2006-05-31

Image Vectorization

Brian L. Price
Brigham Young University - Provo

IMAGE VECTORIZATION

by

Brian Price

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Brian Price

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	William Barrett, Chair
Date	Dan Olsen
Date	Daniel Zappala

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Brian Price in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

William Barrett
Chair, Graduate Committee

Accepted for the Department

Parris Egbert
Graduate Coordinator

Accepted for the College

Tom Sederberg,
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

IMAGE VECTORIZATION

Brian Price

Department of Computer Science

Master of Science

We present a new technique for creating an editable vector graphic from an object in a raster image. Object selection is performed interactively in subsecond time by calling graph cut with each mouse movement. A renderable mesh is then computed automatically for the selected object and each of its subobjects by (1) generating a coarse object mesh; (2) performing recursive graph cut segmentation and hierarchical ordering of subobjects; (3) applying error-driven mesh refinement to each (sub)object. The result is a fully layered object hierarchy that facilitates object-level editing without leaving holes. Object-based vectorization compares favorably with current approaches in the representation and rendering quality. Object-based vectorization and complex editing tasks are performed in a few 10s of seconds.

ACKNOWLEDGMENTS

I would like to thank my professor Dr. Barrett for all the time, support, and direction that he has given me throughout my undergraduate and graduate career. I thank Adobe Inc. for the feedback on our project and funding that helped make it possible. I would also like to thank my family, friends, and roommates for their support.

Contents

List of Figures

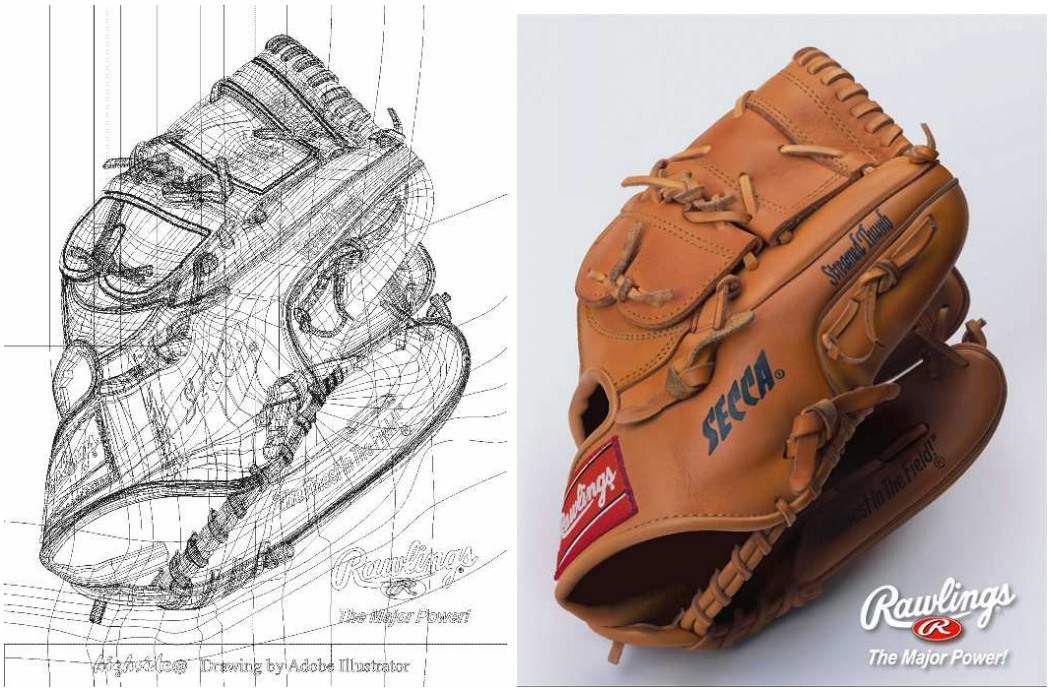
Chapter 1

Introduction

Image vectorization is the process of converting a raster image into a vector graphic. Raster images, such as photographs, are understood by a computer only as an array of pixel values, limiting the operations that can be performed on the image to those which manipulate pixels. Vector graphics, on the other hand, have a geometric data structure associated with them, allowing more direct object-level control. Unfortunately, vector graphics are more difficult to produce than photographs, usually requiring a graphic artist to laboriously hand craft them. Through image vectorization, we can quickly convert raster images into vector graphics with minimal user intervention. This requires the detection of specific objects in the image, followed by a means of fitting a suitable vector structure to them.

There are several motivating reasons for performing image vectorization. Vector graphics are used abundantly in advertising and on the web, mainly for their scalability, stylization, and editability. Producing these manually is an increasingly expen-

sive, time-intensive, high-level task, especially as the demand for increased realism increases. Image vectorization would facilitate the creation of editable (photorealistic) graphical models of complex, real-world objects. For example, the picture of the baseball glove created by the artist Highside [32] in Figure 1.1(b) is not a photograph, but was generated from the vector graphic shown in Figure 1.1(a). Creation of this model required approximately 60 hours by this highly skilled artist. Semi-automated vectorization techniques can significantly reduce the time and talent required for graphic production. Using image vectorization, libraries of such models could be created more readily, extending the tool set for artists in doing image synthesis and scene compositing, while reducing the time and tedium associated with manual model creation.



(a) (b)

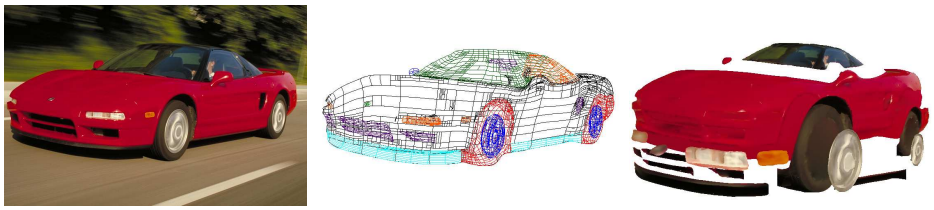
Figure 1.1: (a, b) Mesh created and rendered by Highside [32]. It required 60 hours to create the mesh.

Vector representation and manipulation of objects contained in raster images would also ease many of the tasks associated with image editing. For example, image objects could be scaled or distorted without the pixelation that accompanies traditional interpolation techniques (see Figure 3.20), and without affecting surrounding image areas as occurs with rubber-sheet distortions. Vectorization would allow objects to be reduced in size or deleted without leaving holes in the surrounding objects. A separate vector graphic for each object in an image would also allow it to be relit, filtered or stylized independent of the other objects in the scene, with user-selectable levels of detail. And clearly, any manipulation of image content performed at the object level offers greater efficiency, selectivity and user-friendliness than low-level, pixel based tools.

Techniques for image vectorization could also be used to reverse-engineer a graphic and repurpose existing animation. Such techniques could also provide a medium for the exchange of objects between the world of graphical models and raster images. Vector representation also provides a means of representing image content geometrically (and hierarchically), in a way that describes object adjacency and enables image or object search technologies. Finally, a scalable, hierarchical, vectorized representation may allow a more efficient representation of a raster image and open the door to progressive compression or transmission strategies.

Automatically-generated vector graphics should possess several desirable characteristics (see Figure 1.2). Individual graphic objects should correspond to significant objects in the image. Subobjects of these significant objects should also exist as separate, nested graphic objects. Graphic objects should be easy to edit, with edits on objects being automatically propagated to their subobjects (if desired). Translation

or deletion of subobjects should not leave holes in their parent objects. Vector graphics should also accurately represent the original object at a level of detail, ranging from near-photorealistic to stylized, as specified by the user. Artificial edges should not be introduced in smoothly shaded areas, while sharp edges should remain sharp. These characteristics are usually found in manually-created graphics, but are poorly enforced in current techniques for automatically generating graphics (see Section 2.1).



(a) (b) (c)

Figure 1.2: (a) Original raster image. (b) Vector graphic consisting of 3579 Bezier patches computed from image in (a) in about 1 minute. Meshes for individual objects (e.g. car body, wheels, hubcaps, lights, etc.) are created automatically using recursive graph cut and error-driven grid refinement. (c) OpenGL rendering of the mesh shown in (b). Individual objects are repositioned and scaled to illustrate interactive editability. Note that where objects have been displaced, the underlying objects (car body, tires) have been filled in automatically (no holes) with relevant impressions and indentations preserved.

Constructing a vector graphic in accordance to these characteristics is a difficult challenge. Object-based vectorization requires object segmentation, necessitating powerful user-driven segmentation tools. However, any given image may contain hundreds, to possibly thousands of objects and subobjects, far too many for a user to segment in any practical time frame. General purpose image segmentation is, as yet, an unsolved problem. Recent work has introduced powerful, semi-automated segmentation tools, such as Graph Cut [13], Snakes [31], and Intelligent Scissors [37], to cite only a few. However, even these tools are unacceptable for full image vectorization,

where so many objects must be segmented. Methods for automatic segmentation must be developed.

After the regions are selected for object vectorization, the next challenge is to create a suitable vector graphic. Current vectorization techniques do not support shaded regions, but rather segment the image into flat-filled regions. Accordingly, the entire process of creating a vector graphic must be discovered. Hierarchical ordering must be implemented, and a method of editing an object hierarchy efficiently must be developed. Since object deletions are allowed, occluded portions of an object must be filled, calling upon texture synthesis and image completion. Each of these difficulties must be addressed in order for vectorization to be successful.

In this paper, we present a method for image vectorization conforming to these desired chararctistics. Users may select objects to vectorize by means of an interactive min graph cut selection tool, similar to [35] but with improvements. A vector graphic is then created based on Bezier patches, allowing for smooth shading. Automatic segmentation and hole-filling are provided. Editing tools similar to those in Object-Based Image Editing [8] are used to demonstrate object-level editability.

This work makes several important contributions to the current body of research. We provide a new means for image vectorization that creates graphics that are more realistic, naturally shaded, and far more editable than current techniques, and that contain hierarchical levels of detail. We present a more advanced object-based representation of images through a hierarchy structure, allowing for better organization and easier editing and manipulation of objects in an image. We introduce a new method of automatic image segmentation. We also introduce several improvements to min graph cut image segmentation which allow for easier and faster use.

We validate the success of the project by comparing our results qualitatively and quantitatively with other vectorization algorithms, demonstrating more natural shading and superior editability. We show the utility of the hierarchical object representation by performing complex image edits not easily achievable through other techniques. We also show results of progressive levels of detail, meaning the user may view the same graphic at different levels of stylization.

Chapter 2

Previous Work

Image Vectorization is most closely related to current commercial vectorization programs which also convert raster images into vector graphics. However, Image Vectorization moves beyond these current techniques by offering object-based vectorization, graphic editing tools, various stylization levels, and automatic hole filling. Image Vectorization therefore draws upon not only current vectorization techniques but also those found in image editing, texture synthesis, non-photorealistic rendering, and object selection.

2.1 Vectorization Techniques

Much of the previous work in image vectorization has focused on maps, engineering drawings, documents, or similar raster images that are inherently bitonal in nature [1] [48]. Because the images are bitonal, the problem reduces to curve-fitting boundaries. These techniques work quite well within their intended domain; however, this problem is drastically different than that of producing a vector representation of an object in a color image, which requires not only boundary information but also internal geometry and color information to provide appropriate coloration of object

interiors. Because of this difference, these previous techniques are not applicable to full color image vectorization.

A long-standing approach to full-color image vectorization is found in Adobe's Streamline [2]. Streamline converts raster images to line art, but is targeted primarily at scanned linework rather than photographic images. More recent vectorization tools such as Adobe Illustrator Live Trace [4], Corel's CorelTRACE [16], Siame Vector Eye [42], Macromedia Flash [36] or AutoTrace [7], can be applied directly to full color images, but most of these are still targeted at converting line drawings or flat-shaded color cartoons. When converting color images, they yield vector graphics consisting of many small homogenously-colored objects similar to those produced by posterization algorithms. Examples of the output of these techniques is shown in Figure 2.1.

In the introduction, we described several desirable characteristics of vector graphics as typically found in hand-crafted graphics. Such chararcteristics include accurate representation of the image, correspondence of graphical objects to image objects, and ease of editing. However, the graphics produced by current vectorization programs fall short in many of these areas. Current vectorization programs create an extremely large number of small, irregularly-shaped, flat-filled regions. These can represent scanned artwork with relatively flat shading quite well, but cannot accurately represent the multitude of photographic and artistic images with significant shading and color variation. Rather, they segment smooth-shaded regions into a series of flat color regions, adding artificial edges where the regions meet, as seen in Figure 2.1.

These methods also do not provide a mechanism for the vectorization and direct control of specific objects in an image. Objects are instead oversegmented into small

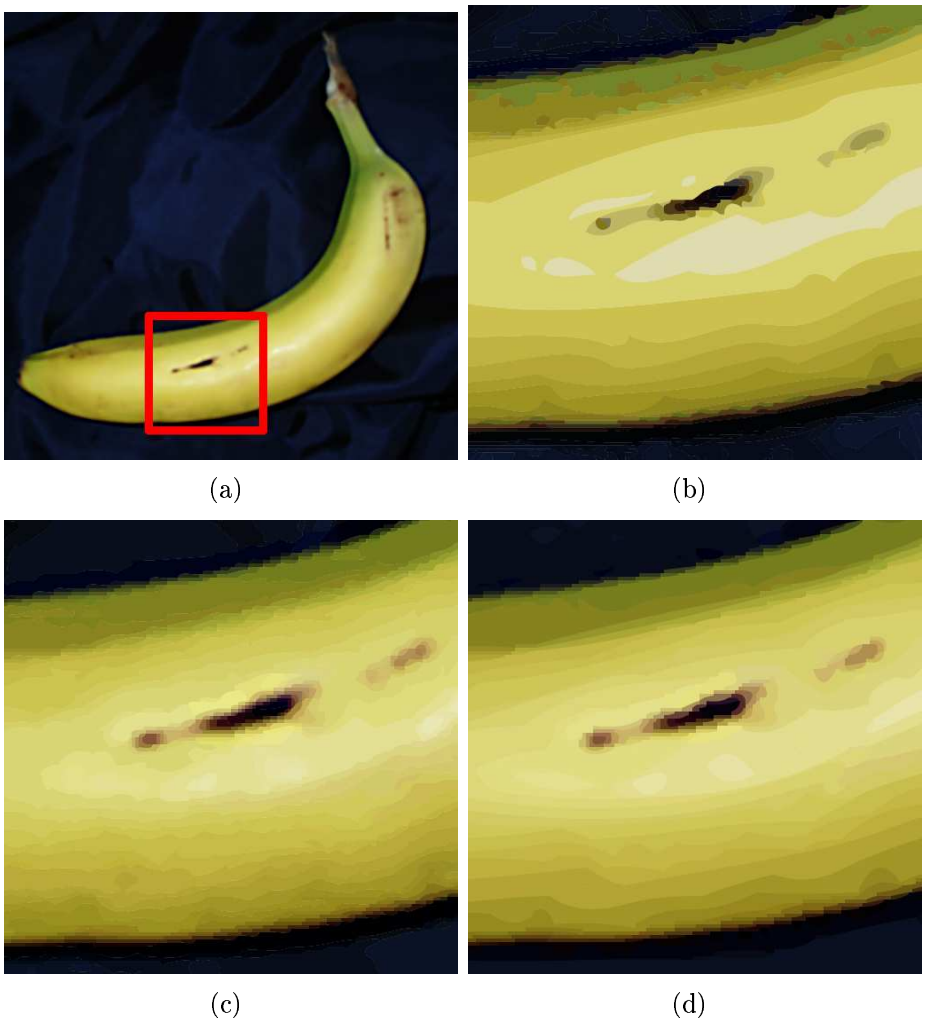


Figure 2.1: The square region of interest in the original image (a) is vectorized by (b) Adobe Illustrator, (c) Vector Eye, and (d) Macromedia Flash.

regions corresponding to similarly colored areas, which rarely correspond to significant objects in the image. Also, they often produce extremely complex boundaries comprised of a polygonal or spline boundary around the similarly-colored region, as shown in Figure 2.2. Since these regions have no connection to one another, have such complex boundaries, and are many in number, editing is difficult and tedious.

Real images are comprised of objects which may contain many subobjects and their corresponding subobjects. Representing objects in images as actual graphical objects allows for more natural and simplified editing of the objects. Hierarchical representation allows for the propagation of edits to subobjects.

(a) (b)

Figure 2.2: Regions from the vectorized banana shown in Figure 2.1 as produced by (a) Adobe Illustrator and (b) Vector Eye.

Without a sense of object hierarchy, it is difficult to perform background completion to prevent holes from appearing in the image during editing. Some of these systems will create a fill behind an object if it is completely enclosed within another object's boundary. However, this often does not happen, meaning that there is no fill behind most of the small regions in the image, which results in holes if a region is moved or deleted.

