Eurographics Symposium on Rendering (2006)
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

Ardeco: Automatic Region DEtection and COnversion

Gregory Lecot and Bruno Levy†

INRIA-ALICE

Abstract

We present Ardeco, a new algorithm for image abstraction and conversion from bitmap images into vector graphics. Given a bitmap image, our algorithm automatically computes the set of vector primitives and gradients that best approximates the image. In addition, more details can be generated in user-selected important regions, defined from eye-tracking data or from an importance map painted by the user. Our algorithm is based on a new two-level variational parametric segmentation algorithm, minimizing Mumford and Shah's energy and operating on an intermediate triangulation, well adapted to the features of the image.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation ; I.4.10 [Image Processing and Computer Vision]: Image Representation Hierarchical

1. Introduction

With the development of standards like Flash or SVG, vector graphics is more and more used on the Internet. This trend is going to accelerate even more in a near future, since new operating systems such as Windows Vista and Mac OS Tiger will use them to draw most components of their graphic user interface. By replacing the set of pixels used in raster images with a set of equations, vector images are usually more compact than bitmaps. A more important motivation to use them is the constant increase of screen resolution. With high-definition TV, 1920 x 1200 pixel is becoming the standard. Printing on large media without introducing pixel artifacts is another important motivation. For those reasons, it is useful to use a resolution-independent representation of images. Another interesting property of vector image is that it is very easy to edit, modify and animate them.

† {lecotllevy}@loria.fr

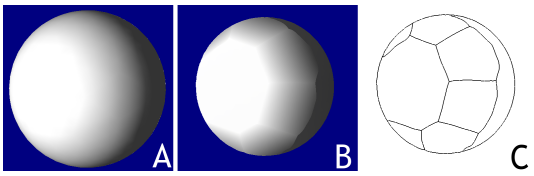


Figure 1: *A: original bitmap; B: vector approximation (8 primitives with linear gradients); C: the primitives.*

However, despite the existing semi-automatic tools (e.g. Silhouette and Vector Eye), producing a vector image remains a user-intensive process, similar to 2D Computer Aided Design. Our goal is to automatically produce a vector image (e.g. in SVG) from a bitmap image. This is typically a difficult *image abstraction* [Hae90] and *non-photorealistic rendering* [GG01], [Her01], [GCS02], [CRH05] problem. The main difficulty comes from the highly constrained nature of the target representation, that uses linear and quadratic gradients to represent colors. As a consequence, not only the algorithm needs to detect boundaries in the initial image, but also it needs to approximate complex color variations by combining several gradients. Figure 1 gives an intuition of the problem: the algorithm needs to both detect the boundary of the sphere and organize the linear gradients inside it to reproduce the shading variations. To achieve this, as done in [Her01] and [CRH05], we generate primitives in a way that minimizes an objective function. In contrast with [Her01], that distributes a large number of *brush strokes* over the image, we generate *regions* filled with linear or quadratic gradients. Note that our fully automatic algorithm may be "blind" to human-specific saliency, such as the eyes of characters. For this reason, we include the possibility of taking user-defined importance into account. This importance map can be manually drawn, or acquired by an eye tracker.

Contributions

- ◇ Our method creates a vector image that approximates a bitmap image. The so-constructed vector image is composed of a set of regions delimited by cubic splines. Each region is filled with a constant color, a linear or a circular gradient. In addition, user-defined importance can be taken into account;
- ◇ The kernel of the algorithm is a novel unsupervised parametric segmentation method, that recognizes higher-order gradients in the original image. Therefore, subtle shading details and highlights can be reproduced in the vector image;
- ◇ To accelerate convergence, we introduce an intermediate "trixel" data structure, storing pre-integrated covariance matrices;
- ◇ These trixels are computed by a generic rasterization algorithm, that efficiently and accurately computes integrals over an image, it is more efficient by several orders of magnitude as compared to [Sec02];
- ◇ Ardeco uses simple parameters (maximum number of trixels and maximum approximation error) ;
- ◇ It minimizes Mumford and Shah's functional by generalizing Variational Shape Approximation to higher-order functions, robust estimators and pre-integrated covariance matrices;
- ◇ **Limitations:** Ardeco is slower than existing vectorization software (minutes instead of seconds). This may be the price to pay for higher-order gradients. It is also not as texture-resistant as supervised segmenters.

2. Previous Work

Image abstraction and stylizing

By the type of primitives it uses (shapes filled with constant colors or gradients), vector graphics favors expressing images using a very specific graphics style. For instance, web surfers are familiar with the specific look-and-feel of web sites designed with Flash. This style is similar to the "art deco" painting movement of the 1920's (see e.g. Figure 2-A), using geometric shapes and simple color gradients. Since the "painting style" of the original image will be modified by the way important primitives are chosen and unimportant details are ignored, our method belongs to the *image abstraction* and to the *non-photorealistic rendering* domain. Image abstraction was made popular by Haeberli's "painting by number" paper [Hae90]. Besides Haeberli's play on words, the result of our method is very similar to the original "painting by number" play sets (see Figure 2-C and Figure 11 at the end of the paper). In a certain sense, when reproducing an image, our method will operate as an artist would do, by decomposing the image into a set of simple primitives. Perception theory is a key aspect in image abstraction and stylization. An abstract image should put to the fore important features, and make the structure of the image clearly appear to the reader. De Carlo *et. al* have presented in [DS02] a method for revealing this structure, by stylizing an image based on importance acquired from an eye-tracker. This shares some common points with our work, with the exception that we produce vector images and that our algorithm can recognize higher-order gradients. Our algorithm applied to De Carlo's images and taking into account his eye-tracker data is demonstrated in Section 5. A similar *video tooning* method was presented in [WXSC04], based on the mean-shift segmentation algorithm with an anisotropic kernel. The mean-shift algorithm is discussed in the next paragraph.

Segmentation

The kernel of our method is a segmentation algorithm, which is a research topic for which much time and effort has been devoted. A complete review of all the possible methods would be well beyond the scope of this paper, for this reason, we focus only on the most popular ones and on the ones related with our method. Computing a segmentation of an image means partitioning it into a set of regions (also called segments), with uniform visual aspect. By "uniform", different methods mean different definitions, ranging from low-level properties (e.g. color) to higher-level properties and visual cues from Gestalt and perception theory (e.g. texture). Mumford and Shah's functional [MS85] give a general formulation, used by a wide variety of methods. As explained in Section 3, our method minimizes an expression of this energy.

