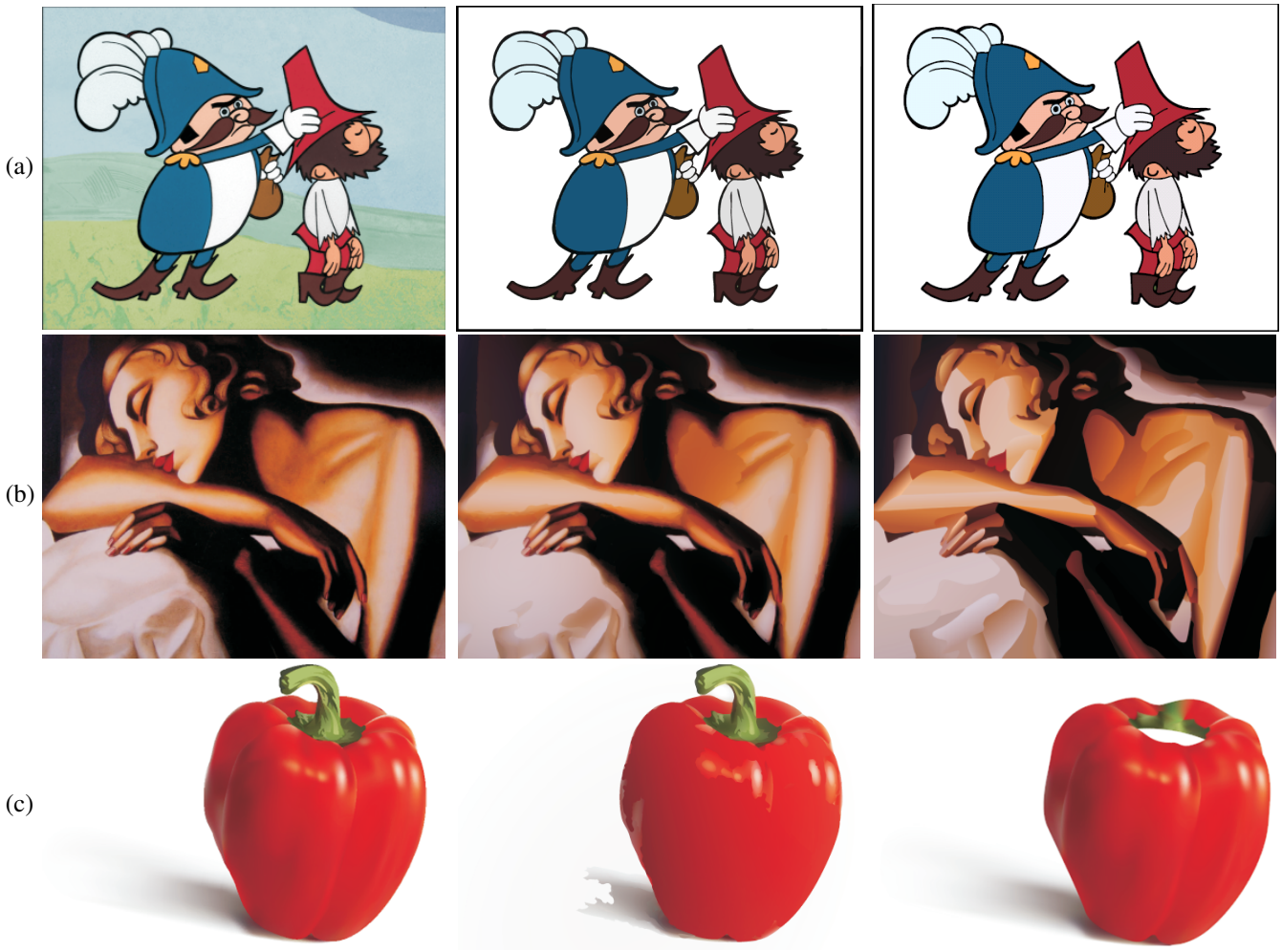


Fig. 11. Comparison with other vectorization method. Columns: original video frames, our results and others' results. Rows: results produced by Sýkora's method, Ardeco, and Sun's method respectively.

in Fig. 12(c). Here, the monster's body is covered in fur, and unsurprisingly, large coloured regions does not represent the texture of the fur well. Indeed, any kind of complex texture is not well handled by our current approach.