2.2 Pixel-Based Editing

Traditional image editing consists of pixel-based operations such as recoloring or moving pixels. Photoshop [3] and The Gimp [44], for example, contain many pixel editing tools, such as paintbrushes, blur and sharpening brushes, clone brushes, and

global filters. Photoshop's Healing Brush copies pixels from a sample location to a destination, much like a clone brush, but matches the texture, lighting, transparency, and shading of the destination while doing so. Oh et al. [39] developed a clone brush which prevents distortion caused by perspective. Rubber-sheet techniques [11] [46] allow for image warping or morphing between two images.

Pixel-based editing approaches are ideal for editing tasks that simply require recoloring or shifting pixels regardless of the their object membership. However, users perceive images as a collection of objects rather than an array of pixels, and many commonly desired editing tasks involve direct manipulation of these objects. Pixel-based techniques are generally inadequate for such operations. Users may sometimes achieve the desired results by selecting the object region and applying pixel-based techniques to that region only, but this still does not give true object level control. Often users are required to go to great lengths to produce the desired manipulation using pixel-based editing.

2.3 Object-Based Editing

A significant extension of Image Vectorization over other current vectorization techniques is its ability to vectorize on an object level. Various object-based methods of image editing exist, with Object-Based Image Editing (OBIE) [8] by William Barrett and Alan Cheney being the most relevant to this project. OBIE can be considered a precursor to this project, inspiring our object-based implementation. We also use OBIE-style widgets for interactive object-based editing operations.

Object-Based Image Editing allows image editing to occur on an object level via easy-to-use interactive tools. Users first select a desired object in the image, and then edit that object as a whole using various translation, rotation, scaling, and warping

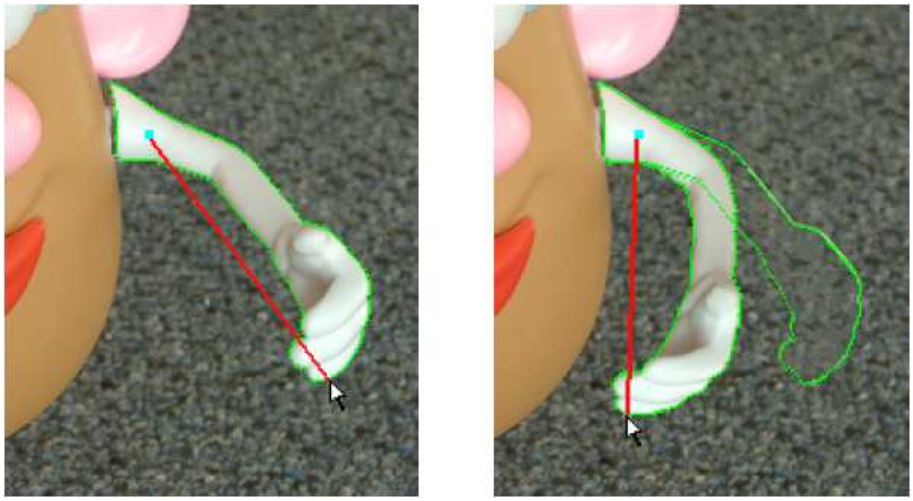


Figure 2.3: Example of image editing using Object Based Image Editing. Potato Head's arm is easily bent due to its object representation. Images from [8].

tools. OBIE utilizes a hierarchical watershed segmentation for object selection and hole filling. OBIE editing tools allow for non-linear transformations as well as indirect object manipulation. OBIE provides an excellent means of performing many image edits more quickly and naturally than pixel-based techniques. An example of an image edit is shown in Figure 2.3.

The most significant difference between OBIE and Image Vectorization is the intended final product of the two methods. For OBIE, the deliverables are only modified raster images. The structures being used to produce the final image were chosen because they were an efficient means of producing the intended result and not for their own intrinsic value. In Image Vectorization, however, the deliverables are vector graphics, and thus the structures used are inherently valuable. Their value increases with their ability to be edited and stylized easily.

The work presented in this paper extends the OBIE framework in several significant ways. The OBIE framework represents objects with an irregular, texture-mapped

triangular mesh, requiring storage of the original image pixel data and leaving objects susceptible to pixelation under certain scaling and editing operations. Our work represents objects using a scalable, regular mesh of Bezier patches, requiring storage of only the node color and mesh geometry. In OBIE, the mesh has one level of detail specified by the chosen level in the watershed hierarchy. In our work, the mesh is hierarchically ordered, supporting coarse-to-fine editing operations and selectable levels of detail in the object rendering. In OBIE, objects are selected by manually tagging watershed regions; here we exploit graph cut for object selection and automated, recursive segmentation and hierarchical ordering/layering of the object into relevant subobjects. This also allows selective, object-level filtering and stylization. Because of the texture layering used in OBIEs automated hole filling, it is best suited for relatively flat or random (stochastic) regions. Our work extends this by extrapolating smooth, shaded surfaces over holes for object background completion (Figure 1c).

Additional methods of object-based editing have been developed. The innovative work by Elder and Goldberg [23] demonstrates object-level editing, even deletion, but requires that the user perform contour grouping, which is analogous to object selection. W. Li et al. [34], cleverly apply semi-automated object selection and layering constraints to create exploded-view diagrams, analogous to what is shown in Figure 1.2(c).

2.4 Object Selection

A popular area of research in computer vision is object segmentation. Due to the difficulty in automatically segmenting images, and the impossibility of automatically knowing which of several objects in an image the user wants selected, most research

focuses on developing interactive segmentation tools to reduce user effort in object selection. Object segmentation tools are generally divided into boundary and region based algorithms.

Among the best known boundary segmentation algorithms is Intelligent Scissors [37], which computes a least cost path from the user's initial mouse click to the current mouse position in order to best localize the object boundary, as shown in Figure 2.4. The costs along the path are based on image gradient and Laplacian information. Snakes [31], another boundary based method, allows the user to again roughly mark the image boundary, then iteratively adapts the outline to match the object boundary by minimizing an energy function based on the snake boundary curvature and image gradient. Boundary based techniques work well in many cases. Unfortunately, some objects have complex boundaries that require excessive user interaction to correctly segment the object. Complex textures also complicate segmentation by obscuring true object boundaries with internal gradients.

Region-based segmentation algorithms rely on regional information provided by a user to select objects. Magic wand [3] selects all pixels connected to an initial user-specified example that fall within a similar color threshold. Intelligent Paint [40] segments the image into watershed regions, then groups regions connected to a user selected region based on statistical similarity. While these techniques take advantage of object regional information, they can easily fail to select objects with weak edges by "leaking" the selection to include image regions exterior to the object.

Recent break-through work in interactive image segmentation [5] [35] [41] has been accelerated by the popularization of minimum graph cut [13] [12]. Min graph cut techniques apply both region and boundary information in segmentation by combining

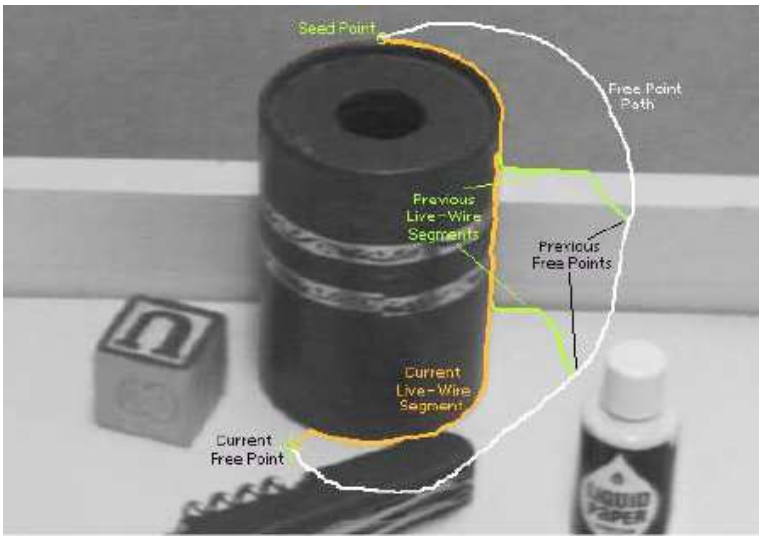


Figure 2.4: Intelligent Scissors selects objects by allowing the user to roughly trace the object boundary. The algorithm uses Laplacian and gradient information to adhere to the true object edge. Images from [37].

both into an energy minimization equation. Segmentation is cast as a graph cut problem, which tries to divide one connected graph into two connected graphs by cutting along the lowest weighted edges. Seeds are placed in the graph at nodes to assign those nodes to one of the two specified subgraphs, and they are used to help determine the membership of other nodes. For images, pixels correspond to nodes in the graph, and the boundary and region information determine the weights of the edges. Generally, users create foreground or background seeds by marking pixels with a paintbrush as foreground or background. These seeds help initialize the graph, and min graph cut consequently divides the graph (image) into two regions, foreground (desired object) and background. An example of this from Lazy Snapping [35] is shown in Figure 2.5. Grab Cut [41] alters this interface by allowing the user to simply draw a bounding box around the object, followed by iterative refinement to localize the correct object.

Figure 2.5: Lazy snapping selects objects by optimizing a min graph cut problem. Foreground seeds (yellow) and background seeds (blue) provide initialization information for the graph cut. The dotted line shows the segmentation. Images from [35].

Building on Lazy Snapping [35] this paper presents several extensions to user-driven graph cut segmentation. (1) Initial object selection is performed by calling the graph cut algorithm continuously, as each new foreground sample is collected with the mouse. This allows the object to grow as each sample is collected, making the user an active, rather than a passive participant, similar to Intelligent Paint [40]. Overgrowth is curtailed by sampling those areas back into the background, as in GrabCut [41]. (2) We apply graph cut to a toboggan-based watershed, allowing fast, pixel-level, boundary localization in one step, rather than two. This also allows graph cut to be called continuously without the need for user intervention. (3) We introduce automated, recursive graph cut for hierarchical segmentation and ordering of image objects.

Although a much more difficult problem, work has been done in automatically selecting objects from images. Gao et al. [25] segment by finding connected components after thresholding in morphological gradient space, then group all remaining pixels to a region using a partition optimization method. Jing et al. [30] define a homogeneity metric, select seed regions based on the most homogeneous regions, then grow and merge these regions to produce a final segmentation. Deng et al. [19] convert the image into a J-image, an image derived from class map information, to assist in segmentation. Fan et al. [24] find edges using an isotropic edge detector, then grow regions out from the centroids of the regions defined by the found edges. All of these techniques work well in some cases, but perform poorly in others. Automatic segmentation also has no means of guaranteeing the selection of the object that is of interest to the user.

2.5 Texture Synthesis

For image edits such as translations and deletions to be effective, there must be relevant imagery underneath the object being moved or deleted. Texture synthesis provides a means of filling in holes in images. Approaches to texture synthesis include pixel-based methods, patch-based method, and inpainting.

Pixel-based texture synthesis involves creating new texture one pixel at a time. Most pixel-based texture synthesis techniques today find root in Efros and Leung's landmark paper Texture Synthesis by Non-parametric Sampling [22]. In this paper, a given pixel is filled by locating another pixel of existing texture whose neighborhood is most similar to the neighborhood of the pixel to be filled. A texture generated by this algorithm is shown in Figure 2.6. Wei and Levoy [45] build on this by searching for a similar texture pyramidally through a multi-resolution tree to allow for faster

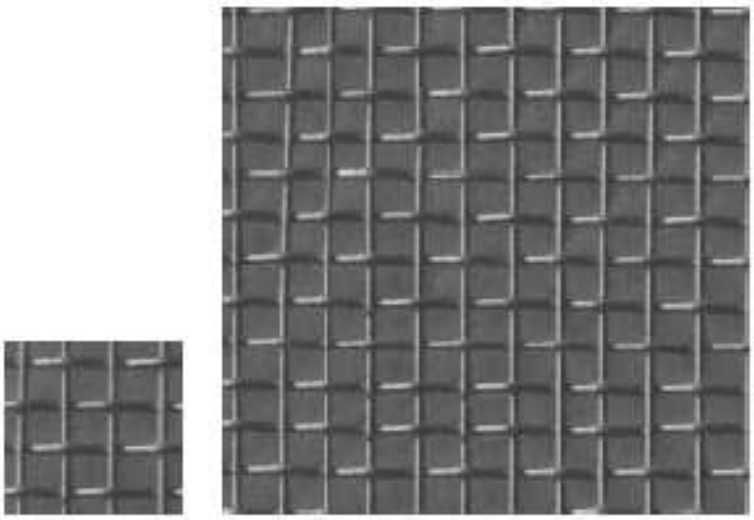


Figure 2.6: Example of texture synthesis from [22]. Images from [22].

texture synthesis. Ashikhman [6] performs synthesis in scanline order using an L-shaped window, and allows for user guidance in directing the synthesis. Drori et al. [20] iteratively fills the hole and computes a confidence measure over the newly-filled pixels to indicate the need for further iterations.

Patch-based texture synthesis, on the other hand, creates new textures by copying a patch of texture into the destination region instead of copying one pixel at a time, greatly speeding the process of synthesis but increasing the probability of repeated patterns. Efros and Freeman [21] lies down overlapping blocks of texture, then computes a least cost path across the overlap region to merge the two. Graphcut Textures [33] similarly lays down texture and finds a seam on overlapping regions, but it does not require the size of the patch to be chosen beforehand and uses min graph cut to find the seam.

Inpainting has produced impressive results for object background completion and hole filling [10], as shown in Figure 2.7. Inpainting fills holes in images from the

Figure 2.7: Example of inpainting. Images from [10].

outside in, while propagating edges into the synthesized region. Inpainting preserves structure well, but tends to oversmooth the region and loose texture. Recent, important extensions recover both structure and texture [9] [17].

2.6 Non-Photorealistic Rendering

Image Vectorization allows the creation of highly stylized versions of the original image, recalling work done in non-photorealistic rendering (NPR). Seeing how all current vectorization algorithms also stylize, they could be considered NPR algorithms too. However, traditionally NPR algorithms do not create vector graphics of the original image, but rather just paint new pixels.

Various types of non-photorealisic rendering techniques have been produced. Some techniques require user interaction to produce the stylized image. Paint by Numbers [26] is an early example of such a system. In this system, the user selects a brush type, then sweeps over the image while strokes are drawn using color samples from the current mouse position. DeCarlo [18] uses eye movement as its form of input. The user looks at the image, and areas where the eye lingers are considered more important and therefore are rendered with greater detail. An example is of this

Figure 2.8: Example of image stylization from [18]. The important features from the original image are located by tracking where the user's eye looks (circles in middle image) to produce the stylized image on the right. Images from [18].

Figure 2.9: Example of image stylization from Image Analogies [29]. The transformation that changed A to A' is applied to B to produce the image B' as output. Images from [29].

technique is shown in Figure 2.8.

Different automatic non-photorealistic rendering algorithms have also been developed. Several different methods have been developed by Hertzmann [27] [28] [29]. These methods include painting rough strokes, then more detailed strokes for areas above an error threshold [27], or treating the image as an energy function that must be relaxed [28]. Image Analogies [29] uses the idea of analogies to transfer a modification performed from one image to another image, as shown in Figure 2.9. Collomosse [15] paints strokes based on image salience. Salience is determined by calculating Gaussian smoothed derivatives over several standard deviations. Least salient brush strokes are painted first, then more salient strokes afterward.

Non-photorealistic rendering techniques provide a variety of ways in which images may be altered to produce more artistic or painterly effects. The different techniques combined provide more versatility than image vectorization alone. However, image vectorization is capable of creating varying levels of non-photorealism while still providing a vector-based graphic representation and the editability that comes with it.

Chapter 3

Methods

The purpose of our vectorization algorithm is to allow users to selectively create editable vector graphics of objects in an image. The graphic will consist of a hierarchical ordering of meshes corresponding to the subobjects in the image, as demonstrated in Figure 3.1. Each graphic mesh should be relatively sparse, and should be able to produce both realistic and stylized versions of the object. They should also be easily editable and scalable.

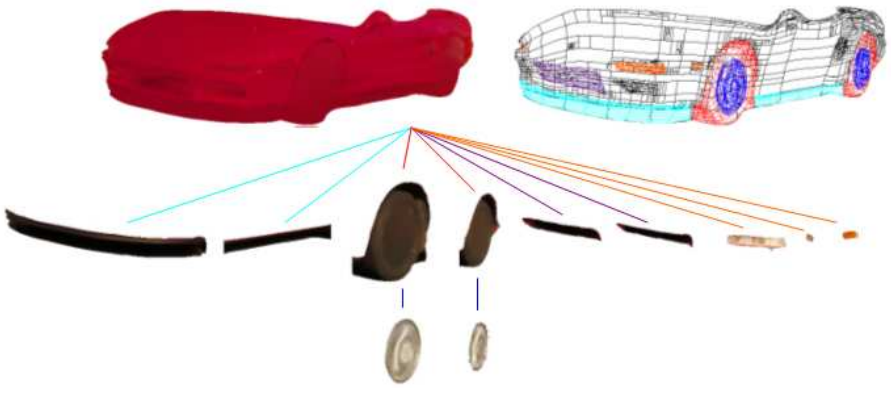


Figure 3.1: Hierarchy tree of the rendered body of the Ferrari (Figure 1.2(a)) and its subobjects

Section 3.1 explains the first step of our method, interactive object selection using an improved min graph cut segementation tool. Section 3.2 discusses the process of creating a graphic mesh that adheres to the object geometry and represents the object appearance within a user-selected level of detail. Section 3.3 introduces an automatic recursive subobject selection algorithm using min graph cut which segments subobjects from the main object, fills the holes behind the subobjects, then recursively treats each subobject as a new object and segments its respective subobjects. Section 3.4 discusses the object hierarchy which is created through the automatic subobject segmentation. Section 3.5 demonstrates the editability and object-level control our graphics provide. Section 3.6 explains the levels of detail, ranging from more abstract to more realistic, that may exist within a single graphic.