Segmentation methods can be classified in different ways. First, one distinguishes supervised methods, that use a training set segmented by human users to "learn" what a "good" segmentation is (see e.g. [MFTM01]). Unsupervised methods solely use a mathematical definition of uniformity for the regions. Our method belongs to this latter category. Second, in parametric methods, regions are represented by a set of coefficients (e.g., coordinates within a certain function basis), whereas non-parametric ones do not use any underlying model function. Since our target vector representation is composed of well-specified functions (i.e. SVG/Flash/Postscript gradients), our method belongs to the parametric category.

Then, two orthogonal families of methods can be distinguished, in function of what they try to optimize (boundaries or regions):

Boundaries optimization: To optimize the boundaries, one of the possibilities is to recast the problem as a classical min-cut/max-flow problem, well known in graph theory. This family of methods is very popular in texture synthesis (see e.g. [KSE*05]). Graph-cut was also used to construct an intermediate "super-pixels" data structure in [RM03], used later in [HEH05] for 3D model popup from photos. Our method also uses an intermediate data structure and a two-level algorithm. The differences with super-pixels are

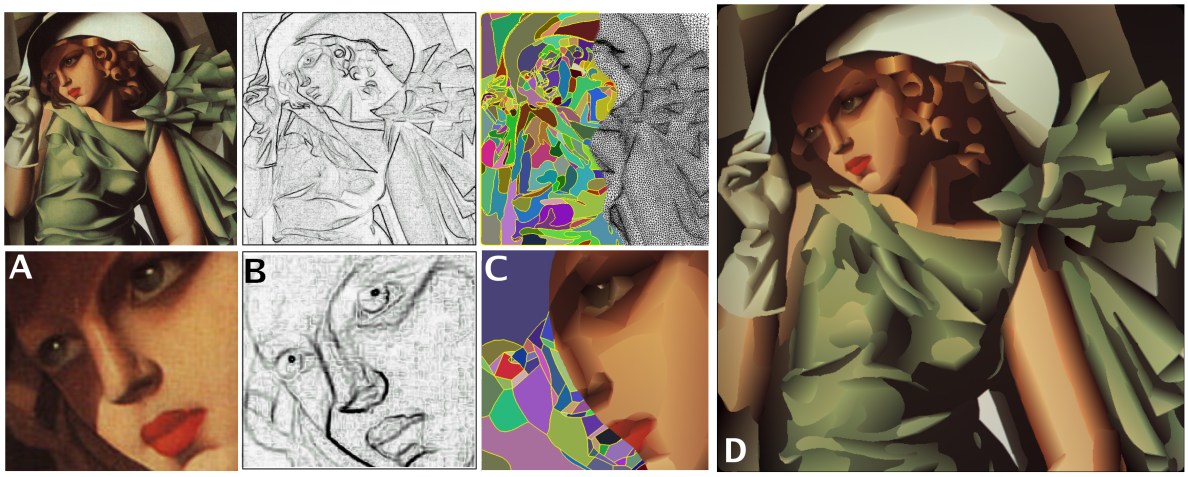


Figure 2: *The principal steps of Ardeco to convert a bitmap image into vector primitives. A: original image; B: saliency estimation; C: saliency-adapted triangulation (trixels) and partition into regions; D: final result (240 cubic splines filled with gradients)*

discussed in Section 3.2. The SVG community also developped algorithms based on an intermediate triangulation [BBNG05], similar to ours. The main difference is that they perform local updates, whereas we minimize a well-defined global energy functional. The boundary can also be represented in parametric form, by the so-called "Snakes" [KWT88], and optimized by an adpated expression of Mumford and Shah's functional. The main limitation with snakes is that it is difficult to change the topology of the boundaries. To relax this constraint, the boundary can also be represented and optimized in implicit form (see e.g. [OF01]).

Regions optimization: Giving a formal definition of "good" segmentations is a difficult task, involving high level notions from perception theory. Some methods learn these notions from human-segmented data bases [RM03]. Another possibility is to use variational approaches. In this latter category, the mean shift method [CM99] is one of the most popular. Its efficiency comes from the generality of the underlying principle. The idea is to optimize an objective function of all the pixel values in feature space, defined so that its minimizer has constant values over the regions. To exhibit such an objective function that detects the most significant regions, the mean shift approach characterizes the local maxima of the density function in feature space, acting as attractors during the optimization process. This density function is defined from the discrete set of samples, smoothed by a density estimator. To improve the results, some refinements of the method were described, such as using an anisotropic kernel for the density estimator [WTXC04]. It still requires some bandwidth parameters, but better adapts to the signal. Our method will be compared to mean shift in Section 5. The main differences are that our method is parametric, does not require any bandwidth parameter and recognizes higher-order gradients. In the context of mesh segmentation, the VSA method (Variational Shape Approximation) described in [CSAD04] finds the plane equations that best approximate the initial mesh. VSA and its relations with our method are discussed in more details in Section 3.1.

3. Ardeco

Our goal is to automatically produce from a given bitmap image the set of vector primitives that best approximates the original image. In other words, we want to translate the image from the "language" of bitmap images (i.e., in terms of pixels) into the "language" of vector graphics (i.e., in terms of vector primitives). Those vector primitives are closed regions, bordered by cubic splines, and filled with a constant, linear or quadratic variation of color (all possible circular gradients and their linear transformations used in vector graphics corresponds to a quadratic function). We will first introduce the pixel-based version of the algorithm (Section 3.1), then explain how to accelerate the algorithm by constructing an intermediate "trixel" data structure (Section 3.2) and using it (Section 3.3). The post-processing algorithms used to produce a standard vector file (e.g. SVG) from our vector representation are described in Section 4. We show some results in Section 5 before concluding and giving some suggestions for future work.

3.1. Pixel-based algorithm

Using a particular color space (in our case, Luv and YIQ give the best results), we now consider an individual channel. We will explain later how to combine the three channels. For a given channel, let $g(x,y)$ represent the intensity level at a given pixel (x,y). The image will be parti-

tioned into a set of n regions R_i, and intensity will be approximated in each region using a linear combination of basis functions $(\phi) = (\phi_1, \phi_2 \dots \phi_m)$. For instance, for linear gradients, $(\phi) = (1, x, y)$, for quadratic gradients, $(\phi) = (1, x, y, x^2, xy, y^2)$, and for constant colors, $(\phi) = (1)$. In the generated vector image, the intensity $f_i(x,y)$ in region R_i will be given by a linear combination of the basis functions: $f_i(x,y) = \sum_{j=1}^{m} \alpha_{i,j} \phi_j(x,y)$.