3) Foreground objects adjacent to the background have a similar color. This can result in foreground objects being assigned to the background. An example is Jerry's foot in Fig. 8(b). (Note that we compare the colors after color modeling. After color modeling, the region color of the foot is considered to be similar to that of the background.)

We note that certain art-styles like watercolor painting, and highly detailed cartoon clips with shining light effects as well as those containing complex textures may cause the above issues, and thus are not well suited to our method. We provide examples of such cases in Fig. 12. Nevertheless, while the results are suboptimal in terms of fidelity to the original, or quality of segmentation, we are still able to obtain meaningful output—our algorithm degrades gracefully.

In some cartoons, the background is much more complicated than the foreground (as it only needs drawing by the artist once, and more effort can thus be expended on it—e.g. see the forest in Fig. 8(c)). Vectorizing the background image would result in many small regions, and a large file size. One possible response is to produce a somewhat less detailed background by adjusting the region segmentation process to use a larger value for the fine detail parameter. (This may not be as bad as it sounds, as the user mainly concentrates his gaze on the foreground objects). Another solution would be to simply store the background as a bitmap—as this is a common background for all frames, this would add relatively little extra storage requirement to the final result.

VIII. Conclusions

We have presented a system for transforming 2D rasterized animated cartoons into vectorized representation. The output animations are visually flicker free, smaller in file size than the input, and contain large regions, potentially suited to applications such as editing an multimedia information retrieval. Decorative lines are output separately to colored regions.

Our system takes advantage of the particular nature of cartoons, to achieve high quality image decomposition with

Fig. 12. Cases producing poor results. First column: original cartoon images, second column: our vectorized output, third column: our decomposition result.

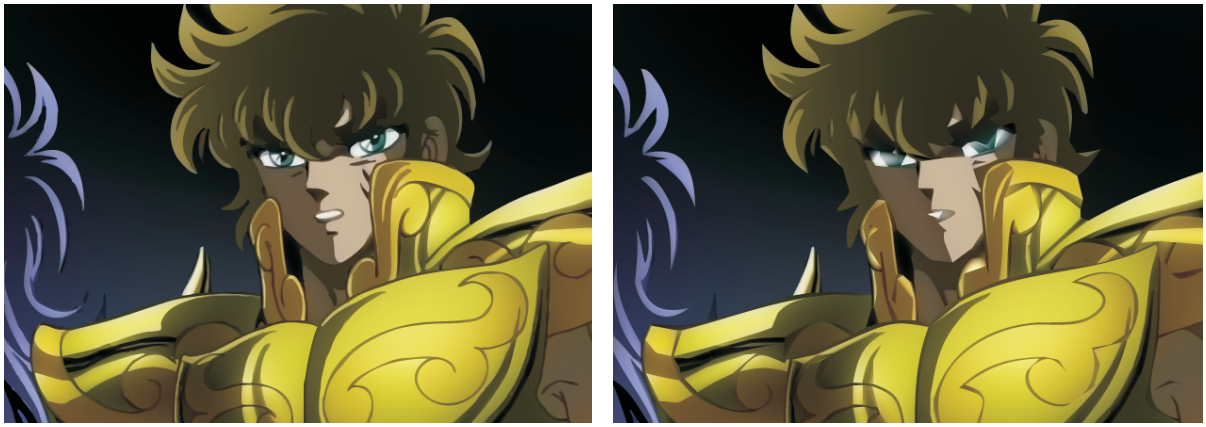


Fig. 13. Vectorized representations with different fidelity of detail. Data sizes: left, 24kB, right, 9kB.

more meaningful segmentation results compared to existing methods. At the same time, the segmentation supports arbitrary color models for each object. Nevertheless, we may also apply our segmentation algorithm to real images, as shown in Fig.14.

A number of extensions would further enhance our system. The most desirable is to provide higher-level understanding of the foreground regions, by using a more sophisticated method to merge adjacent regions with semantic significance. This requires sophisticated handling of changes in shape of objects between frames to eliminate hand-drawn error. Furthermore, a whole film may have many scenes, shared amongst which there may be common characters in identical poses; these too should be identified, both to provide further compression, and to enhance the usefulness of the results for high level processes such as editing and information retrieval. Currently, we cannot handle cartoons with ill-defined region boundaries, caused e.g. by shining light, smoke, flames and motion blurs, nor can we handle cartoons with textured regions. We hope to extend our segmentation approach to cope with these.