3.1 Object Selection

The first step in vectorization is selecting an object to be vectorized. Our user segmentation tool is similar to Lazy Snapping [35], but with significant improvements. We use graph cut to perform object segmentation in subsecond times, allowing the segmentation to update with each mouse movement, providing interactive feedback as opposed to only updating results after the mouse is released. Interactive rates are achieved by performing graph cut on a watershed hierarchy. Also, we perform a coarse-to-fine boundary localization with each mouse movement instead of doing so as a secondary step. Finally, our tool includes automatic background seeding, which allows a selection result to be displayed immediately after the first mouse click.

3.1.1 Graph Formulation and Weighting

We use the min graph cut algorithm in [13] which casts object selection as a graph optimization problem. A graph $\boldsymbol{G}$ is formed consisting of two terminal nodes

T_f and T_b, representing object foreground and background, and a set of nodes $\boldsymbol{P}$ for each pixel in the image plus an additional background node, p_b, not associated with any pixel. All nodes p are initialized as foreground or background (as seeded by the user) or unknown. The set $\boldsymbol{V}$ of all edges in $\boldsymbol{G}$ contains an edge (p, T_f) and (p, T_b) for each $p \in P$ as well as an edge (p, q) for nodes p and q corresponding to each neighboring pair of pixels. Each of these edges is weighted, and then the min graph cut algorithm divides the graph into two pieces, foreground (containing T_f) and background (containing T_b). We weight the edges of the graph as follows:

$$\begin{aligned} &\text{For } p, q \in \boldsymbol{P} \\ &\qquad (p, T_f) = \begin{cases} M \text{ if foreground} \\ 0 \text{ otherwise} \end{cases} \\ &\qquad (p, T_b) = \begin{cases} M \text{ if background} \\ 0 \text{ otherwise} \end{cases} \\ &\qquad (p, q) = \frac{1}{||C(p) - C(q)|| + 1} \\ &\qquad (p_b, p) = m, \end{aligned} \tag{3.1}$$

where M is a large number representing high cost, m is a small number used for background preseeding, and C(p) the color at node p. The node p_b is an initial background seed, and is (weakly) connected to each node corresponding to an image boundary pixel. Each edge between a boundary node and p_b has a low cost, m. This assumes that the boundaries of the image are background, but since the edge is such a low cost, it is easily overridden by the effect of foreground seeds in the same region as the boundary pixels. This allows object selection to begin with the first (foreground) stroke of the user.

3.1.2 Watershed Hierarchy

Similar to Lazy Snapping, we achieve a speed increase by operating over watershed regions instead of pixels. We use the watershed (toboggan) hierarchy seen in [38] [8] [40]. The first level of this hierarchy is created by performing a watershed algorithm on the pixels to oversegment the image. In other words, the gradient magnitude image is calculated, and each pixel then "slides" down the gradient magnitude image, moving from pixel to pixel according to which neighbor has the lowest value, until a low point is reached. All pixels sliding to the same low point form a watershed in the initial hierarchy level. Each subsequent level of the hierarchy is formed by grouping watersheds from the previous level according to statistical similarity calculated using the student's t-distribution. Each level of the watershed hierarchy completely segments the space, with each level of the hierarchy being completely composed of a union of watersheds from the previous level, except for the initial level which is a union of pixels. Watersheds tend to maintain gradients in the image, such that object boundaries are still well represented at higher levels of the watershed hierarchy.

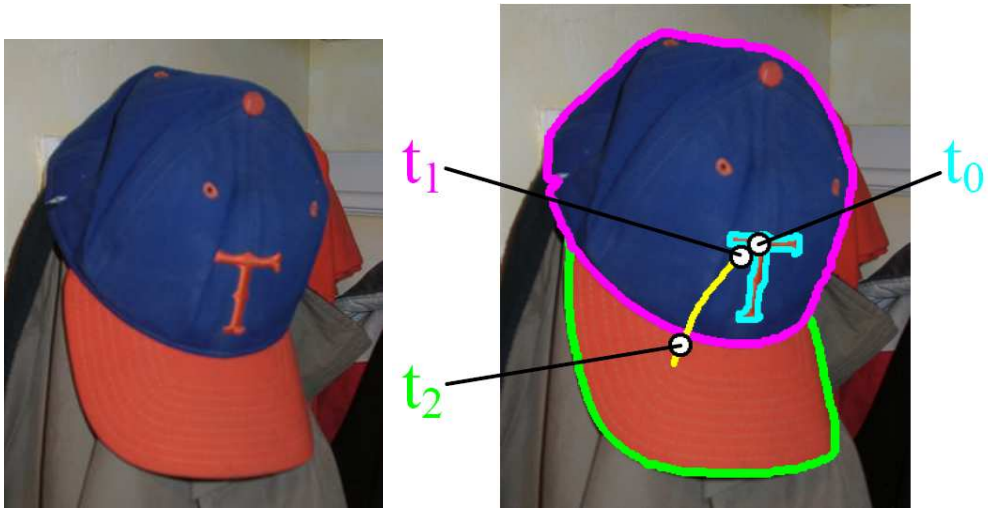
Since these watersheds exist in a hierarchy, this allows us to use a coarser level of the hierarchy if desired, reducing the size of the graph substantially, resulting in much faster response from the graph cut algorithm ($< .1$ sec), and therefore faster, interactive segmentation. Figure 3.2 demonstrates how regions from the initial watershed level (created by watershedding on pixels) are combined into larger regions. The regions in 3.2(a) are grouped and merged to create the larger watershed regions shown in 3.2(b). The grey boundaries in 3.2(b) are boundaries that were eliminated when the regions were grouped.

(a) (b) (c)

Figure 3.2: Watershed regions for (a) first level, (b) second level of hierarchy. Boundaries of combined watersheds are shown in grey. (c) Boundary refinement is limited to yellow pixels within watersheds bordering the coarse boundary (blue). Red (rim) pixels are used to seed foreground and background.

3.1.3 Persistent Graph Cut

As in [35] [41] the user marks the image with a (yellow) paintbrush to place foreground seeds (Figure 3.3). (Background seeds are initialized as described above.) With each mouse movement, min graph cut is called and the segmentation is displayed. Because min graph cut is called with each mouse movement, the user interaction follows a painting metaphor, where the user moves the cursor over the image, receiving constant and instant feedback, until the correct segmentation is computed. The background seed p_b allows the user to receive visual feedback with the first (foreground) mouse click, increasing the responsiveness of the tool. Figure 3.3 illustrates the user interaction. The user first clicks inside the "T", which selects it, as shown by the cyan outline in 3.3(b). As the user moves the cursor continuously (yellow path), the segmentation grows from the region bordered by cyan (t_0) to magenta (t_1) and finally to green (t_2). The selection updates as fast as the user can move the mouse.



(a) (b)

Figure 3.3: The segmentation changes at each t_i,. Segmentation boundaries are shown in cyan, magenta, and green. The user cursor path is shown in yellow. The time between t_is is as fast as the user can move the cursor ($<<$ 1 second).

3.1.4 One-Step Boundary Localization

With each mouse movement, graph cut is called for each level of the watershed hierarchy down to the pixel level. The catchment basins immediately adjacent to the object boundary at the bottom of the hierarchy form a rim (yellow region in Figure 3.4). Graph cut is applied automatically to the pixels in the yellow rim to localize the boundary (in magenta) to the pixel level. At each level of the watershed hierarchy, foreground and background seeds (green border of yellow rim) are taken from the inside and outside edges of the rim created by the watersheds around the initial boundary, and a new boundary is computed. Thus, for each mouse click or movement, an entire coarse-to-fine segmentation is computed. The localized boundary of the cap example is shown in Figure 3.5.

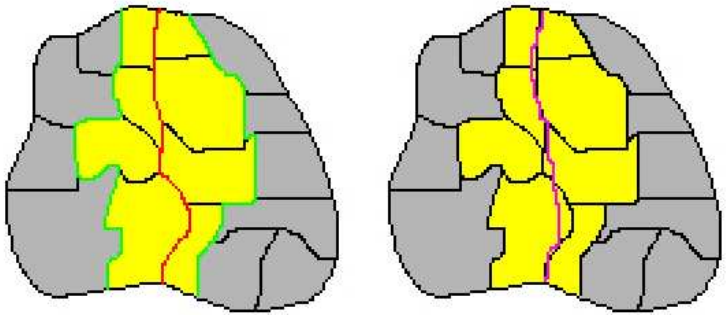


Figure 3.4: The initial segmentation of watershed regions produces a coarse object boundary (red). The pixels in the watersheds along the boundary (yellow) are used to refine the object boundary. Seeds are placed on the edges of the watersheds (green), and a finer boundary (magenta) is found.

(a) (b)

Figure 3.5: (a) The rim around the watersheds bordering the coarse segmentation (blue) become seeds for a new region (yellow) where min graph is applied, producing a new segmentation. (b) Final segmentation (green).

3.2 Mesh Creation

Once an object is selected, we create a mesh that allows the object to be rendered, scaled, and edited at hierarchical levels of detail. Although we constrain the mesh to be a warped, rectangular grid, we have found that this works well for a surprisingly

wide variety of objects. To create a mesh, we first locate four points on the object boundary to serve as corner points. The corners help define endpoints for a least-cost search that traverses the object, creating mesh axis lines. The resulting mesh is then rendered and refined.

3.2.1 Curvature Analysis for Corner Detection

The first step in mesh creation is locating four points along the object boundary to serve as corners of the rectangular grid (see blue dots in Figure 3.7(a)). This can sometimes be problematic, since many objects do not have four easily identifiable corners. We find candidate corner points by searching for points of high curvature along the boundary of the object.

Points of high curvature usually reveal relevant extremities in the geometry of the object silhouette, and are used to create a set of potential corner points. The curvature, κ_i, and slope, s_i, at boundary point, b_i, are computed by averaging over a window, w, for robustness to noise while still catching gradual bends in the object boundary, according to the equation

$$\kappa_i = \frac{1}{w} \sum_{j=i-w/2}^{i+w/2} (s_{i+1} - s_i), \tag{3.2}$$

where

$$s_i = \frac{1}{w} \sum_{j=i-w/2}^{i+w/2} (b_{i+1} - b_i) \tag{3.3}$$

and

$$w = max(P/20, 4) \tag{3.4}$$

for object perimeter P. The n=8 points with the highest κ_i comprise the set, C, of potential corner points. Additional potential corner points may be added to C if the current points are distributed in a "triangular" fashion, meaning that if the boundary were divided at each of the potential corner points, two consecutive segments would have a greater length than all other segments combined. Such an example is shown in Figure 3.6(b), where the boundary of the red part of the strawberry produces corners that form a triangular formation, with only one at the tip and all others near the leaves. When this occurs, the two long segments are each searched for a new corner point using Equation 3.2, producing two new corners as shown in Figure 3.6(c).

The best four corners from set C are chosen over $\binom{n}{4}$ according to

$$\max_{i,j,k,l \in C} ||i-j||+||k-m||+\alpha\frac{w}{l}(||j-k||+||m-i||)+\beta\sqrt{\frac{w}{l}+1}(||i-k||+||j-m||), \quad (3.5)$$

where $||i-j||+||k-m|| < ||j-k||+||m-i||$ and w, l = width, length of the minimum bounding box around the object. In other words, the edges between i and j and between k and m comprise the short sides of the rectangle to which the corners belong, as illustrated in Figure 3.6(d). The best corners are the combination of points that maximizes the lengths of the sides of this rectangle. However, the lengths of the long sides and diagonals of the rectangle are scaled to better reflect the shape of the object (by way of the bounding box ratios) and to reflect the relative importance of each (by scalars α and β). We generally set α =.83 and β =.33. If the computed corner points appear to be unsatisfactory, the user can easily adjust their position, subject to the constraint that the points are attached to the object boundary.

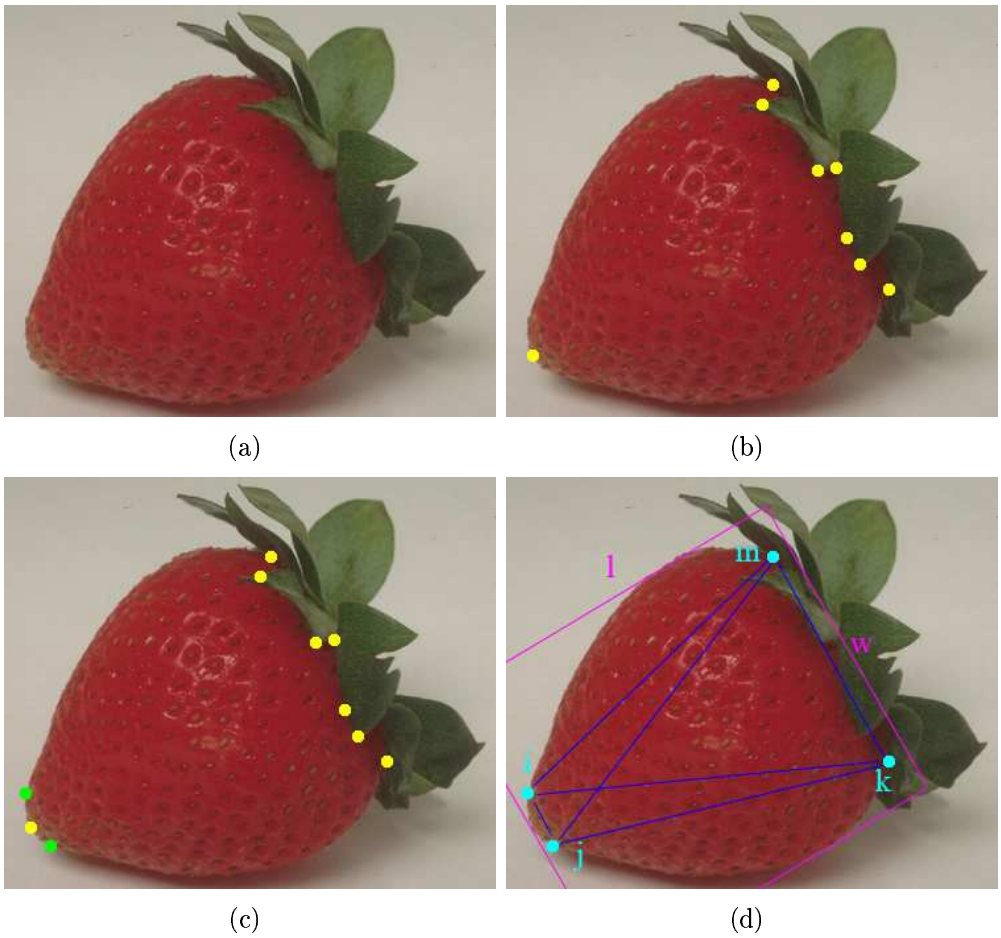


Figure 3.6: (a) The boundary of the red part of the strawberry (b) is searched for corners (yellow), producing a "triangular" distribution. (c) New corner points are added (green), and (d) the best four corners (cyan) are selected. The length and width of the magenta boundary box, as well as the distances between points (blue) are used in calculating the best corners.