Following the formalism described by Mumford and Shah [MS85], we aim at satisfying the three objectives they have formalized in their seminal paper:

- ◇ **fitting:** we will minimize a norm of the approximation error;
- ◇ **smoothness of the regions:** the constant, linear or quadratic functions that we use already satisfy this criterion;
- ◇ **minimal boundary length:** we use a variation of Lloyd's algorithm that partitions a given space into compact cells.

Given a set of n points $\mathbf{p} = (p_1, \dots p_n)$, referred to as sites, Lloyd's algorithm [Llo57] constructs a partition of the plane, defined by the Voronoi diagram of the sites (p_i), and minimizing the following energy functional:

$$F_{\text{Lloyd}}(\mathbf{p}) = \sum_{i=1}^{n} \int_{R_i} \left\| p_i - \begin{pmatrix} x \\ y \end{pmatrix} \right\|^2 dxdy$$

Note that the compactness of the cells also means that the length of the boundary is minimal. As a consequence, this energy corresponds to the boundary term of Mumford and Shah's objective function. We now study how to inject the fitting term in it.

Our goal is to partition the original image into a set of regions R_i, and compute in each region R_i a function f_i that best approximates the image function g. To simultaneously find the regions R_i and the coefficients $\alpha_{i,j}$ that define the functions f_i, we use a variation of Lloyd's formulation, similar to the one used by Cohen-Steiner *et. al.* in their VSA (Variational Shape Approximation) method [CSAD04], and including a fitting term. We generalize the VSA method in different ways, well adapted to our image-processing setting:

- ◇ we use higher-order functions f_i (or "proxies" in VSA parlance);
- ◇ VSA minimizes a quadratic deviation, that may be sensitive to outliers, frequently encountered in image processing. To be resistant to outliers, we use a robust M-estimator to compute the parameters $\alpha_{i,j}$ of the functions f_i;
- ◇ we keep the cells-compactness term in the energy functional. This corresponds to Mumford and Shah's boundary length term;

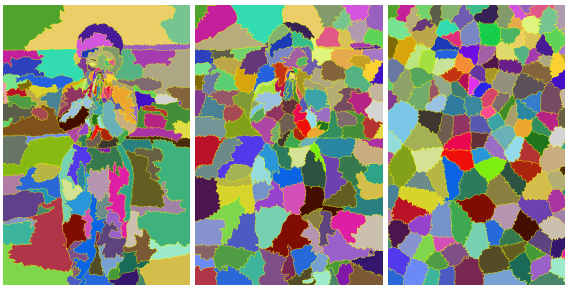


Figure 3: *Influence of the parameter λ, weighting the cells compactness criterion. From left to right: $\lambda = 0$ uses the fitting term alone; $\lambda = 0.75$ constructs compact cells aligned with the features; $\lambda = 1$ ignores the fitting term and constructs a Voronoi tessellation.*

- ◇ to improve efficiency, we will introduce a two-level algorithm, with an intermediate "trixel" data structure, that stores pre-integrated covariance matrices for a triangular group of pixels.

We find the optimum partition and approximation in each region of the partition simultaneously, by minimizing the energy functional given in Equation 1:

$$F(\alpha, \mathbf{p}) = \lambda F_{Lloyd}(\mathbf{p}) + (1-\lambda) F_{fit}(\alpha) \qquad (1)$$

In this energy functional, as usually done with methods derived from Mumford and Shah's formulation, the parameter $\lambda \in [0,1]$ corresponds to the user defined importance of boundary length with respect to fitting the data. Figure 3 shows the influence of the parameter λ.

The fitting term $F_{fit}(\alpha)$ is given by Equation 2:

$$F_{fit}(\alpha) = \sum_{i=1}^{n} \int_{R_i} \rho\left(r_i(x,y)\right) dxdy$$

$$\text{where:} \begin{cases} r_i(x,y) &= g(x,y) - f_i(x,y) \\ &= g(x,y) - \sum_j \alpha_{i,j} \phi_j(x,y) \\ \\ \rho(r) &= |r|^\nu / \nu \quad ; \quad \nu = 1.2 \end{cases} \qquad (2)$$

The function ρ defines an M-estimator, resistant to outliers (see e.g., [Zha96]). Note that if using $\rho(r) = r^2$ instead, F degenerates to the standard least-squares fitting. A family of possible M-estimators exists, in our case we use the $\mathscr{L}_p$ norm, that gives the best result (we also tried $\mathscr{L}_1\mathscr{L}_2$, "fair" and Huber's M-estimators [Hub81] that gave similar results but were either less efficient or less stable).

We first explain the basic algorithm, operating at the pixel level. We will then introduce the more efficient trixel-based algorithm (Section 3.2). Our algorithm computes a label image $region(x,y)$ and the coefficients $(\alpha_{i,1}, \dots \alpha_{i,m})$ defining the approximation f_i in each region (see Figure 4, next page). The algorithm is decomposed into two steps, applied repeatedly in an interleaved manner, until the approximation error reaches a user-defined threshold.

Figure 4: *Segmentations and image approximations obtained with the pixel-based Ardeco algorithm, using regions of degree 0, 1 and 2.*

The first step (Algorithm 1) greedily grows regions pixel by pixel, with a priority dependent on the fitting and on the compactness of the cell. The integer 2D array $region(x,y)$ stores the region labels.

Algorithm 1 : flood_fill()

S : priority queue $< (x,y), i, cost >$ sorted by $cost$
// initialize S with the sites
for $i = 1$ **to** n { push$(S,(x_i,y_i),i,0)$ } **end for**
// grow the regions
while not S is empty
  $(x,y), i, cost \leftarrow$ pop(S)
  if $region(x,y) = 0$
    $region(x,y) \leftarrow i$
    for each *neighbor* (x',y') **of** (x,y)
      if $region(x',y') = 0$
        $cost \leftarrow \lambda \left\| (x',y')^t - p_i \right\|^2 + (1-\lambda)\rho\left(r_i(x',y')\right)$
        push$(S,(x',y'),i,cost)$
      end if
    end for
  end if
end while

The second step (Algorithm 2) updates both the sites p_i and the parameters $\alpha_{i,j}$ that define the functions f_i.