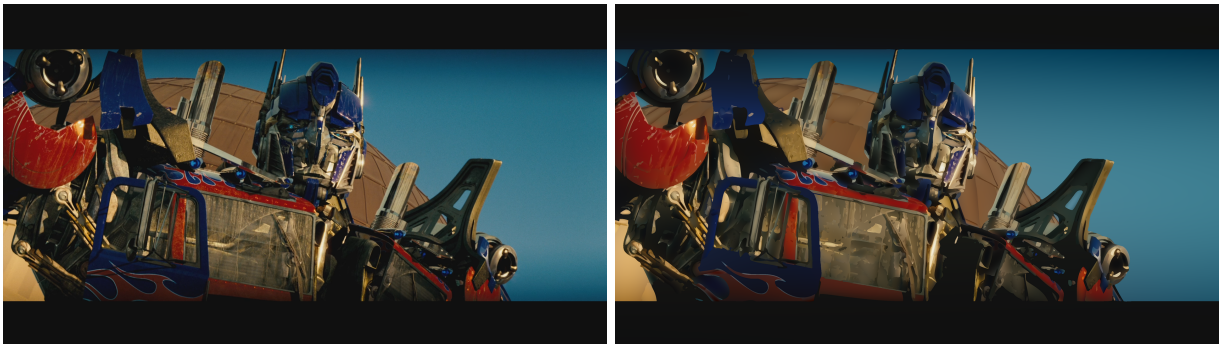


Fig. 14. Vectorization of a real video frame (1920×1080). Left: original. Right: vectorized result, with 2000 regions

ACKNOWLEDGEMENTS

The following cartoons were used in this paper: Tom and Jerry, Saint Seiya, Animatrix, Naruto, Gourd Brothers, Transformers and Monsters Inc.

This work was supported by the National Basic Research Project of China (Project Number 2006CB303106), the Natural Science Foundation of ChinaProject Number U0735001 and Specialized Research Fund for the Doctoral Program of Higher Education (Project Number 20060003057), and an EPSRC UK travel grant.

REFERENCES

[1] G. Lecot and B. Levy, "Ardeco: Automatic Region DEtection and COnversion," in *Proceedings of the 17th Eurographics Symposium on Rendering*, pp. 349–360, 2006.

[2] J. Sun, L. Liang, F. Wen, and H.-Y. Shum, "Image vectorization using optimized gradient meshes," *ACM Trans. Graph.*, vol. 26, no. 3, p. 11, 2007.

[3] http://vetormagic.stanford.edu/.

[4] D. Sýkora, J. Buriánek, and J. Žára, "Colorization of black-and-white cartoons," *Image and Vision Computing*, vol. 23, no. 9, pp. 767–852, 2005.

[5] D. Sýkora, J. Buriánek, and J. Žára, "Sketching cartoons by example," in *Proceedings of the 2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pp. 27–34, 2005.

[6] D. Sýkora, J. Buriánek, and J. Zára, "Video codec for classical cartoon animations with hardware accelerated playback," in *First International Symposium of Advances in Visual Computing*, vol. 3804, pp. 43–50, Springer, 2005.

[7] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

[8] C. L. Zitnick, N. Jojic, and S. B. Kang, "Consistent segmentation for optical flow estimation," in *International Conference on Computer Vision*, pp. II: 1308–1315, 2005.

[9] M. P. Kumar, P. H. S. Torr, and A. Zisserman, "Learning layered motion segmentation of video," in *International Conference on Computer Vision*, pp. I: 33–40, 2005.

[10] J. Xiao and M. Shah, "Motion layer extraction in the presence of occlusion using graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 1644–1659, October 2005.

[11] I. Koprinska and S. Carrato, "Temporal video segmentation: A survey," *Signal Processing Image Communication*, vol. 16, pp. 477–500, Jan. 2001.

[12] Y. Gong and X. Liu, "Video shot segmentation and classification," *icpr*, vol. 01, p. 1860, 2000.

[13] D. G. Lowe, "Object recognition from local scale-invariant features," in *International Conference on Computer Vision*, pp. 1150–1157, 1999.

[14] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[15] C. Steger, "An unbiased detector of curvilinear structures," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 2, pp. 113–125, 1998.

[16] M. Sonka, V. Hlavac, and R. Boyle, "Image processing, analysis, and machine vision," in *Chapman and Hall*, 1993.

[17] B. Price and W. Barrett, "Object-based vectorization for interactive image editing," *The Visual Computer*, vol. 22, no. 9, pp. 661–670, 2006.