3.2.2 Axis Creation

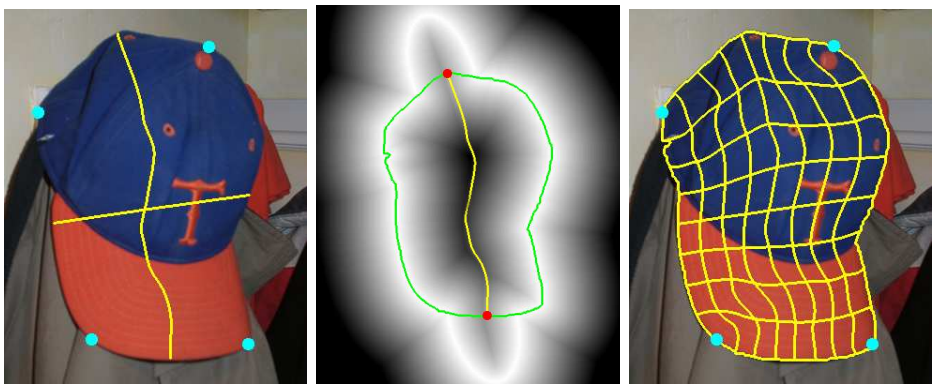
Once the corners have been found, mesh lines can be formed. Major and minor axes (Figure 3.7(a)) are first found by performing a least-cost search from the midpoint of one side of the rectangle across the object to the midpoint of the opposite side (red dots in Figure 4b). Each pixel is a node in the search and the minimum cost path

is calculated, in Intelligent Scissors fashion [37], using the local cost function, $l(p, q)$ between pixels p and q, where the cost functions f_D, f_G, and f_I refer to distance (Figure 3.7(b)), gradient and isocontour and w_D, w_G, and w_I are their corresponding weights. The local cost function is given as:

$$l(p, q) = w_D f_D(p) + w_G f_G(p) + w_I f_I(p) \tag{3.6}$$

$f_D(p)$ is a geometry cost based on the value of an inverted distance map at pixel, p, which coerces the minimum cost path for the major axis into the center of the object (yellow line, Figure 3.7(b)). $f_G(p)$ is a gradient cost, which draws the path toward interior object edges. $f_I(p)$ is a color isocontour, cost which penalizes the path for moving to pixels of different color. The weights w_D, w_G, and w_I, which set the importance of the each cost, are set (interactively) by the user using sliders. Since the gradient and isocontour costs often cause the axes to avoid subobjects that will be removed later (Section 4), we often set their weights to zero, as has been done in Figure 3.7.

Use of a standard distance map to compute the minimum cost path causes pixels around the search target to have high cost, due to the nature of the medial axis. This usually pushes the search away from the middle and causes undesirable results (Figure 3.8(b)). Simply removing the section of the boundary around the target often alleviates this problem, but in many cases this also removes needed information, which can cause the axis to veer away from its desired target (Figure 3.8(c)). To correct for this, we replace a section of the object boundary with a section of an ellipse oriented toward the low valley in the distance map to guide the axis more directly towards the goal (Figure 3.8(d)). More specifically, a low point is found by "walking downhill"

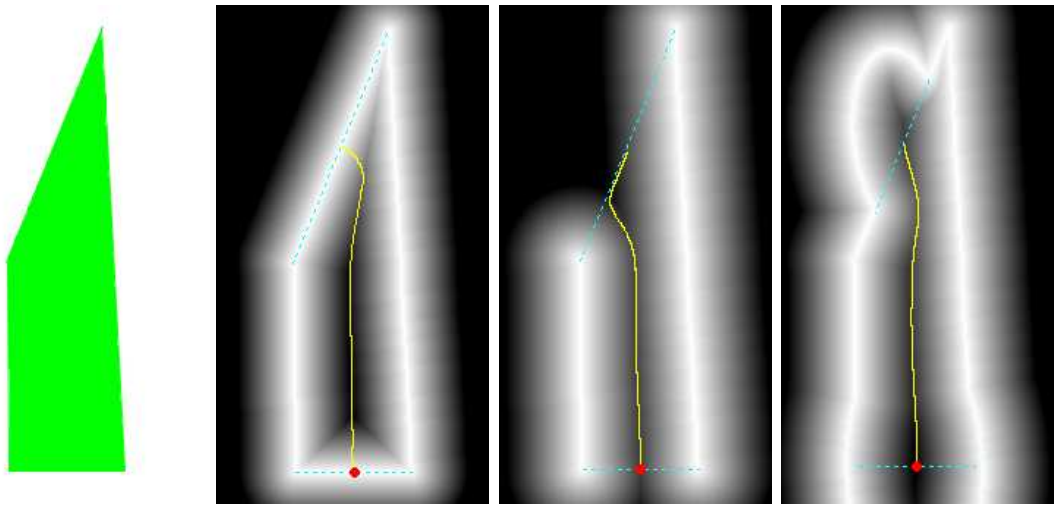


(a) (b) (c)

Figure 3.7: (a) Corner points (blue) and major and minor axes (yellow). (b) Distance map used to find major axis (yellow). White = low cost; black = high cost. Red dots = search endpoints. Object boundary for the cap is in green. (c) Resulting mesh.

from the two corners nearest the target, at each step moving to the pixel with the lowest cost, until they meet. This is assumed to be part of the medial axis of the shape, or the valley in the distance map. The algorithm then walks downhill from the target until it intersects the path from one of the two corners. The algorithm "walks uphill" as slowly as possible, meaning moving to the pixel with the smallest value greater than the value of the current pixel, from this point until it reaches a point p where the uphill path differs from its downhill path from the target. The ellipse is centered on the target and oriented such that the major axis points toward p. We set the length of the major axis of the ellipse to twice the distance from the target to p and the length of the minor axis to half the length of the major axis. These ellipses account for the unusual looking bulges at the top and the bottom of the distance map (as in Figure 3.7(b)), but it accomplishes the desired objective.

Once the major and minor axes are found, additional mesh lines are found by subdividing the object and applying equation 3.6 to each half recursively (Figure 3.7(c)).



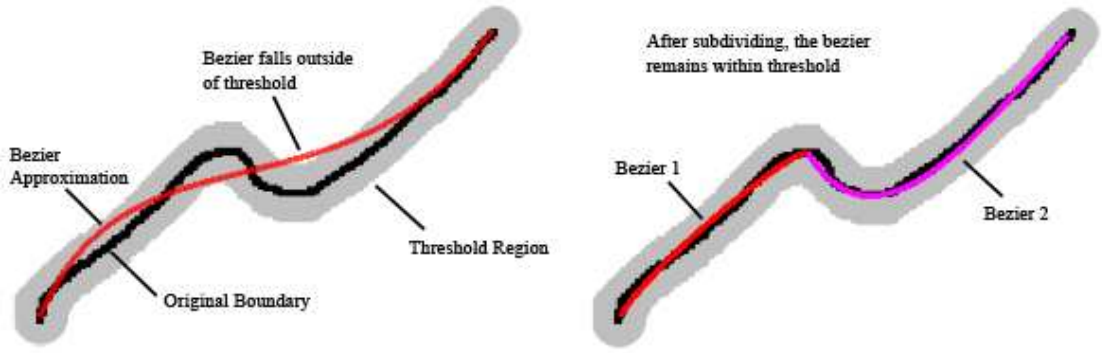
(a) (b) (c) (d)

Figure 3.8: (a) Original shape. (b) Its distance map, where black indicates low cost and white high cost. The red dot is where the search begins, and the yellow line is the axis. The high values near the search goal repel the axis causing it to bend. (c) Without the edge, the axis bends the other way until it hits the boundary of the object (shown by the cyan dotted line), then follows it to the goal. (d) With the added ellipse, the axis reaches the goal on a straighter (though slightly bent) coarse.

When doing so, the previously found axis becomes part of the new object boundary, and the endpoints of the axis become new corner points.

The algorithm as just explained produces a good mesh for the object interior for most cases. However, problems can occur at the object boundary. Since the object boundary is being approximated by Bezier curves, it will not always exactly fit the boundary. Often, this stylization is exactly what is wanted in a vectorization. Just as the color of the object interior is stylized, the boundary is stylized by becoming smoother. The problem occurs when the user wants the boundary to be more exact, or when two objects are touching. To allow for more exact boundaries, the user may adjust a boundary tolerance parameter. The tolerance describes the maximum

distance the graphic boundary may be from the true boundary of the object. If the boundary created, as previously described, exceeds this tolerance, the offending Bezier is subdivided. The maximal curvature point along the portion of the true boundary that the Bezier approximates is calculated using the curvature constraint described in Section 3.2.1, and the Bezier is split at this point. Figure 3.9 illustrates the process.



(a) (b)

Figure 3.9: (a) The Bezier curve (red) contains points whose distance from the original boundary (black) is too large. (b) After subdividing, the two new Bezier curves (red and magenta) fall within the threshold.

Even with an adjustable boundary tolerance, problems occur when objects share a common boundary. In this case, they are vectorized separately, and since the boundaries are not exact, there may be noticeable gaps between the objects. This is especially noticeable when automatically generated subobjects are suppose to meet but have small, often pixel-sized, gaps between them. The user may want a smoothed boundary for the overall object, but still may want bordering subobjects to touch. To handle this, when bordering subobjects are created, their common border is checked

for gaps by rendering the subobjects and their parent object each in a different color to encode them, tracing the subobject's common border, and looking for any pixel along that border which belongs to the parent object. If gaps exist, the boundaries are slightly nudged toward each other. This removes visible gaps while allowing an overall smoothed boundary.

When subobjects have a shape difficult to capture by a rectangular mesh, the object is automatically subdivided into more easily represented objects. Figure 3.10 shows an example, where the object formed by the two wheels and the shadow underneath the car has a difficult shape to represent and needs to be divided. The rectangularity of the object, the ratio of the object area to the area of the tightest bounding box, is calculated. If it is too low, the object boundary is searched for the point of highest curvature (red), using the algorithm in Section 3.2.1, which is also located in a concavity. Straight paths across the object interior are computed for discrete angles, and the shortest path to the opposing boundary (green) becomes the line of division given that the opposing intersection is some minimal distance along the object boundary (Figure 3.10(c). Each half is then treated as a separate object, having a separate mesh computed for it, and again being subdivided if necessary (Figure 3.10(d)).

3.2.3 Mesh Representation and Rendering

After creating mesh lines, we fit each mesh cell with a Bezier patch. The Bezier patch is formed by first approximating each side of the cell with a Bezier curve. This yields 12 of the necessary control points for the Bezier patch. Each additional control

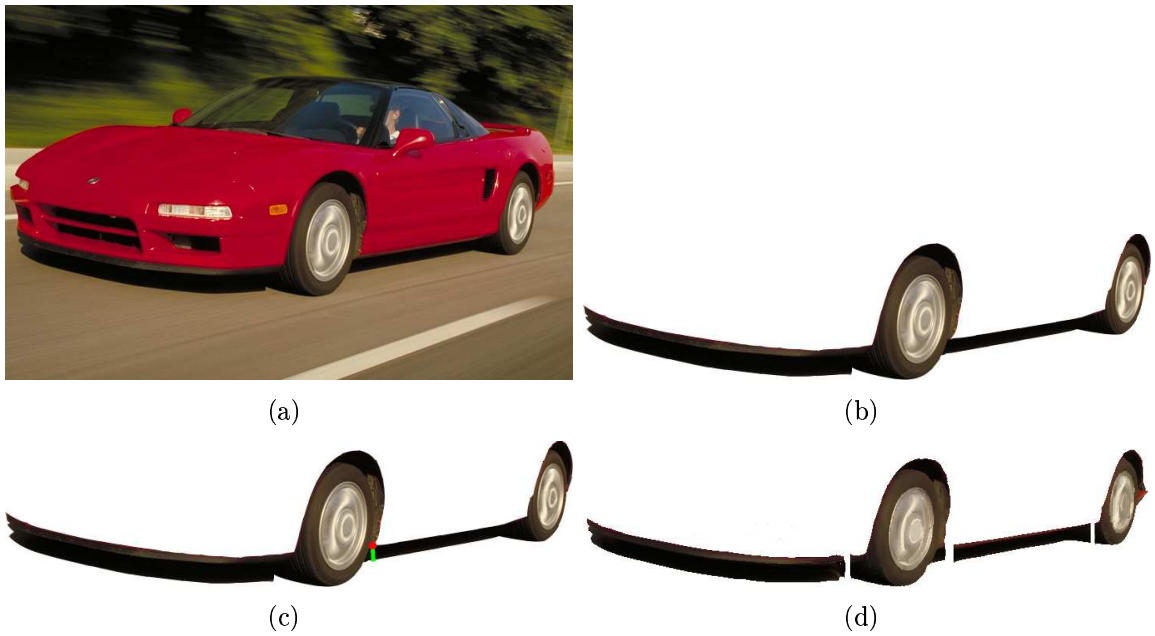


Figure 3.10: (a) Original image. (b) Shadows and tires form a difficult object to represent. (c) The highest concave curvature point (red) is found, and the shortest path across the object (green) is computed. (d) The object is divided in two, and the resulting objects are again divided, yielding four objects.

point, c, at position (i, j) in the 4x4 Bezier patch control grid is interpolated using:

$$\begin{aligned} c(i,j) &= u_2 c(0,j) + u_1 c(3,j) + v_2 c(i,0) + v_1 c(i,3) - \\ & \quad u_2 v_2 c(0,0) - u_2 v_1 c(0,3) - u_1 v_2 c(3,0) - u_1 v_1 c(3,3), \end{aligned} \tag{3.7}$$

where

$$u_1 = i/3, u_2 = 1.0 - u_1, v_1 = j/3, \text{ and } v_2 = 1.0 - v_1 \tag{3.8}$$

After the geometry of the patch has been determined, colors must then be assigned to each of the control points. We choose colors by sampling the image from the point of the patch to which the control point corresponds. The corresponding point on the

patch surface for a given control point is calculated using de Castlejau's algorithm on the columns and rows of the Bezier patch and assuming values of t = 1/3 and t = 2/3 for the interior control point on each Bezier curve. The sampled color is assigned to the control point, and the patch is shaded using OpenGL. The OpenGL rendering is explained in more detail in B.

3.2.4 Mesh Refinement

After being rendered, the meshes are refined to provide higher resolution of detail. Each patch containing points whose render error (difference between original and rendered versions) is above a user-selectable threshold (level of detail) is subdivided as shown in Figure 3.11, where the center grid was divided in half, and one half into fourths, to produce the grid on the right. These smaller patches provide more control points and thus more color samples over the region of error. This process is iterative, checking all new patches to see if their errors are below threshold, and subdividing if necessary. This process continues until a predetermined number of iterations occur or until all patches fall below the error threshold or a user-defined size. Each iteration runs in subsecond times for normal-sized objects or up to a second for large objects or objects requiring many refinements. We generally refine the mesh after subobject selection (explained in Section 3.3) to avoid refining regions which are occluded by (nested) subobjects.

Mesh refinement may cause T-junctions in the mesh, as shown on the right grid in Figure 3.11. These can potentially cause discontinuities in the surface because more control point are used in calculating the color gradients on one side of the junction (the right side in Figure 3.11) than on the other (the left side). However, in practice if the color is varied enough to cause a discontinuity, there will be enough error to

force both sides to subdivide, maintaining the same distribution of control points and preventing problems (if the left side also divided in half, the control points would be aligned with those on the right side).

Figure 3.12 illustrates the increase of resolution achieved using a refined grid. The original grid resolution in Figure 3.12(b) is used to create the rendering shown in Figure 3.12(e). The refined grid in Figure 3.12(c) produces a rendering (Figure 3.12(f)) which more closely matches in original (Figure 3.12(d)).

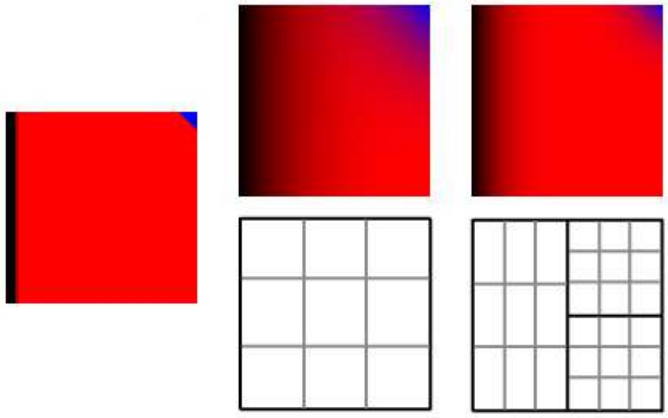


Figure 3.11: The image on the left is approximated with an initial grid (center) and a refined grid (right). Notice how the colors produced by the refined grid more accurately represent the image.