Algorithm 2 : update_sites_and_fit()

for $i = 1$ **to** n
  // update the fitting
  $(\alpha_{i,1}, \dots \alpha_{i,m}) \leftarrow \operatorname{argmin} \int\limits_{x,y \in R_i} w(r_i^{(k-1)}(x,y)) r_i^2(x,y) dxdy$
  // update the sites
  $(x_i, y_i) \leftarrow \operatorname{argmin}_{x,y \in R_i} (\rho\left(g(x,y) - f_i(x,y)\right))$
end for

In this algorithm, $r_i(x,y) = f_i(x,y) - g(x,y)$ denotes the residual at the current iteration, depending on the unknown $\alpha_{i,j}$ parameters. The function $r_i^{(k-1)} = f_i^{(k-1)}(x,y) - g(x,y)$ denotes the residual at the previous iteration, and $w(x) = \rho'(x)/x$ denotes the weighting function of the M-estimator yielded by ρ. In more details, the parameters $(\alpha_{i,1}, \dots \alpha_{i,m})$ are updated by solving the linear system given Equation 3.

$$A_i \left(\alpha_{i,1}, \dots, \alpha_{i,m}\right)^t = b_i \qquad \text{where:}$$

$$\begin{cases} A_i & = & \int\limits_{R_i} w\left(r_i^{(k-1)}(x,y)\right) \Phi(x,y) \Phi^t(x,y) dxdy \\ b_i & = & \int\limits_{R_i} w\left(r_i^{(k-1)}(x,y)\right) g(x,y) \Phi(x,y) dxdy \\ \Phi(x,y) & = & (\phi_1(x,y), \dots \phi_m(x,y))^t \end{cases} \tag{3}$$

Equation 3 corresponds to the standard way of re-casting the computation of an M-estimator in terms of a sequential reweighted quadratic optimization (a variant of Newton's method to optimize a non-linear function). For the $\mathscr{L}_p$ norm, the w weighting function is given by $w(r) = |r|^{v-2} = 1/|r|^{0.8}$. As can be easily checked, the weighting function w quickly decreases with the value of the residual r, which means the importance of outliers is reduced. To initialize the estimator at the first iteration, we use the solution of the least-squares problem. Note that the dimension of the linear systems to be solved at each iteration corresponds to the dimension of the function basis (ϕ), which means simple direct solvers can be used. In our implementation, we use the Cholesky factorization with pivoting available in LAPACK. In the specific case of constant colors, we simply use the median instead of the M-estimator. The median is more robust (i.e. less sensitive to outliers), but more computationally intensive, since it requires sorting the samples in each region.

Putting everything together gives Algorithm 3, that creates regions one by one and optimizes them until convergence is reached. The parameter ε is the user-defined maximum approximation error. At each iteration, a new region is created around the pixel with maximum approximation error. Then, *n_inner_iter* iterations (5 in our experiments) of our generalized Lloyd relaxation are applied.

Adapting the algorithm to color images requires small modifications: first, each channel (L,u,v) has its own associated

Algorithm 3 : ardeco()

while $\sum_i \int_{R_i} \|(f-f_i)(x,y)\|^2 dxdy > \varepsilon$
  create new region from worst pixel
  for k **in** $1 \ldots n\_inner\_iter$
    flood_fill()
    update_seeds_and_fit()
  end for
end while

function basis $(\phi)^L$, $(\phi)^u$, $(\phi)^v$. Then, all operations are separated for each component, except in the region-growing algorithm, in which the *cost* variable is obtained by summing the contributions of L, u,v.

Figure 4 shows the result of our algorithm. The main limitation of this method is the speed of convergence. For a 1024×1024 image, the algorithm did not converge after 30 minutes. As done in [RM03], to speed-up the algorithm, we introduce a higher-level data structure. We will construct a triangulation of the image, well adapted to the discontinuities. The facets of this triangulation define groups of pixels, that will be called *trixels* in what follows. Section 3.2 explains how to construct this triangulation, and Section 3.3 shows how to use it to define the two-level Ardeco algorithm.

3.2. Constructing the Trixels

To accelerate the algorithm, we introduce an intermediate *trixel* data structure, that partitions the image into a set of triangles. In the subsequent variational segmentation step, each trixel will become indissociable: the final regions will be defined as groups of trixels. For this reason, the trixels need to be smaller in zones rich in details, and their edges need to be aligned with the discontinuities of the image.

The super-pixel approach developed in [RM03] also uses an intermediate data structure. The main difference is that we use a CVT (centroidal Voronoi tessellation) steered by an image saliency map instead of the graph-cut algorithm they use. This has the following two consequences: first, as shown further, since the trixels are triangles, some integral computations required by our algorithm can be made simpler by using quadratures. Second, corners of the original images can be easily constrained to appear as vertices in the CVT, which ensures they will be preserved in the final segmentation. Centroidal Voronoi Tesselations were applied in [Sec02] to image stipplling. Our algorithm is subpixel exact and more efficient with several orders of magnitude (our algorithm computes 20000 exact weighted barycenters per second, therefore takes 15 seconds to stipple an image, whereas [Sec02] takes 20 minutes).

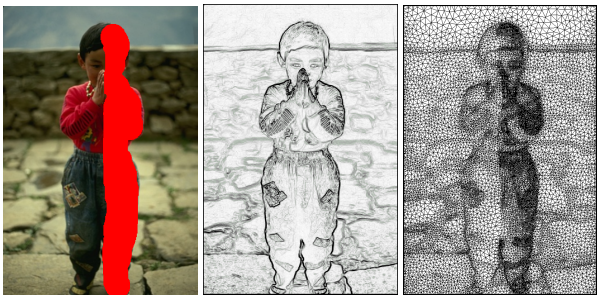


Figure 5: *Constructing the intermediate "trixel" data structure. User-defined importance map (right part of the character), saliency map and resulting trixels, adapted to both maps.*

We first compute an image saliency map $s(x,y)$. We experimented different methods, including Canny edge detectors and curvature approximants. Finally, the simple "compass" filter with Sobel's weight [RT01] used in [MZD05] gives satisfying results and is trivial to implement. The result of a corner detector is then inserted in this map, by giving a high importance ($s(x,y) = 10000$) to the pixels x,y tagged as corners. In our specific case, false positives are not a real problem (they will only artificially densify the CVT where they appear). For this reason, the SUSAN algorithm [SB97] with a low acceptance threshold ($threshold = 20$) gives satisfying results. Note that since the saliency map is integrated over larger regions in subsequent steps of the algorithms, we do not need using saliency estimators based on global quantities or large neighborhoods.