3.3 Automated Recursive Subobject Segmentation

Usually complex real-world objects consist of multiple subobjects as well, each of which could contain its own subobjects. While we could select subobjects individually, as described above, and create grids and render them individually, this would soon become unacceptably tedious for 10's to 100's of subobjects. To automate the process of subobject selection, we apply graph cut to the selected object recursively. With-

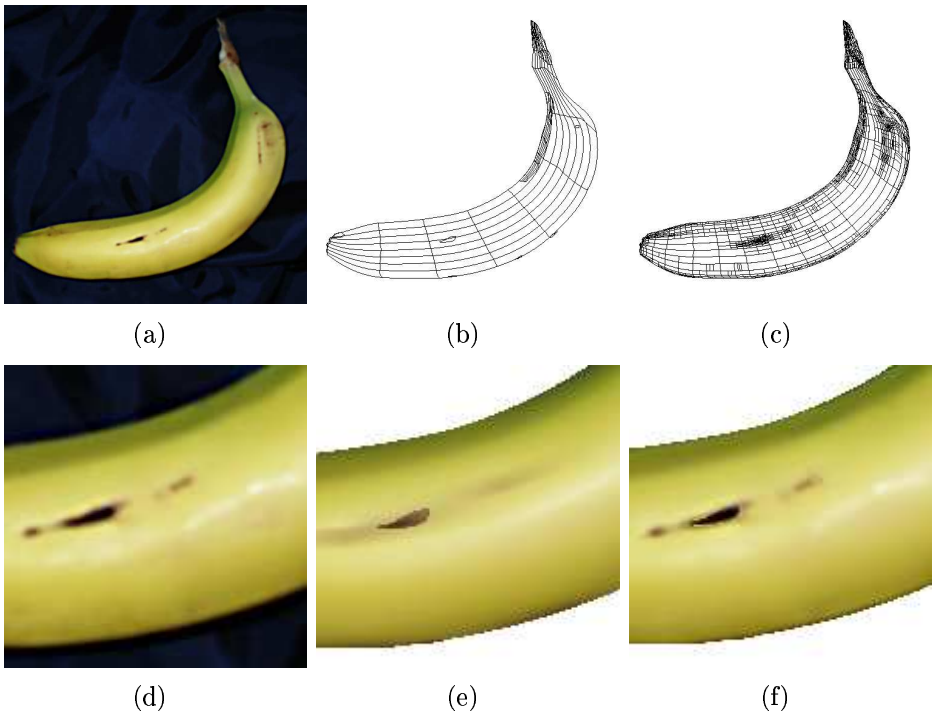


Figure 3.12: (a) Original image. (b) Coarse grid (103 patches). (c) Refined grid (1406 patches). (d) Zoomed portion of original image. (e) Coarse rendering (5.75 error per pixel) from (b). (f) Refined rendering (2.08 error per pixel) from (c) after 5 refinement iterations. The total running time for all five iterations was about 2.5 seconds.

out loss of generality, we assume that the initially selected object is heterogeneous, consisting of multiple (nested) subobjects, which we treat, alternately, as foreground and background. Subobjects are automatically detected and removed, with the space behind them being filled. Subobjects are then treated as new objects and are also segmented recursively, etc.

The main steps of the algorithm are as follows:

1. Automatically seed foreground and background.

2. Fill background.

3. Segment subobjects with min graph cut.

4. Vectorize subobjects and recursively segment their subobjects.

3.3.1 Automatic Foreground/Background Seeding

Subobjects are identified as regions whose color is significantly different from the overall color of the object. These regions are determined based on the difference between the initial, coarse rendered object (Figure 3.13(a)) and the corresponding pixel values in the original image (Figure 3.3(a)). This difference is thresholded to find the largest connected component, C, of pixels (yellow color in Figure 3.13(d)) whose error is less than some (small) threshold, ε_s. We assume C to be representative of the bulk of the currently selected object, and thus assume its color to represent the color of the main object without the subobjects.

3.3.2 Background Filling

One of the advantages of vector graphics over raster images is that even occluded portions of an object are defined, such that moving an occluding (sub)object does not create a hole in the image. Accordingly, areas of objects occluded by subobjects are filled, which not only provides a means of background completion but also provides information for the recursive segmentation of subobjects via graph cut.

Areas of an object which are occluded by subobjects are filled by estimating the color of occluded mesh nodes using a least squares fit (LSF), as illustrated in Figure 3.14. The set of control points N_c whose color is within 3σ of the mean of the largest connected component C are assumed to represent the homogenous object, and are used as data by the LSF to replace the color of the remaining control points, N_b. The control points of an object form a grid, as shown in Figure 3.14(a), where the black boxes indicate control points to be filled and the colored squares represent

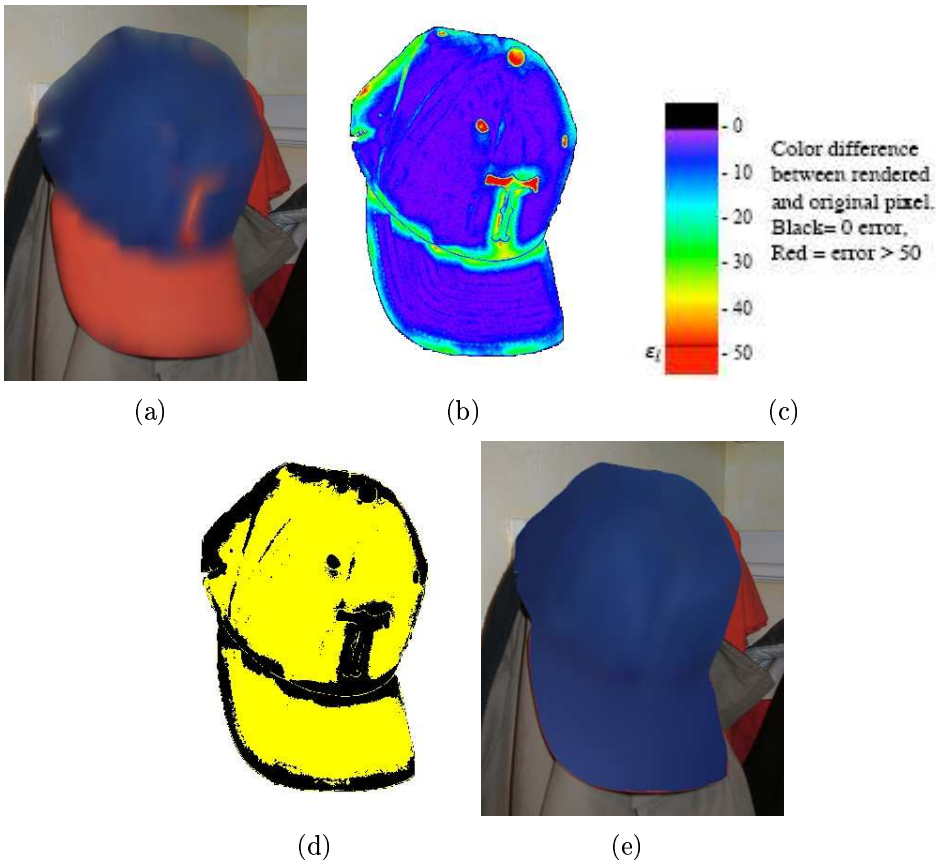


Figure 3.13: (a) Initial rendering of the coarse grid (Figure 3.7(c)). (b) Error between original image and initial rendering. (c) Error legend = RGB color difference between rendered and original image. (d) Connected components (yellow) of pixels below (small) error threshold, ε_s. (e) Coarse grid rerendered using colors from largest connected component.

the control ponints in N_c with their respective colors. The color of the control points in N_b is determined by performing an LSF over each row and column of this grid using the colors of control points in N_c (Figures 3.14(b) and 3.14(c)). The color at each point in N_b is calculated as the average of the colors produced by the LSF over the point's corresponding row and column. After one pass, there may remain points which have not been filled because neither the column nor row they belonged to had

any points initially, such as the upper right point in Figure 3.14(d). A second pass of this algorithm is performed to guarantee each control point is filled (Figures 3.14(e)-3.14(g)).

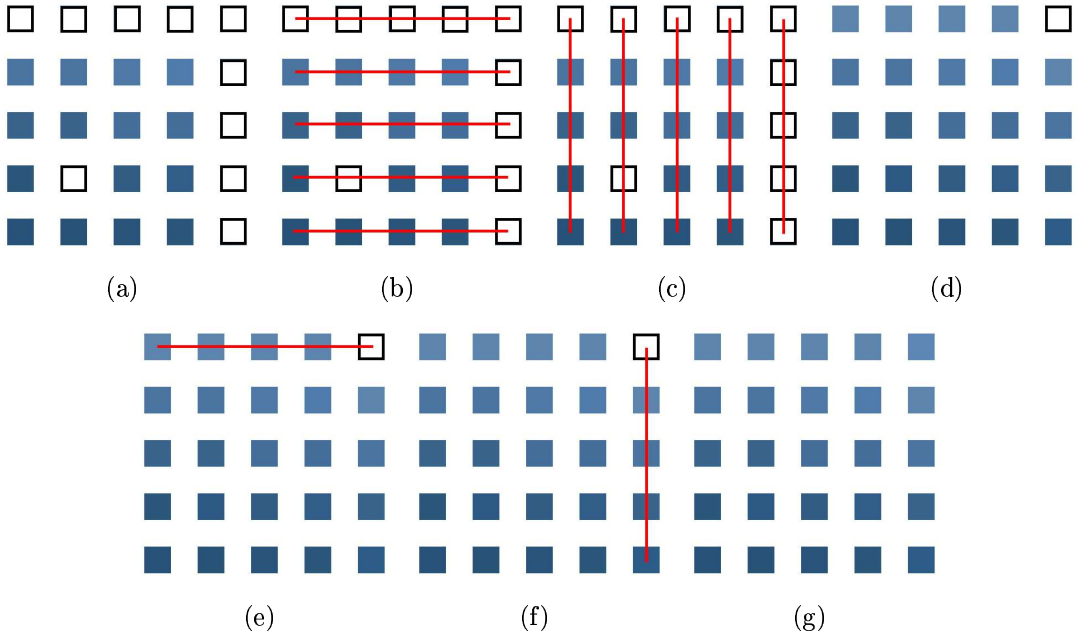


Figure 3.14: (a) Initial control point grid. Black boxes indicate points to be filled. (b) LSF performed over each row. (c) LSF performed over each column. (d) Control points are filled by averaging the result from the row and column. Additional holes are filled by performing LSF on each (e) row and (f) column to produce (g) the final result.

However, this method, as described, has problems when a given row or column does not have enough control points to yield a good approximation. This causes streaks in the rendered graphic. To compensate for this, the slope and intercept of each line produced by the LSF for the rows or columns is also fit with a LSF, which is then used to replace the colors in the control points (Figure 3.13(e)).

This hole-fill is a temporary fill to assist in subobject selection rather than a final hole fill, due to the fact that the exact subobjects are not yet known and consequently

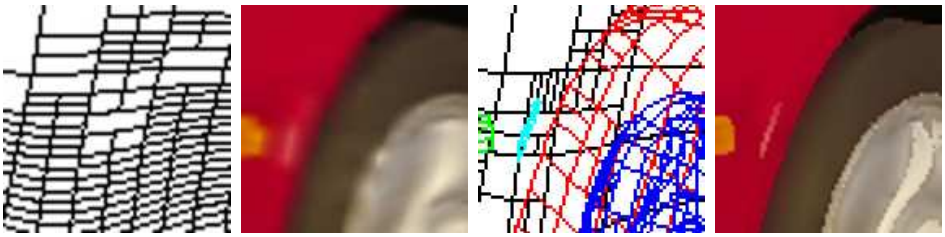
the regions needing to be filled are not precisely known. After all objects have been selected, the color estimation is again performed over each object. Since subobject locations are now certain, reperforming the hole fill corrects errors in the object color caused by control points incorrectly classified as background or subobjects.

3.3.3 Subobject Segmentation

Once the objects main color has been established, the rendered object is treated as the primary background for this object (Figure 3.13(e)). We then subtract this newly rendered object from the original image pixels to generate an error map. Areas of large error, ε_l, (i.e. red), where ε_l is user-selectable, indicate the presence of subobjects (Figure 3.16(a)), and act as foreground seeds, while pixels in C (from Figure 3.13(d)) act as background seeds. These seeds are then sent to min graph cut to segment the entire object into subobjects. Thus the error map serves as a mask for the parsing of the originally selected object into its component subobjects. The same weighting scheme used in Section 3.1 is used to set the weights for the subobject segmentation.

This algorithm locates regions that have large color differences from the supposed background color to separate them into subobjects. This greatly aids in rendering, because the edge between an object and its subobject will become blurred if a Bezier patch spans them (Figure 3.15(b)). In order to reduce the blurring, the patches must be subdivided excessively, adding undesired bulk to the mesh. Subobject representation allows these edges to remain sharp (Figure 3.15(d)). Additionally, subobject segmentation increases object-level control by allowing subobject to be edited separately from their parent.

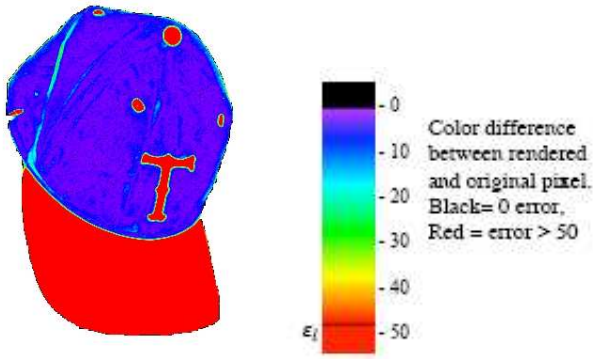
However, subobjects do not always coincide with objects that the user would desire. This is especially the case when a subobject is similar in color to the background.



(a) (b) (c) (d)

Figure 3.15: Without subobject segmentation, the (a) graphic is (b) blurred unless excessive subdivisions occur. With subobject segmentation the (c) graphic remains (d) sharp.

For this reason, we allow the user to manually select subobjects using the min graph cut method described in Section 3.1. In fact, and as an important note, preselection of multiple, conspicuous subobjects to begin with (e.g. eyes and mouth on a face, green and red objects in Figure 3.19, or straps and labels on the baseball glove, green objects in Figure 4.12(b)) is very simple to do and results in a much more regular mesh for the underlying main object, because the mesh lines are not deflected by the subobjects.



(a) (b)

Figure 3.16: (a) Error map used to detect subobjects. Pixels $> \varepsilon_l$ become foreground seeds for graph cut segmentation. Pixels with low error (Figure 5b) = background seeds. (b) Error legend = RGB color difference between rendered and original image.

3.3.4 Subobject Vectorization and Recursive Segmentation

After a subobject is segmented, it is treated as a separate object and vectorized independently. The entire automatic subobject segmentation process is then repeated for each of it. This process is illustrated in Figure 3.17. Here, the initial object to be selected is the body of the Ferrari, shown in the first column. The body becomes the background object (second column) with its holes filled and foreground objects (such as the tire in the third column) are generated. The automatic segmentation is then applied recursively by considering each foreground object (tire, second row in Figure 3.17) as a new object from which new completed backgrounds (solid tire) and foreground objects (tire hub cap) are generated.

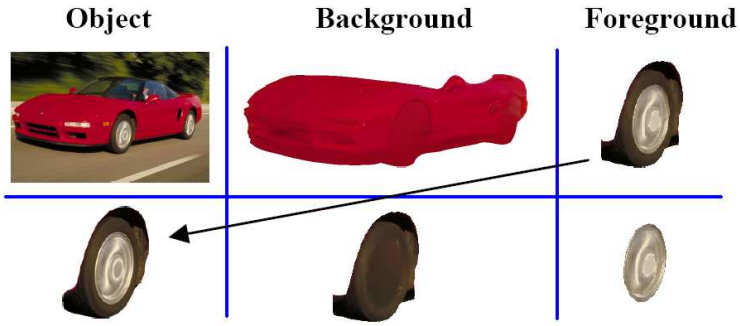


Figure 3.17: Recursive graph cut. Each object to be segmented has a background (red car body) and foreground object (tire) computed. Foreground objects in turn become background objects to be segmented at the next level.

3.3.5 Automatic Segmentation Algorithm

A summary of the algorithm is as follows:

Algorithm 1: Recursive Object Parsing using Min. Graph Cut

1. Select entire (heterogeneous) object (Fig. 3.3)

2. Create (coarse) grid (Fig. 3.7(b))

3. Render coarse grid (Fig. 3.13(a))

4. Compute difference, $D =$ | rendered object - original image |

5. Find largest connected component C of pixels with error $< \varepsilon_s$ in D (Fig. 3.13(d))

6. Create histogram of pixels in C and compute mean, m_c

7. Revisit (coarse) Grid:

   (a) If node color $> 3\sigma$ from m_c replace with one-dimensional least squares fit (LSF) from surrounding node colors

   (b) Then replace with global, bidirectional LSF for horizontal and vertical grid lines

   (c) Rerender coarse grid with updated node colors (Fig 3.13(e)) (This is the rendered object background O_b)

8. Subtract O_b from the original image pixels

9. Find each connected component, K_i, in diff. image with error $> \varepsilon_l$ (Fig. 3.16(a)) (Note: ε_l is user-selectable, default = 25)

10. Repeat steps 10a. and 10b. recursively

    (a) Invoke graph cut to segment each K_i into subobjects, S_j

    (b) Repeat steps 2-8 for each S_j

11. Repeat LSF for all objects using color known to belong to object

12. Perform mesh refinement (Fig. 3.11)

(a) Render object at initial coarse level

(b) Subtract rendered object from original image pixels

i. For each patch

A. If (local) error > user selectable threshold subdivide patch and repeat the process

3.4 Object/Subobject Hierarchy

Previous object-based representations of images, such as [8], have no way of storing information about the relationship between objects. Rather, each object is represented independent of all other objects. However, users often intuitively understand images on an object level, and assume relationships between objects.

Objects in images generally relate to one another in a hierarchical manner, as seen in Figure 3.18, with main objects containing subobjects which themselves contain subobjects. In an image, these can best be thought of as layers, with each two-dimensional subobject layer lying on top of its parent object layer. In this manner, an entire scene may be constructed as a hierarchy, with the main background composing the parent layer for the entire hierarchy. Through our subobject selection, we create such a hierarchical structure for storing object information.

This provides a more powerful means of object-level control, allowing edits on parent objects to be propagated (optionally) to children, while children can still be edited independently of the parent. Because objects are completely filled, subobjects can also be deleted or removed from their parent object without leaving holes in the parent. The separation of subobjects from their parents not only provides a more intuitive object-level control but also allows for edits that cannot be performed with previous techniques.

Often subobjects which may be important for accurate rendering will be insignificant to the user. The hierarchy allows the user to work with substantial objects without having to worry about their subobjects. Objects can also be regrouped by the user to allow for better control.

Hierarchical ordering of objects and subobjects is illustrated in Figure 3.18. Here, some of the major subobjects of the Ferrari, such as the wheels, shadows, and lights, are shown. The hubcaps, being subobjects of the tires, indicate the multi-tier capability of the hierarchy. Other subobjects important to rendering but likely not relevant to the user, such as the highlights on the car body or hubcaps, are not shown in the diagram, but likewise may be ignored by the user because the hierarchy allows for them to be edited along with their parent object. This is especially important for complex objects which may be comprised of hundreds of such objects.

Additional examples of subobjects are shown in Figure 3.19.

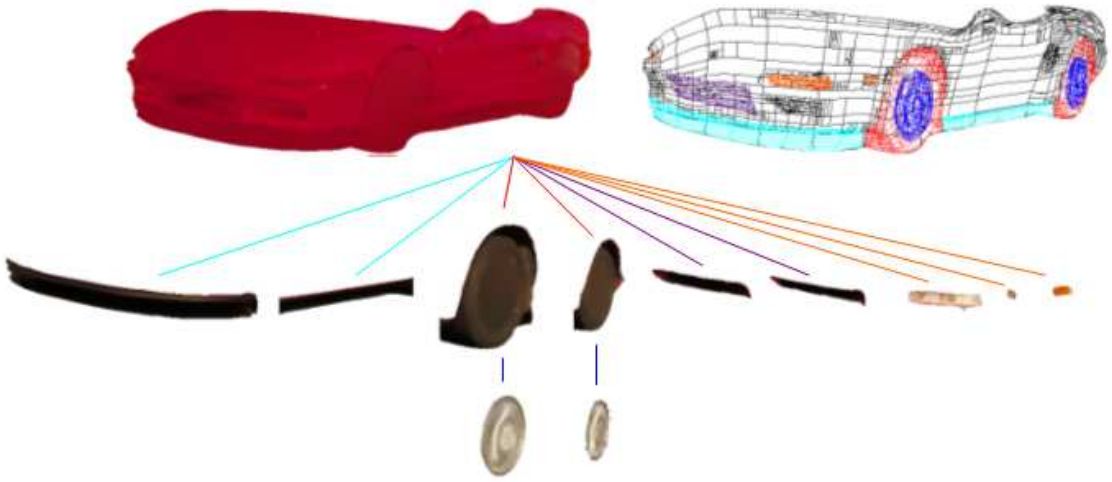
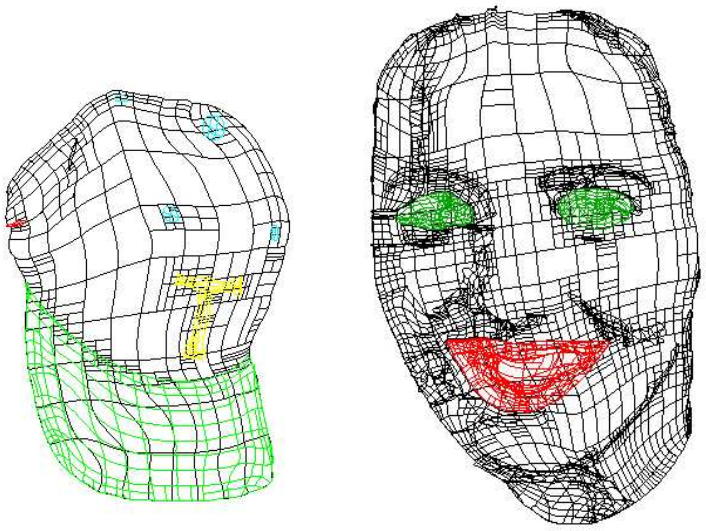


Figure 3.18: Hierarchy tree of the rendered body of the Ferrari (Figure 1.2(a)) and its subobjects



(a) (b)

Figure 3.19: Subobject meshes computed for cap and model face.

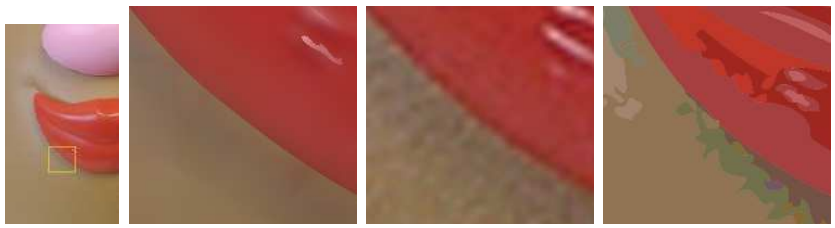
3.5 Object Editing

With image objects now represented as a renderable mesh, a variety of editing operations become available. Our graphics may be manipulated by pulling control points, much like other stroke-based or gradient mesh-based graphics, thus making themselves candidates for currently existing tools. However, our hierarchical object-based structure provides more direct and simplified control of image objects, allowing both object and subobjects to be edited together. Additionally, the object hierarchy automatically fills the areas under subobjects, preventing holes from appearing during editing. Some specific types of editing are illustrated below.

3.5.1 Object Scaling

One of the most common image editing operations is that of scaling, which may use a variety of interpolation techniques. While much good progress has been made in image interpolation [47], it still causes pixelation at high scales (Figure 3.20(c)).

Since our vector graphics are scalable, this pixelation does not occur (Figure 3.20(b)). Current vectorization techniques also allow for scaling without pixelation, but since they are essentially flat-filled regions, all natural shading is lost and unwanted edges appear (Figure 3.20(d)).



(a) (b) (c) (d)

Figure 3.20: (a) Potato Head's mouth (yellow region) is shown at 700% (b) using image vectorization, (c) bicubic interpolation, and (d) Adobe Illustrator Live Trace.

3.5.2 Interactive Object and Subobject Editing

As vector graphics, our objects are designed for easy editing. We make use of editing tools similar to those described in OBIE [8] to edit our meshes (Figure 3.21). These tools allow users to apply various transformations and deformation to objects. Using the object hierarchy tree, the edits can be propagated to the subobjects to allow for natural object level editing. This provides individual object and subobject control, allowing the user to easily perform complex edits that cannot be replicated with warping techniques without heroic effort.

In order to maintain interactivity in editing complex objects, only the coarsest level of detail is displayed while editing, as shown in Figure 3.22(b). This provides better visual feedback than simply displaying the mesh, as other vectorization techniques do (Figure 3.22(c)).

(a) (b) (c)

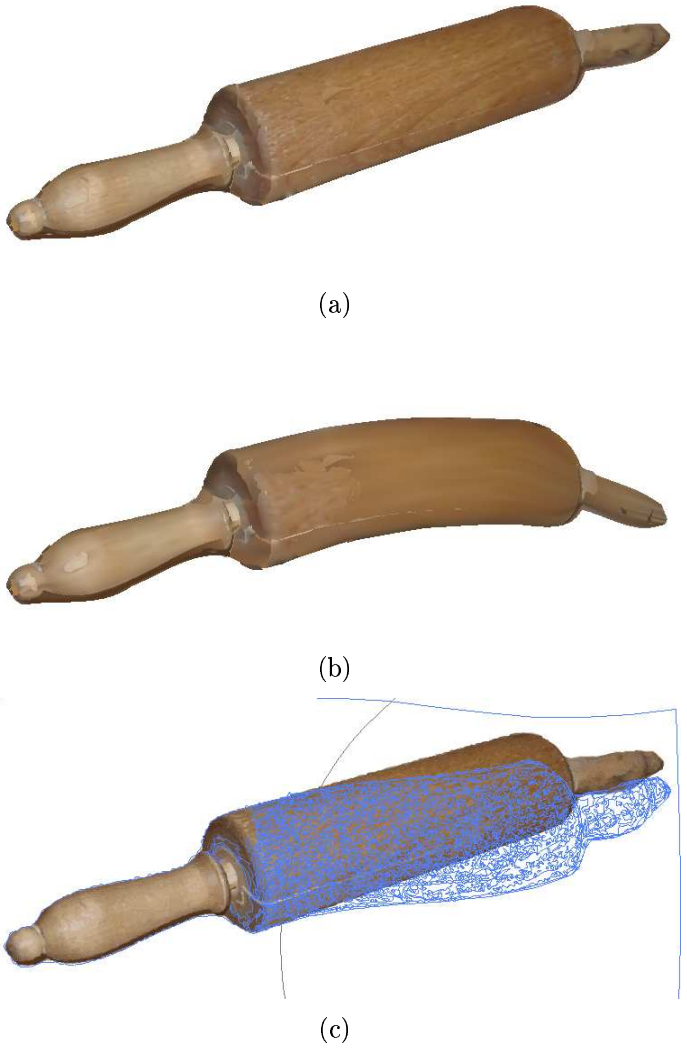
Figure 3.21: (a) Photo of a jacket (282x480) and (b) its graphic (6 objects, 4378 patches. (c) A simple edit shows we have nothing up our sleeve.

3.5.3 Hole Filling

When subobjects are created, the area beneath them is automatically filled as described in Section 3.3.2. Because of this, when subobjects are edited independently from their parent object, no holes are left in the parent object, as seen in Figure 3.23.

3.6 Progressive Levels of Detail

After an object or subobject is created, it is refined over a series of iterations to add more detail and create a more realistic rendering of the original object, as described in Section 3.2.4. Before refinement, the graphic of the object with its subobjects represents a rough approximation of the object which, although lacking in some detail, gives the general appearance of the object and conveys the basic shapes and colors of the scene. Since additional detail is added iteratively, successive iterations of the refinement form a hierarchy of detail.



(a)

(b)

(c)

Figure 3.22: (a) Original render by Image Vectorization. (b) During edit, lower level of detail shown to maintain interactivity. (c) Adobe Illustrator maintains interactivity by displaying grid.

Figure 3.24 illustrates such a detail hierarchy. From the initial rendering in 3.24(b), the rose is identifiable, but appears to be a graphical illustration rather than a real object. Many parts are overly smoothed, and expected detail is missing. With more refinements (3.24(c) and 3.24(d)), more detail is seen and a greater level

(a) (b)

Figure 3.23: (a) Wounded leg. (b) The cut is "healed" by removing the cut and leaving the hole fill behind. Only the region around the cut is vectorized.

of realism is achieved.

These progressive levels of detail allow the user to select from or specify varying levels of stylization. They also may provide a means of progressive transfer of data. While many patches are needed for the most refined renderings of images (3.24(d)), initial levels of the hierarchy capture much of the information at a far lower storage cost. Figure 3.25 shows the number of patches required to represent the rose from Figure 3.24(a) at the given error per pixel per channel in RGB space. As can be seen, the number of patches can be greatly reduced while incurring only a small increase in the overall error.

(a) (b) (c) (d)

Figure 3.24: Progressive Levels of Detail. (a) The original photo is progressively refined to show greater detail. Results are shown after (b) 0 iterations, (c) 2 iterations, and (d) 4 iterations of mesh refinement. The graphics and renderings of all levels of detail are shown in Figure 4.26.

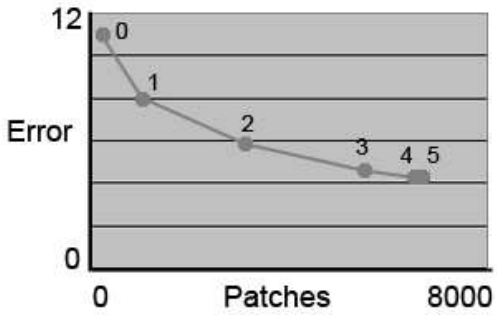


Figure 3.25: The error per pixel per RGB channel for a given number of patches for the renderings of Figure 3.24. The number next to each data point indicates the iteration number which produced the given data. The graphics and renderings of all levels of detail are shown in Figure 4.26.

Chapter 4

Results

Image vectorization can be used for versatile, object-based image editing, photorealistic (or non photorealistic) rendering, or even reverse-engineering a graphic. Image vectorization results compare favorably with those produced by other vectorization programs. Figure 4.1 shows a typical result of image vectorization. The original image (Figure 4.1(a)) is segmented as shown in Figure 4.1(b). Since a single graphic represents detail in a hierarchy, the rose can be rendered as a lower resolution of detail (Figures 4.1(c)- 4.1(d)) and at a higher resolution (Figures 4.1(c)- 4.1(d)) as specified by the user. The high resolution version has been modified to demonstrate editability. The rose petals' fall is created by translating, scaling, and warping the petals using the techniques explained in Section 3.5, all of which may be performed in a matter of seconds.

Figures 4.2-4.3 show more results of vectorized image objects. The bowl, Figure 4.2(a), was selected in <1 sec., followed by mesh creation (∼5 sec.) and segmentation and mesh creation of subobjects (∼40 sec.). The board, Figure 4.3(a)

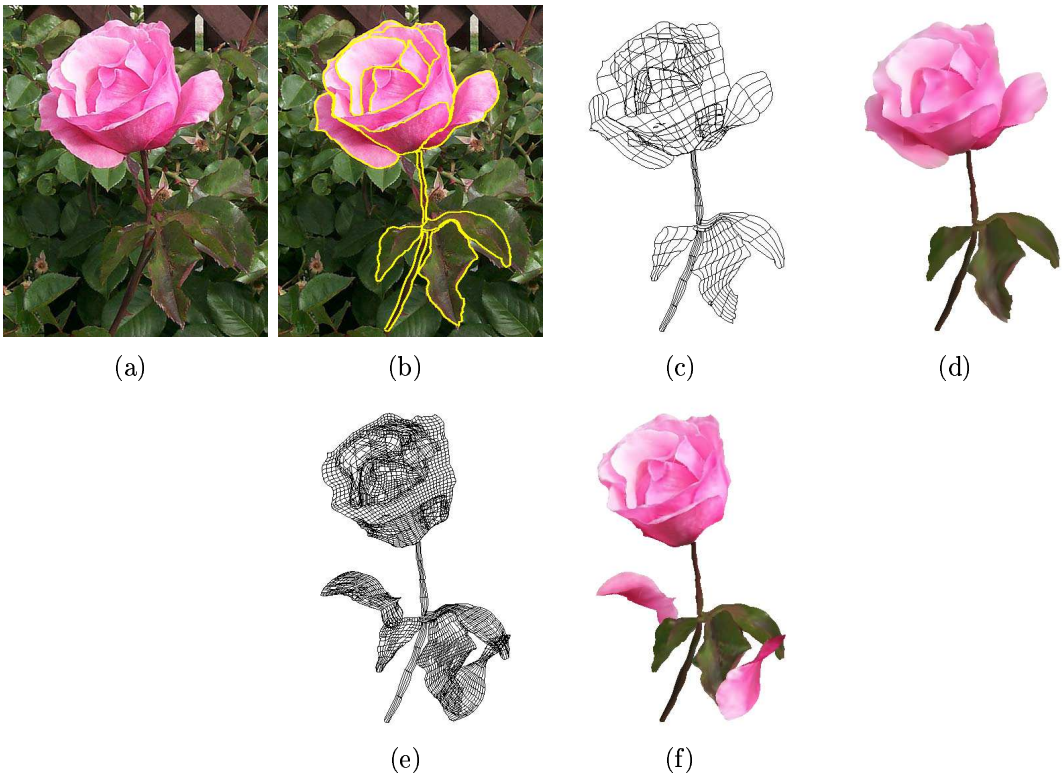


Figure 4.1: (a) Original raster image. (b) Hierarchical graph cut segmentation into 29 objects (c) Vector graphic (367 Bezier patches). User-selected (low) level of detail and (high) stylization. (d) OpenGL rendering of (c). (e) User-selected higher level of detail (3069 patches) computed in about 50 seconds. (f) OpenGL rendering and editing of the mesh shown in (e) "She loves me ... she loves me not ... "