Then, we compute a CVT steered by the saliency map $s(x,y)$, using Lloyd's relaxation. Given a set of vertices (v_i) (initially located at random locations), Lloyd's relaxation optimizes the compactness of the cells, by iteratively relocating each vertex to the barycenter of its Voronoi cell (Algorithm 4).

Algorithm 4 : Lloyd_relax()

for each vertex v_i
  $v_i^{(k+1)} \leftarrow \int_{D_i} s(x,y) \begin{pmatrix} x \\ y \end{pmatrix} dxdy \,/\, \int_{D_i} s(x,y) dxdy$
end for
update Delaunay triangulation

In this algorithm, $v_i^{(k+1)}$ denotes the location of vertex i at the next iteration, and D_i denotes the Voronoi cell of the vertex v_i. The uniform barycenters of the original Lloyd relaxation are replaced with barycenters weighted by the saliency map $s(x,y)$. To compute the Delaunay triangulation, we use the CGAL library [FGK*00]. The main remaining difficulty is to evaluate the integrals of the saliency map $s(x,y)$ over the Voronoi cells D_i. To deal with this issue, we have designed a sub-pixel accurate generic rasterizer (Appendix A).

At this point, it is possible to simply generate *max_nb_vertices* vertices at random locations then use Lloyd's relaxation to optimize them. However, it is more efficient to process in a multi-resolution manner, by iteratively splitting the triangles sorted by their integrated saliency (see Algorithm 5). In this algorithm, $s(T_i) = \int_{T_i} s(x,y) dxdy$ denotes the integrated saliency in triangle T_i. In our experi-

ments, we used $max\_nb\_trixels = 30000$, $nb\_Lloyd = 10$ and $refine\_factor = 10$.

Algorithm 5 : generate_trixels()

```
insert 1000 vertices at random locations
while nb_trixels < max_nb_trixels
  sort all the trixels T by decreasing s(T)
  insert a vertex in the first refine_factor % triangles
  for k in 1.. nb_Lloyd
    Lloyd_relax()
  end for
end while
```

To be able to capture details in user-selected zones, it is easy to inject a user-defined importance map $I(x,y)$ in the algorithm, by simply multiplying the saliency map $s(x,y)$ with $I(x,y)$ in all computations. The result is shown in Figure 5. The right half of the character has 5 times as importance as the rest of the image.

3.3. Two-level Ardeco

Once the trixels are constructed, we can now proceed to re-express the pixel-based algorithm explained in Section 3.1 in terms of trixels. The main idea is based on the observation that the matrix A_i and right-hand side b_i used to estimate the parameters $(\alpha_{i,j})$ of the gradient attached to region R_i can be expressed as the sum of matrices A_T and right hand sides b_T attached to the trixels T included in region R_i. As a consequence, the A_T's and b_T's can be computed *before* entering the main loop of the Ardeco algorithm, as shown in Algorithm 6.

Algorithm 6 : pre_integrate()

```
for each trixel T
  A_T ← ∫_T Φ(x,y)Φ(x,y)^t
  b_T ← ∫_T Φ(x,y)
  for k in 1...nb_iter_M_estimator
    solve A_T (α_{T,1} ... α_{T,m})^t = b_T
    A_T ← ∫_T w(r_T(x,y)) Φ(x,y)Φ^t(x,y) dxdy
    b_T ← ∫_T w(r_T(x,y)) g(x,y)Φ(x,y) dxdy
  end for
  I_T ← ∫_T I(x,y) dxdy
end for
```

This results in a significant speedup, since no pixel access nor rasterization is required anymore in Ardeco's main loop. In the algorithm, r_T denotes the residual function in triangle T defined by $r_T = g - f_T = g - \sum_j \alpha_{T,j}\phi_j$. In the case an importance map $I(x,y)$ is defined by the user, the algorithm also computes the importance of the trixels I_T by integrating the importance map.

The matrix A_T and right-hand side b_T are simply the coefficients of the linear systems at the last iteration of the reweighted quadratic optimization process (see Equation 3, previous section). As can be seen (Figure 6), trixels carry more information than just the set of pixels intersected by a given triangle: for a given trixel, not only f_T gives a good approximation of color variation in trixel T, but also A_T

Figure 6: *Trixels with the attached pre-integrated higher-order representation of degree 0, 1 and 2 (2000 trixels).*

and b_T capture the higher-order information related with T. Note that this also includes the information of which pixels are outliers (through the weighting function w of the M-estimator).

Once the trixels T and associated pre-accumulated higher-order information A_T, b_T is computed, it is possible to recast the original Ardeco method (Algorithms 1 and 2) in terms of trixels (see Algorithm 7).

In this algorithm, each region R_i has a "seed" trixel T_i (the discrete version of the sites of a Voronoi tessellation). The point c_i is the center of gravity of T_i. The two integrals over T', involved in the computation of the *cost* term, are evaluated using the classical quadrature, given here for a function $D(x,y)$:

$$\int_T D(x,y)dxdy \simeq \frac{|T|}{6}\left(d_1^2 + d_2^2 + d_3^2 + d_1d_2 + d_2d_3 + d_3d_1\right)$$

where d_1, d_2, d_3 denote the value of D at the vertices of T.

Figure 7: *Left: Result of the two-level algorithm, obtained in 24 s. Right: The pixel-based version took 187 s on the same image.*

Algorithm 7 : ardeco_trixels_iteration()

```
//flood_fill
for i from 1 to n   { push(S, T_i, i, 0);   b_i ← 0;   A_i ← 0 }   end for
while not S is empty
  (T, i, cost) ← pop(S)
  if region(T) = 0
    A_i ← A_i + A_T;   b_i ← b_i + b_T;   region(T) ← i
    for each T' adjacent to T
      if region(T') = 0
        cost ← λ ∫_{T'} ‖ (x, y) − c_i ‖^2 dxdy + (1 − λ) ∫_{T'} ρ(r_i(x,y)) dxdy
        push(S, T', cost, i)
      end if
    end for
  end if
end while
//update sites and fit
for i from 1 to n
  solve A_i(α_{i,1} ... α_{i,m})^t = b_i

  T_i ← argmin_{T ⊂ R_i} (∫_T ρ(r_i(x,y)) dxdy)
end for
```

Note that the algorithm is not exactly equivalent to the pixel-based version, since the pre-accumulated weighting terms $w(r_i(x,y))$ characterizing the outliers are computed with respect to the trixel's approximant instead of the region's approximant. This does not make a difference in practice, since pixels that are outliers with respect to a specific region are likely to be outliers with respect to the trixel they belong to. Figure 7 shows the segmentations obtained with both versions of the algorithm. Similar segmentations are obtained, with a significant speed-up for the trixel version.

Comparision with segmentation algorithms: As explained in the introduction (see Figure 1, first page), in addition to detecting the boundaries, our algorithm needs also to approximate complex color variations by several gradients. As a consequence, our algorithm does not really belong to the category of segmentation methods. We cannot apply the classical segmentation benchmarks to our algorithm, since the multiple gradients would be interpreted as *oversegmentation* by these benchmark. However, we give some comparisons with the mean-shift algorithm, used in image and video stylization. Figure 8 compares the result of our algorithm to the original mean-shift algorithm [CM99] and to the anisotropic generalization [WTXC04], using a fixed number of regions (287 in this example), the images and results are from the cited references. Despite the obvious improvements of the anisotropic generalization, mean-shift still generates a large number of small regions in zones of sharp lighting variations. In contrast, since it uses higher-order gradients and does not require any bandwidth parameter, our method better balances the regions. As a consequence, a significantly smaller number of regions can be used.

Figure 9 shows the main failure mode of our algorithm, caused by certain textures that have intermediate scale and high contrast. Ardeco fails considering all these details as a whole and may generate a large number of regions (one per spot on the shirt), before segmenting the more important features of the image (eyes and mouth).

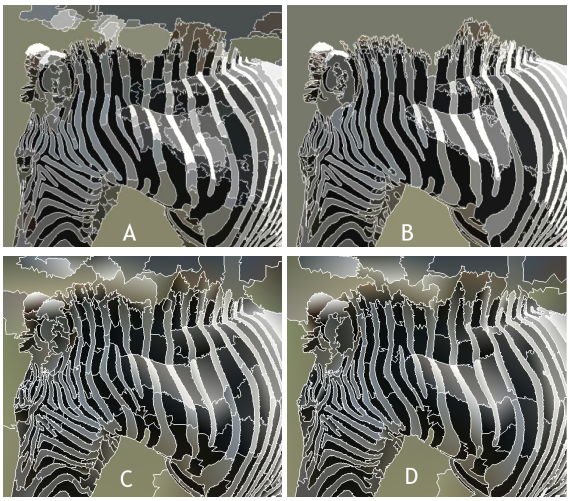


Figure 8: *Compared results of mean-shift (A), anisotropic mean-shift (B) (results from [WTXC04]) and Ardeco (C), using the same number of regions (287) and $\lambda = 0$. As can be seen, Ardeco creates well-balanced regions. As a consequence, it is possible to use a much smaller number of region (160), while keeping significant details (D).*

Now that we have presented the two-level Ardeco algorithm, we proceed to show how to export its result to standard vector graphics file formats. The output of Ardeco is very near to the representation used by standard vector file formats, but still requires some post-processing to be fully compatible with them.

4. Post-Processing

The first problem concerns the gradients used in vector file formats, that are not as general as those generated by Ardeco. The other problem concerns the conversion of region boundaries into splines, supported by most standard file formats. Using splines results in smoother boundaries and more compact files.

Figure 9: *Ardeco may fall directly in the traps of Berkeley's segmentation data set. A large number of regions may be generated.*

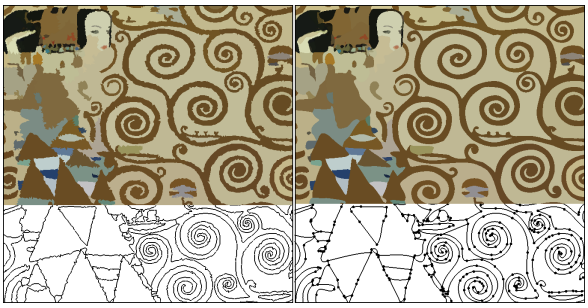


Figure 10: *Converting polygonal regions into Splines.*

4.1. Converting Gradients

The linear gradients computed by Ardeco are represented by three independent linear equations (i.e. 9 coefficients). Standard vector file formats are more restricted, since they use the same gradient direction for r, g and b. To enforce this condition, we find the "average gradient direction", by computing the eigen vector associated with the smallest eigen value of the 2×2 matrix $V_r V_r^t + V_g V_g^t + V_b V_b^t$ where V_r denotes the gradient vector associated with r (resp. g, b).

Quadratic gradients are more difficult to convert to standard vector file formats. In those file formats, higher-order gradients are limited to elliptic gradients (expressed by a linearly transformed circular gradient). Depending on the eigen values δ_1, δ_2 of the quadratic form computed by Ardeco, we distinguish the following cases:

- $\delta_1 > 0, \delta_2 > 0$: we output a transformed circular gradient. The rows of the transform (i.e. the axes of the ellipse) are given by the eigen vectors;
- $\delta_1 \times \delta_2 = 0$: this corresponds to a "cylindrical" gradient, that we approximate by two linear gradients;
- $\delta_1 < 0$ or $\delta_2 < 0$: this corresponds to a "hyperbolic" gradient. We approximate it by cutting the concerned region into four parts, in the direction of the eigen-vectors.

4.2. Converting Region Boundaries

Converting region boundaries into splines can be easily achieved by standard methods, since the vertices are ordered along the boundaries. We first determine the vertices that correspond to corners. A corner is either a vertex shared by more than two region or a vertex for which the corresponding pixel in the image was tagged as a corner (by SUSAN). A cubic spline segment is then attached to each polygonal line connecting two corners. Then, we apply to each segment regularized spline fitting and recursive subdivision (see e.g. [Chu80]) until the approximation error drops below a user-defined threshold (corresponding to 1 pixel in our case).

Figure 10 shows the algorithm applied to polygonal curves (left). The number of vertices in the so-constructed spline (right) is dramatically reduced (1105 control points instead of 9286 vertices in this example).

5. Results

The timings of the algorithm are given in the table below. (Lena, 30k trixels, $\lambda = 0.001$, $\varepsilon = 20$). As can be seen, using our two-level algorithm, image resolution has a small influence.

nb regions	degree 0 161	degree 1 138	degree 2 124	degree 3 106
512x512	72 s.	83 s.	105 s.	125 s.
2048x2048	105 s.	120 s.	142 s.	172 s.