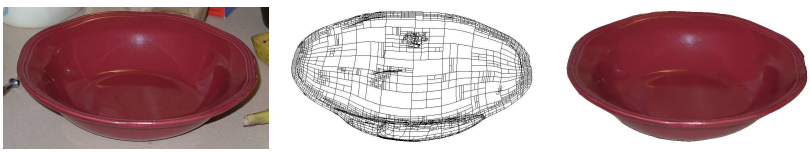
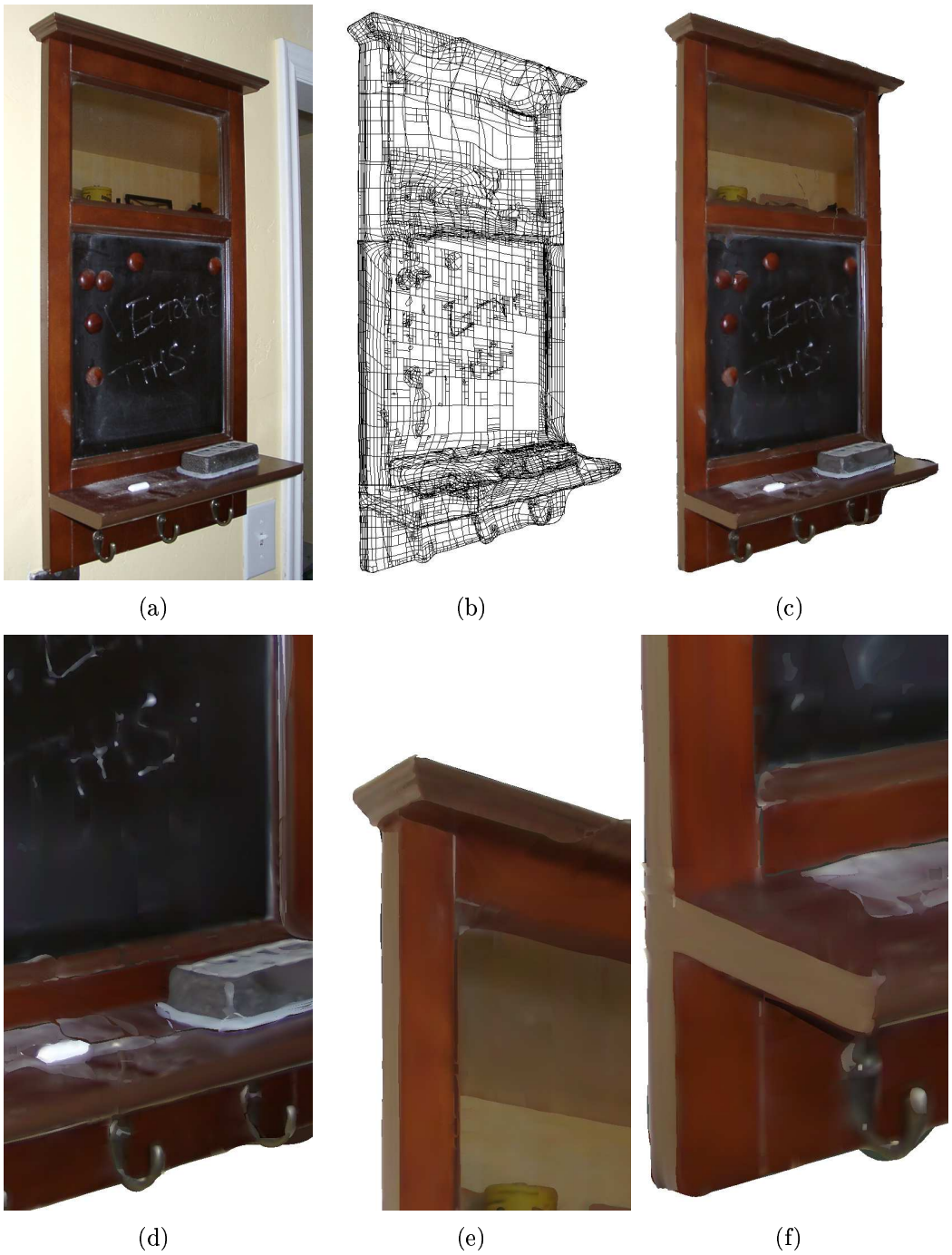


Figure 4.2: (a) 800x427 image (b) Mesh (c) Rendered: 64 objects, 2894 patches.

was selected in 1-2 seconds, followed by mesh creation (∼5-10 sec.), with about 1-2 minutes for recursive segmentation and mesh creation for all of the 250 subobjects.



(a) (b) (c)

(d) (e) (f)

Figure 4.3: (a) 550x1024 image. (b) Mesh. (c) Rendered: 250 objects, 6346 patches. (d, e, f) Close-ups of rendering.

4.1 Comparison to Other Vectorization Techniques

Image vectorization produces results quite different than those typically generated by other vectorization techniques. One such example is our ability to capture smoothly shaded surfaces, as shown in Figure 4.4. Since other vectorization techniques (such as Adobe Illustrator Live Trace, Corel CorelTRACE, Siame Vector Eye, Macromedia Flash, and AutoTrace) use arbitrary flat-shaded regions defined using polygons or curves, they cannot capture these smooth surfaces but rather produce a striated or terraced look with obvious contouring.

Since our graphics are object-based and hierarchically structured, it is easy to edit objects and subobjects. Because of the hierarchy, subobjects can be edited with their parent object, or object and subobjects can be edited separately, often revealing the filled, occluded portions of objects higher in the hierarchy tree. Other vectorizations lack such a hierarchy, causing them to create separate, disassociated objects which must all be selected before an editing operation can take place. After such operations, holes in the background are usually visible. Figure 4.5 demonstrates a simple translation using Image Vectorization and Adobe Illustrator, which is typical of other vectorization techniques. The label in Image Vectorization is an object, so to move it the user simply has to select the label and move it. The area behind the label is already defined, such that no hole is left in the image. To move the label using Illustrator, the user must select all 163 objects which comprise the label, which is tedious, time-consuming, and error-prone, and then translate them (as a group). Also, a hole is left in the glove because the area under the label has not been defined by an object.

Complex edits, such as warps, are easily created in Image Vectorization using

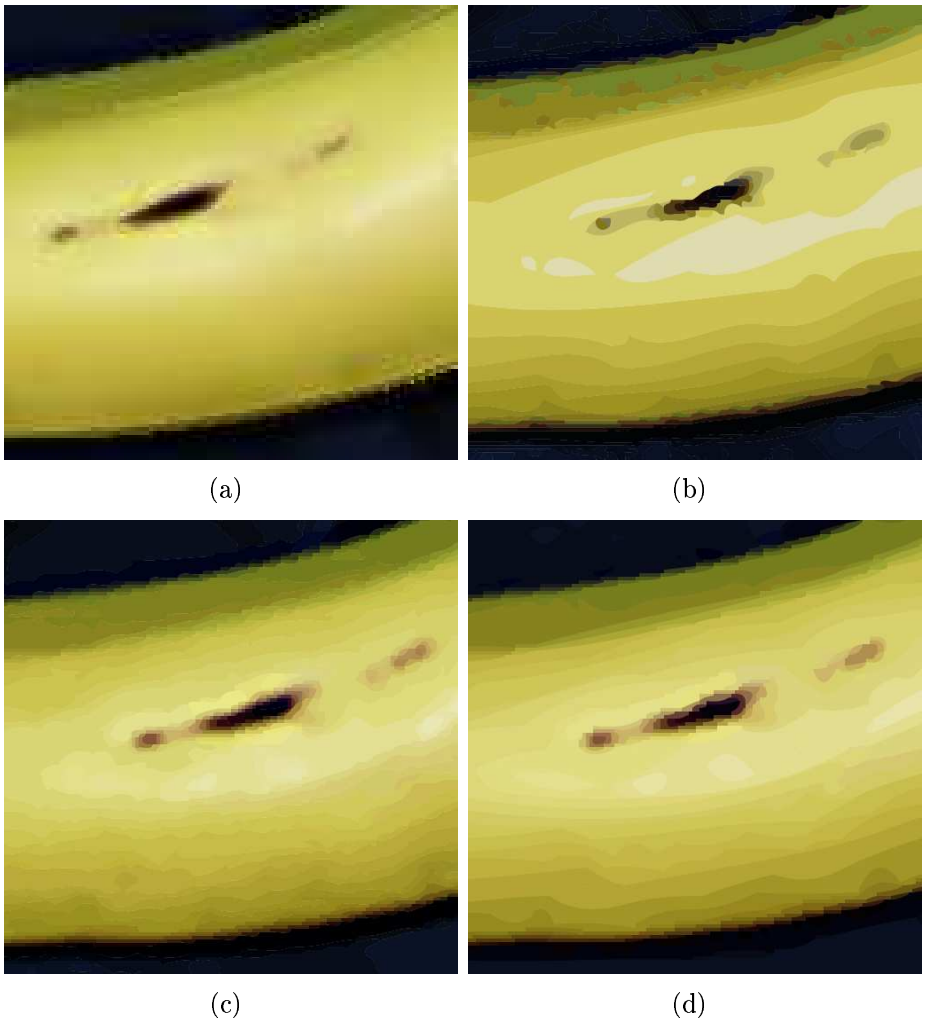


Figure 4.4: Comparison of vectorization techniques on the indicated portion of the image in Figure 2.1(a) using (a) Image Vectorization, (b) Adobe Illustrator, (c) Vector Eye, and (d) Macromedia Flash.

OBIE-style tools on the objects. The hierarchy automatically propagates these edits to the children objects. Similar edits can be performed on graphics produced by vectorization techniques using warping or other such tools. However, since the image is comprised of many small subobjects, holes can appear as object boundaries warp independent of one another (Figure 4.6(b)). Adobe Illustrators Live Paint tool does

(a) (b)

Figure 4.5: Translation of the label on the mit using (a) Image Vectorization and (b) Adobe Illustrator. Only one selection (the label) is required in (a), and the label subobjects are moved with the parent. The mit graphic extends under the label, leaving no hole. For (b), all 163 objects comprising the label must be selected. Also, a hole is left in the object.

allow such operations to occur without creating holes (Figure 4.6(c)), but at a large time cost. For the edit in Figure 4.6(c), each stroke with the warp brush required 20-40 seconds preprocessing/postprocessing time on top of the user editing time. Consequently, it required about 100 seconds to complete this very simple edit, while our edit in Figure 4.6(a) required only about 5 seconds.

Figures 4.7-4.9 show examples of objects which have been vectorized by Image Vectorization, Adobe Illustrator, Siame Vector Eye, and Macromedia Flash. Figure 4.10 compares the error from the original image over these and other examples. In this comparision, the number of Bezier patches for our technique is kept relatively equal to the number of regions for other techniques. Given equal number of regions, our method is comparable to other techniques, having less regions in some instances and more in others. However, note that the complexity of the regions in other techniques can vary wildly, from being a small region whose border is defined by only a few

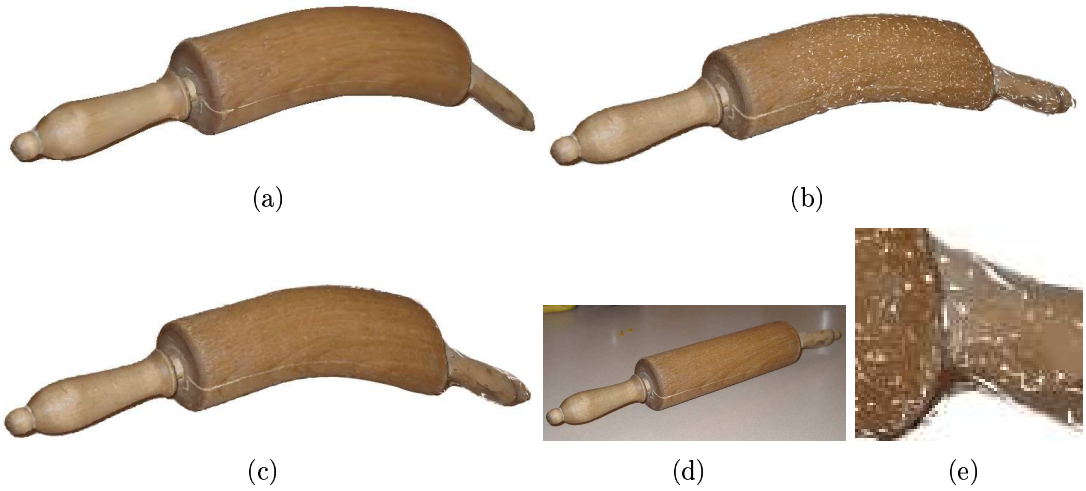


Figure 4.6: Rolling pin edited with (a) Image Vectorization and (b) Illustrator without Live Paint enabled and (c) with Live Paint enabled. (d) The original image. (e) Close up of (b) showing holes created by editing. The edit in (a) required 5 seconds to perform while the edit in (c) required 100 seconds.

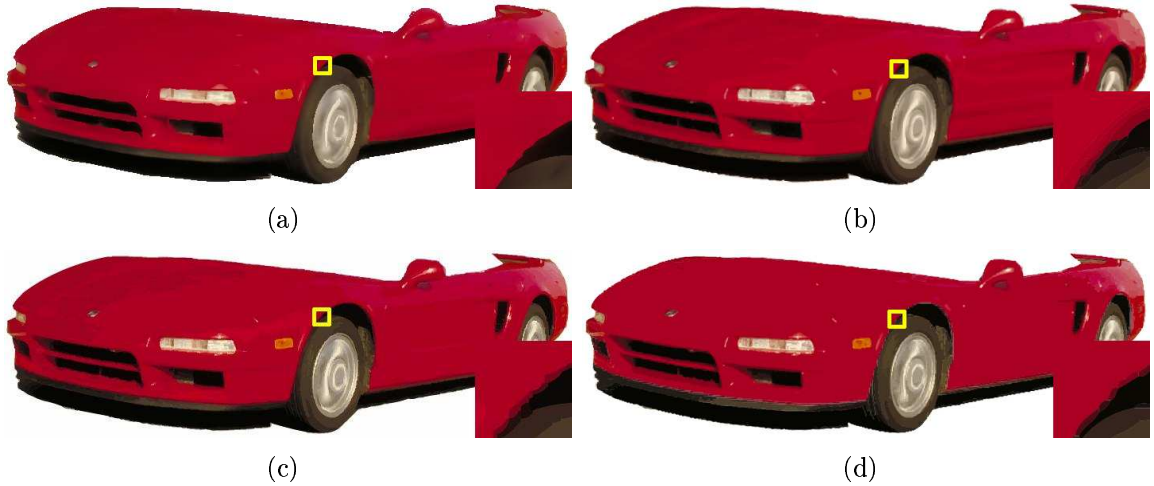
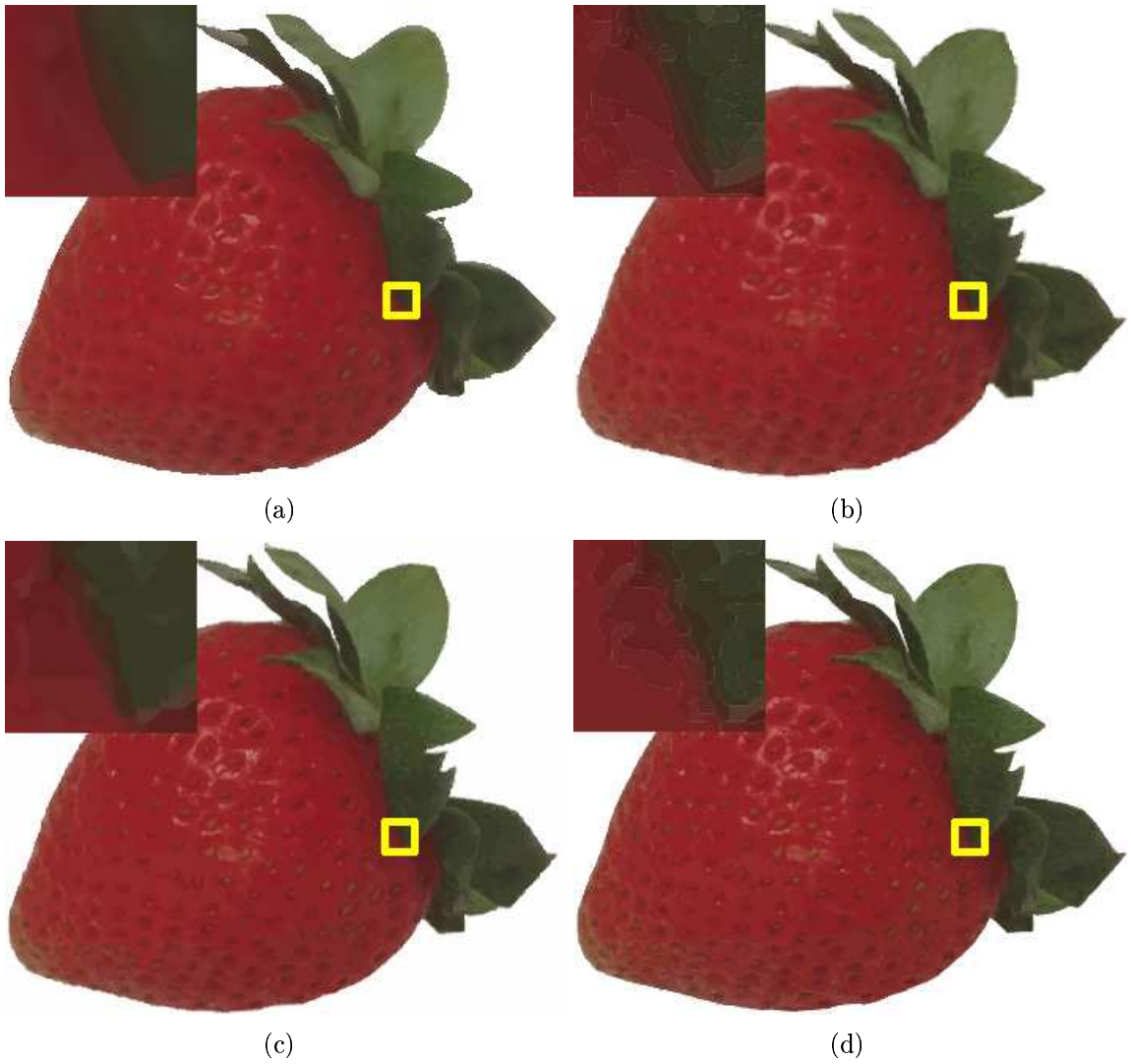