The algorithm applied to various data sets is demonstrated in Figure 11.

The first row shows that our method successfully reproduces a painting already in the "art deco" style. It recognizes the color gradients and simple geometric shapes that compose the image. Increasing the cell compactness λ creates a mosaic-like image.

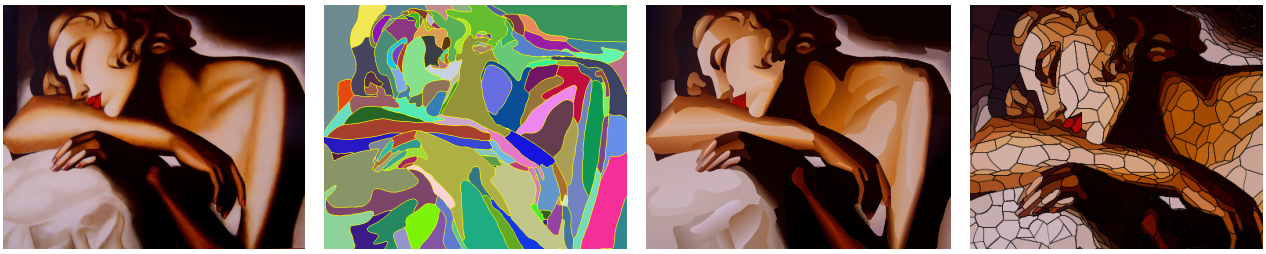
Second row: Stylizing "Lena", with constant and linear gradients. Importance was painted on the face.

Third row: Using importance acquired from an eye-tracker (data from [DS02], available on the web). Importance varies from 1 to 30, and is proportional to both the distance to fixations and fixation time. In the result (middle image), the variations in object detail convey the notion of depth-of-field. However, vector images with constant colors still have a "flat" looking. The perceptual depth-of-field effect is improved (right image) by using Ardeco's higher-order gradients and a Gaussian filter (supported in SVG and Flash8) applied to the primitives in the background (i.e, primitives with no importance).

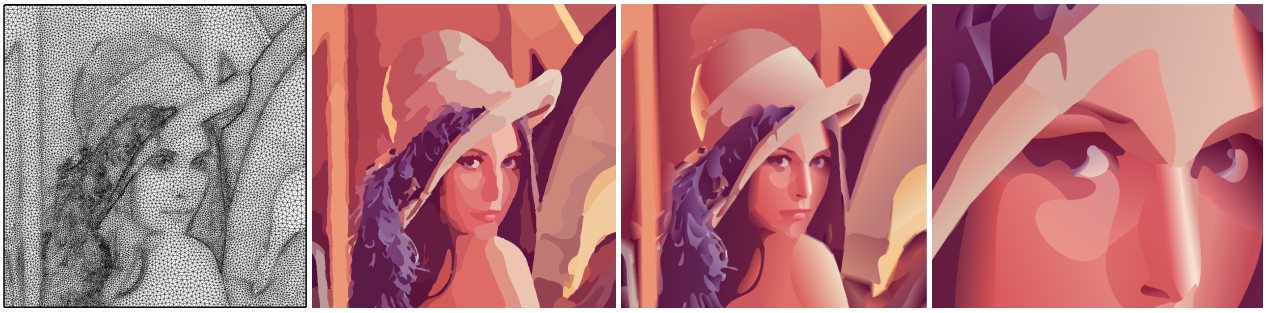
Last row: User-defined importance can fix the segmentation when the algorithm misses some features.

Conclusion

In this paper, we have presented a new automatic segmentation method that produces vector images from bitmaps. Overall, we have proposed an efficient two-level numerical scheme to minimize Mumford and Shah's energy functional and that recognizes regions with higher-order gradients. These gradients make it possible to create vector images that convey a better sensation of depth and an overall lighting atmosphere.



An “art deco” painting by Tamara de Lempicka is reproduced in vector form by our method, using only 180 primitives and gradients. Rightmost image: increasing the cell-compactness term ($\lambda = 0.75$) and using constant colors generates a mosaic-like image.



Trixels, constant colors and higher-order gradients. See how the classical “Lena” image has been stylized by the method (242 primitives).

Importance computed from De Carlo et. al’s data and adapted triangulation; constant colors; higher-order gradients with SVG Gaussian filter. Note how the higher-order gradients combined with the blurred background improve the perception of depth.

Automatic result, and result obtained with an importance map (in red). Black strokes are superimposed as in De Carlo’s paper. The strokes are extracted from the constant-color segmentation and superimposed to the higher-order gradients.

Figure 11: *Stylizing and vectorizing various images. In all these examples, except in the upper-right image, we used* $\lambda = 0.01$ *(boundary length minimization) and* $\varepsilon = 20$ *(maximum approximation error). Note how higher-order gradients reproduce the lighting, conveying the same atmosphere as the original bitmap images. The vector files are available in the supplemental material (in SVG and PDF file formats).*

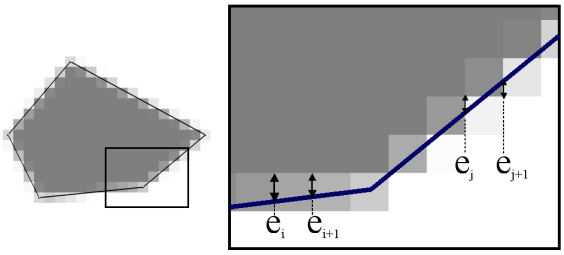


Figure 12: *Using Bresenham's algorithm to compute exact integrals over images.*

Appendix A: A Generic Rasterizer

In Ardeco, several steps of the algorithm require computing integrals over polygonal subsets of images:

- ◇ weighted barycenters: $\int_{D_i} s(x,y) \begin{pmatrix} x \\ y \end{pmatrix} dxdy / \int_{D_i} s(x,y)dxdy$
- ◇ pre-accumulated covariance matrices: $A_T = \int_T \Phi^t \Phi dxdy$
- ◇ trixels importance: $I(T) = \int_T I(x,y)dxdy$

This process may be thought of in term of rasterization, then implementing it on the GPU is an idea that comes immediately to mind. However, the GPU programming model does not allow individual pixels to communicate, which prevents it from computing integrals. For this reason, we used a software rasterizer instead. To efficiently find all the pixels inside a convex polygon, we use the classic algorithm, that first rasterizes the contour of the polygon, then processes each individual horizontal scan-line (see Figure 12). We use a variation of the highly efficient implementation available on the web [Lev95], developed in the "pre-GPU-hic" ages. However, in our case, special care needs to be taken when computing integrals over an image. Since trixel edges may be near discontinuities, it is especially important to compute the exact filling ratio for the concerned pixels in order to avoid artificially increasing their importance. A slight variation of the classic antialiased polygons algorithm can give the exact filling ratio. The basic idea is to use the displacements e_i and e_{i+1} computed by Bresenham's algorithm (see Figure 12). In the two different cases of Bresenham's algorithm, the filling ratio is then given by $0.5(e_i + e_{i+1})$ if $e_{i+1} < 0$ or $0.5(e_j * e_j)/(e_j + e_{j+1})$ if $e_{j+1} > 0$. To compute the filling ratio at the corners of the polygon, we directly compute the intersection between the pixel and the polygon, using Sutherland and Hogdman's reentrant clipping algorithm. The resulting algorithm computes the exact filling ratio for all pixels, and is reasonably fast (20000 polygons per second on a 2 Ghz Pentium M), which is more efficient by several orders of magnitude as compared to the method described in [Sec02]. In terms of implementation, the functionality is exposed through a generic class (see Algorithm 8).

Algorithm 8 Generic rasterizer

```
template <class Process> class GenericRasterizer {
public:
        void beginPolygon() ;
        void addVertex(int x, int y) ;
        void endPolygon() ;
        PROCESS process ;
} ;

class IntegralProcess {
public:
        void addPixel(int x, int y, double coverage) {
            value += image->getPixel(x,y) * coverage ;
        }
        double value ;
        Image* image ;
} ;
```

The `addPixel()` function is called for each pixel or pixel fragment of the polygon. Various `Process` classes can be used, to compute integrals, means or medians, or to integrate covariance matrices. This is extensively used in our implementation. The complete C++ code of the generic rasterizer is available in the supplemental material.

References

[BBNG05] BATTIATO S., BLASI G. D., NICOTRA S., GALLO G.: Svg rendering for internet imaging. In *IEEE CAMP conf. proc* (2005).

[Chu80] CHUNG W.-L.: Automatic curve fitting using an adaptive local algorithm. *ACM Trans. Math. Softw. 6*, 1 (1980).

[CM99] COMANICIU D., MEER P.: Mean shift analysis and applications. In *IEEE ICCV conf. proc.* (1999).

[CRH05] COLLOMOSSE J. P., ROWNTREE D., HALL P. M.: Stroke surfaces: Temporally coherent artistic animations from video. *IEEE Trans. Vis. Comput. Graph. 11*, 5 (2005), 540–549.

[CSAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M.: Variational shape approximation. *ACM Transactions on Graphics (SIGGRAPH conf. proc.)* (2004).

[DS02] DECARLO D., SANTELLA A.: Stylization and abstraction of photographs. In *SIGGRAPH conf. proc.* (2002).

[FGK*00] FABRI A., GIEZEMAN G.-J., KETTNER L., SCHIRRA S., SCHÖNHERR S.: On the Design of CGAL, a Computational Geometry Algorithms Library. *Softw. – Pract. Exp. 30*, 11 (2000), 1167–1202. www.cgal.org.

[GCS02] GOOCH B., COOMBE G., SHIRLEY P.: Artistic vision: painterly rendering using computer vision techniques. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering* (New York, NY, USA, 2002), ACM Press, pp. 83–ff.

[GG01] GOOCH B., GOOCH A.: *Non-Photorealistic Rendering*. A. K. Peters, 2001.

[Hae90] HAEBERLI P.: Paint by numbers: Abstract image representations. In *SIGGRAPH conf. proc.* (1990).

[HEH05] HOIEM D., EFROS A. A., HEBERT M.: Automatic photo pop-up. *ACM Trans. Graph. 24*, 3 (2005), 577–584.

[Her01] HERTZMANN A.: Paint by relaxation. In *Computer Graphics Intl. conf. proc.* (2001), pp. 47–54.

[Hub81] HUBER P.: *Robust Statistics*. Wiley, New York, 1981.

[KSE*05] KWATRA V., SCHODL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics (SIGGRAPH conf. proc.)* (2005).

[KWT88] KASS, WITKIN, TERZOPOULOS: Snakes: Active contour models. *Int. J. of Comp. Vision* (1988).

[Lev95] LEVY B.: Tagl: an efficient software rasterizer, 1995. www.loria.fr/~levy/software.

[Llo57] LLOYD S.: *Least squares quantization in PCM*. Tech. rep., Bell Laboratories Technical Note, 1957.

[MFTM01] MARTIN D., FOWLKES C., TAL D., MALIK J.: A database of human segmented natural images. In *ICCV conf. proc.* (2001).

[MS85] MUMFORD D., SHAH J.: Boundary detection by minimizing functionals, I. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition* (1985), pp. 22–26.

[MZD05] MATUSIK W., ZWICKER M., DURAND F.: Texture design using a simplicial complex of morphable textures. *ACM Transactions on Graphics (SIGGRAPH conf. proc.)* (2005).

[OF01] OSHER S., FEDKIW R.: Level set methods: An overview and some recent results. *UCLA IPAM reports* (2001).

[RM03] REN X., MALIK J.: Learning a classification model for segmentation. In *Proc. 9th Int'l. Conf. Computer Vision* (2003), vol. 1, pp. 10–17.

[RT01] RUZON M., TOMASI C.: Edge, junction and corner detection using color distribution. *IEEE Trans. on Pattern Analysis and Machine Intelligence* (2001).

[SB97] SMITH S. M., BRADY J. M.: Susan, a new approach to low level image processing. *Int. J. Comput. Vision 23*, 1 (1997), 45–78.

[Sec02] SECORD A.: Weighted voronoi stippling, 2002.

[WTXC04] WANG J., THIESSON B., XU Y., COHEN M.: Image and video segmentation by anisotropic kernel mean shift. In *ECCV (2)* (2004), pp. 238–249.

[WXSC04] WANG J., XU Y., SHUM H.-Y., COHEN M. F.: Video tooning. *ACM Transactions on Graphics (SIGGRAPH conf. proc.)* (2004).

[Zha96] ZHANG Z.: Tutorial on parameter estimation techniques, 1996. www-sop.inria.fr/robotvis/personnel/zzhang/Publis/Tutorial-Estim/Main.html.