Figure 4.7: Ferrari graphic created by (a) Image Vectorization, (b) Adobe Illustrator, (c) Macromedia Flash, and (d) Siame Vector Eye. The area in the yellow box is shown zoomed to 500% in the lower right corner. Note the smooth-shaded appearance of the Image Vectorization zoom compared to the terraced look of the other techniques.

points to being a large, complex region whose border is defined by hundreds of points (Figure 2.2). Also note that for editing purposes, each region in other techniques



(a) (b)

(c) (d)

Figure 4.8: Strawberry graphic created by (a) Image Vectorization, (b) Adobe Illustrator, (c) Macromedia Flash, and (d) Siame Vector Eye. The area in the yellow box is shown zoomed to 500% in the upper left corner. Note the smooth-shaded look of Image Vectorization compared to the terraced look of the other techniques.

is treated as a separate object, while in our technique the patches are contained in objects, and each example has a few to a few tens of objects. Additionally, although the average error as reported would be virtually unperceivable for a user were it distributed evenly over the whole image, the error is often more concentrated in specific

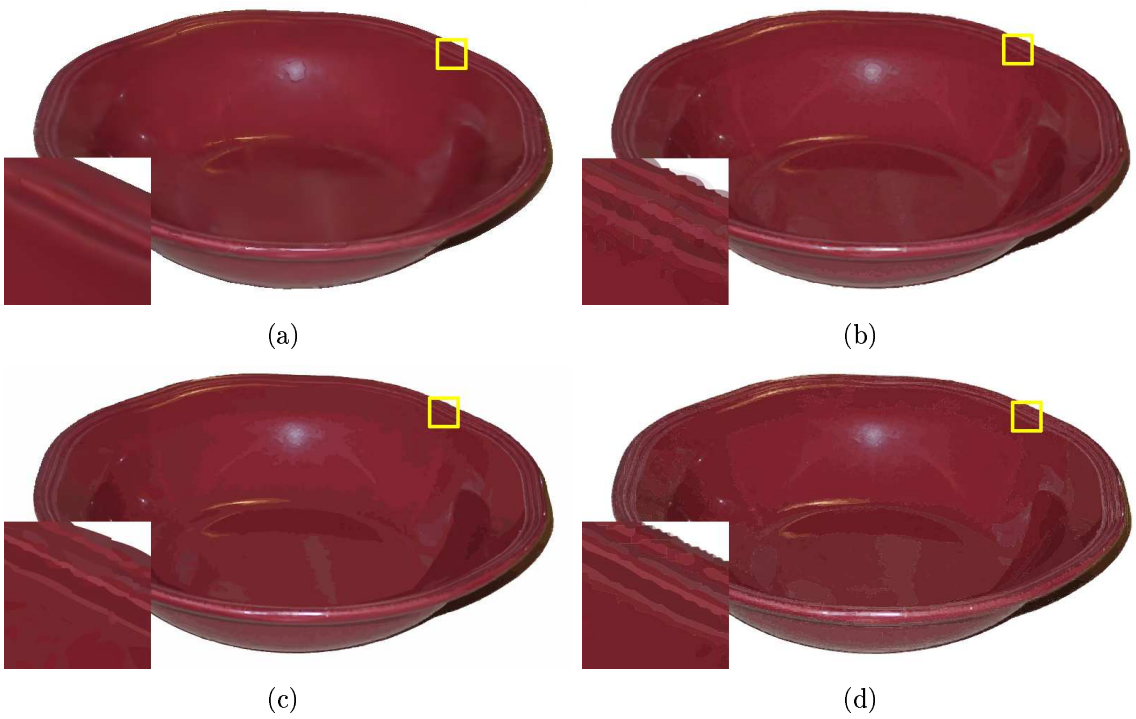


Figure 4.9: Bowl graphic created by (a) Image Vectorization, (b) Adobe Illustrator, (c) Macromedia Flash, and (d) Siame Vector Eye. The area in the yellow box is shown zoomed to 500% in the lower left corner. Note the smooth-shaded appearance of the Image Vectorization zoom compared to the terraced look of the other techniques.

	IV		Illustrator		Vector Eye		Flash	
Image	Regions	Error	Regions	Error	Regions	Error	Regions	Error
Ferrari	1714	4.67	1768	3.26	1735	5.54	1711	3.98
Banana	2100	2.32	2090	4.97	2099	4.74	2155	2.67
Strawberry	3018	3.7	3030	2.48	3021	2.33	2914	3.17
Bowl	2842	2.77	2837	2.85	2896	3.83	2919	2.93
Pin	3663	2.38	3732	1.58	3683	3.47	3745	2.06

Figure 4.10: The error of various vectorization techniques is compared when the number of patches or regions is relatively constant. The original images are shown in Figures 1.2(a), 3.12(a), 3.6(a), 4.2(a), and 4.6(d) respectively.

areas, leading to visual errors or discontinuities such as the terraced appearance of smoothly shaded regions seen in other techniques.

4.2 Comparison to Hand-Made Examples

Image vectorization can be used to reverse engineer manually-crafted vector graphics, as demonstrated in Figures 4.11 and 4.12. For each of the simple objects (leaves and stems) in Figure 4.11, selection is instantaneous, while recursive segmentation, mesh creation and rendering required less than 10 seconds per object. The original graphic consisted of 312 patches, while our graphic consisted of 695 patches. The error between our graphic and the original is about 3.3 per pixel. However, most of the error is around the edges of the leaves and stem where they do not completely overlap due to the smoothing of the object boundary.

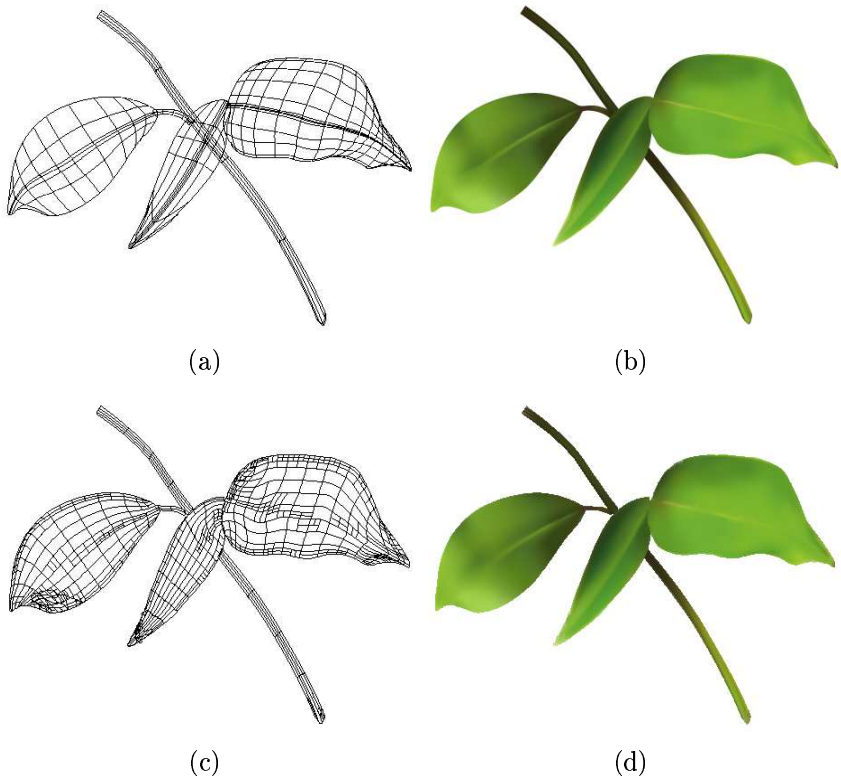
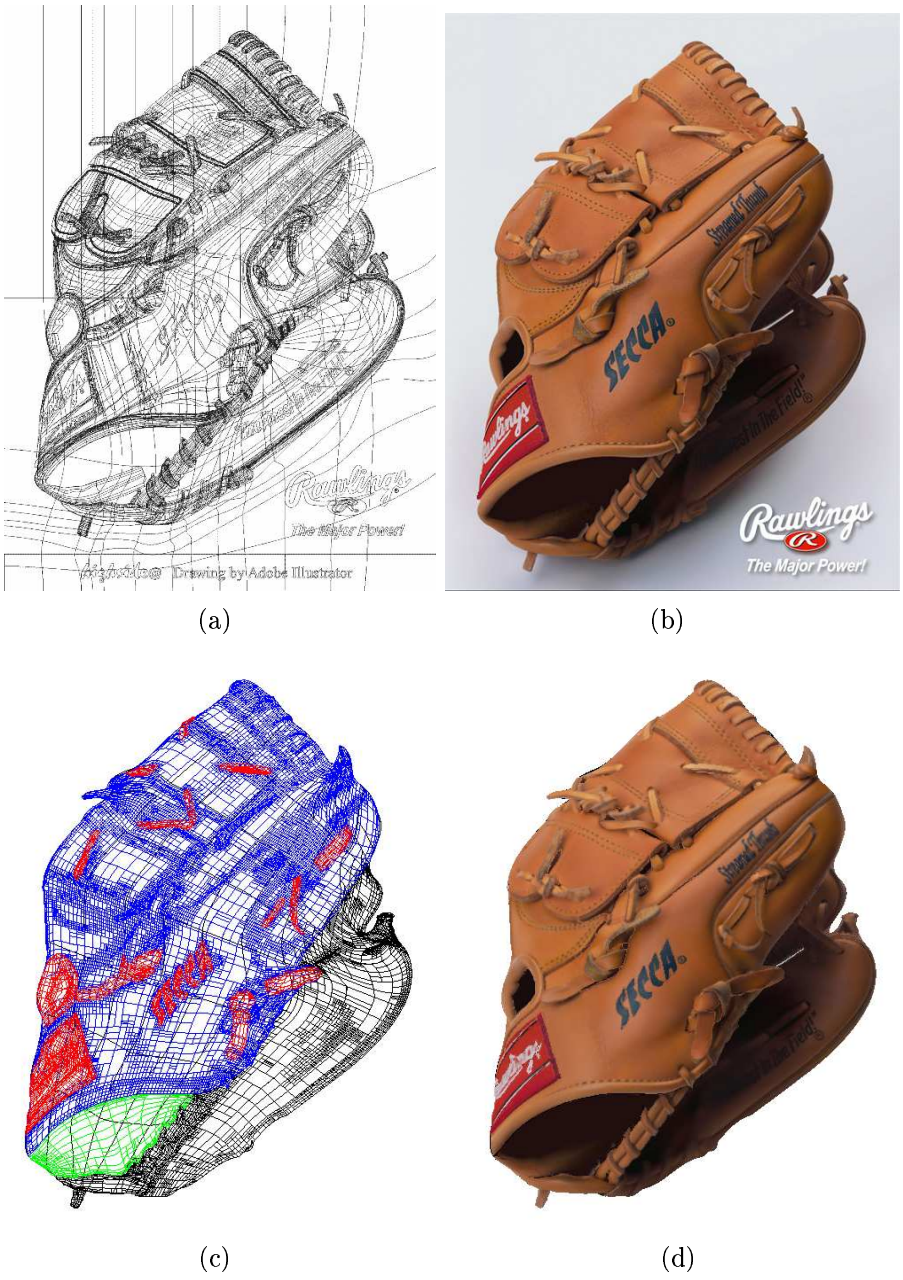


Figure 4.11: (a, b) Mesh created and rendered with Adobe Illustrator, 5 objects, 312 patches. (c, d) Fig. b as 462x377 raster image reverse engineered with image vectorization. 12 objects, 695 patches.

The vector graphic in Figure 4.12(a) required approximately 60 hours of a highly skilled artists time to create and contains several hundred gradient mesh lines. Image vectorization of Figure 4.12(b), taken as a raster image, required a total of about 4 minutes: Initial object selection (∼1-3 seconds per piece); pre-selection of each of the (green) subobjects (∼1-2 minutes); recursive graph cut segmentation (∼40 seconds for the large piece, 10-15 seconds for each of the two smaller pieces); mesh creation and rendering (∼1 minute). Our rendering has an error of about 3.7 per pixel when compared to the original glove.



(a) (b)

(c) (d)

Figure 4.12: (a, b) Mesh created and rendered by Highside [32]. (c, d) Mesh and rendering with object-based vectorization using (a) as a raster image. Blue and green indicates subobjects pre-selected from the black region, and red indicates subobjects of the blue region. 102 objects, 15796 patches.

4.3 Editing Results

Image vectorization provides a hierarchical arrangement of objects and their subobjects, allowing for complex image edits to be performed easily. Figures 4.13 illustrates examples of such edits in a series showing a baseball tearing through baseball glove. Figure 4.13(a) shows object layering by placing the ball between different subobject layers of the glove. Edits, such as the warps in Figure 4.13(b), are performed on the main object and are automatically applied to the subobjects (ties on the glove). The ability to edit subobjects independent of the main object as well as hole filling behind subobjects is shown where the label and letters on the glove are knocked off. The baseball tearing out of the back of the glove (Figure 4.13(c)) would be extremely difficult to mimic using other image editing or vectorization techniques. However, due to the object-level representation of image vectorization, the edit may be performed easily by a user in less than a minute.

(a) (b) (c)

Figure 4.13: Catching a "hot one." Image editing simulates label and letter being knocked off and ball tearing through glove. The glove is generated by vectorizing Figure 4.12(b).

The example in Figure 4.14 illustrates the power of hierarchical object control. Each face has two subobjects, the eyes, which have as subsequent descendent subobjects, the irises, then pupils, and finally the highlights in the pupils, forming five hierarchical layers of relevant objects. The left eyes were switched by simply moving the entire eye along with its subobjects. The right eyes, however, were edited by swapping the irises (without its subobjects) and the highlights. In other words, each woman in her right eye has her own sclera (white part of the eye) and pupil, but has the other woman's pupil and highlight.

(a) (b)

Figure 4.14: The eyes in (a) are switched in (b). Left eye was switched as a whole, right eye by switching only the colored iris and the highlight on the pupil.

A sequence of edits is illustrated in Figure 4.15. Figure 4.15(a) shows the unedited graphic. In Figure 4.15(b) the petal is scaled much too large, so it is rescaled and moved into place in Figure 4.15(c). In Figure 4.15(d) a leaf is translated, and then edited using the OBIE-style bend-stretch tool so that the thin connection to the stem is shortened, and then is moved into place higher up the stem in Figure 4.15(e). A

second leaf is moved higher up the stem in Figure 4.15(f). The broken stem is repaired by stretching the bottom half until it touches the upper half in Figure 4.15(g). In Figure 4.15(h) the last leaf is translated into position. Figure 4.15(i) shows the completed rose.

Additional examples of image editing, highlighting the editability, hierarchical object structure, and hole filling, can be seen in Figures 4.16-4.18. Other editing examples have been seen previously in Figures 1.2(c), 3.21(c), 3.23(b), 4.1(f), 4.5(a), and 4.6(a).

4.4 Zooming Results

Vector graphics are inherently scalable, suggesting the application of image vectorization for image zooming and resampling. Vectorizations by nature stylize the image to some degree, removing fine textures and approximating image boundaries, thus making our image vectorization not applicable in current form to general image resampling problems. However, our ability to produce smooth surfaces and capture sharp boundaries demonstrate potential use of image vectorization for handling specific image scaling cases. Results of such scales are shown in Figure 4.19.

Visual comparisons of our scaling results to other vectorization techniques are shown in Figures 4.7-4.9. As mentioned previously, our techniques scales objects smoothly, as opposed to other vectorization techniques which result is a striated or terraced look when smooth surfaces are zoomed.

4.5 Levels of Detail Results

Mesh refinement allows a graphic to be accurate to the original object within an error threshold. Since the grid refinement is constructed hierarchically, the same graphic can be used to display various levels of detail by altering how many levels

of the hierarchy are being displayed. Figures 4.20, 4.22, and 4.24 illustrates graphics and their rendering at different levels of detail. Figures 4.21, 4.23, and 4.25 show how the error diminishes as the number of patches increases. The error asymptotes depending on the level of error set by the user. Figure 4.26 shows all levels of detail for the rose from Figure 4.1(a).

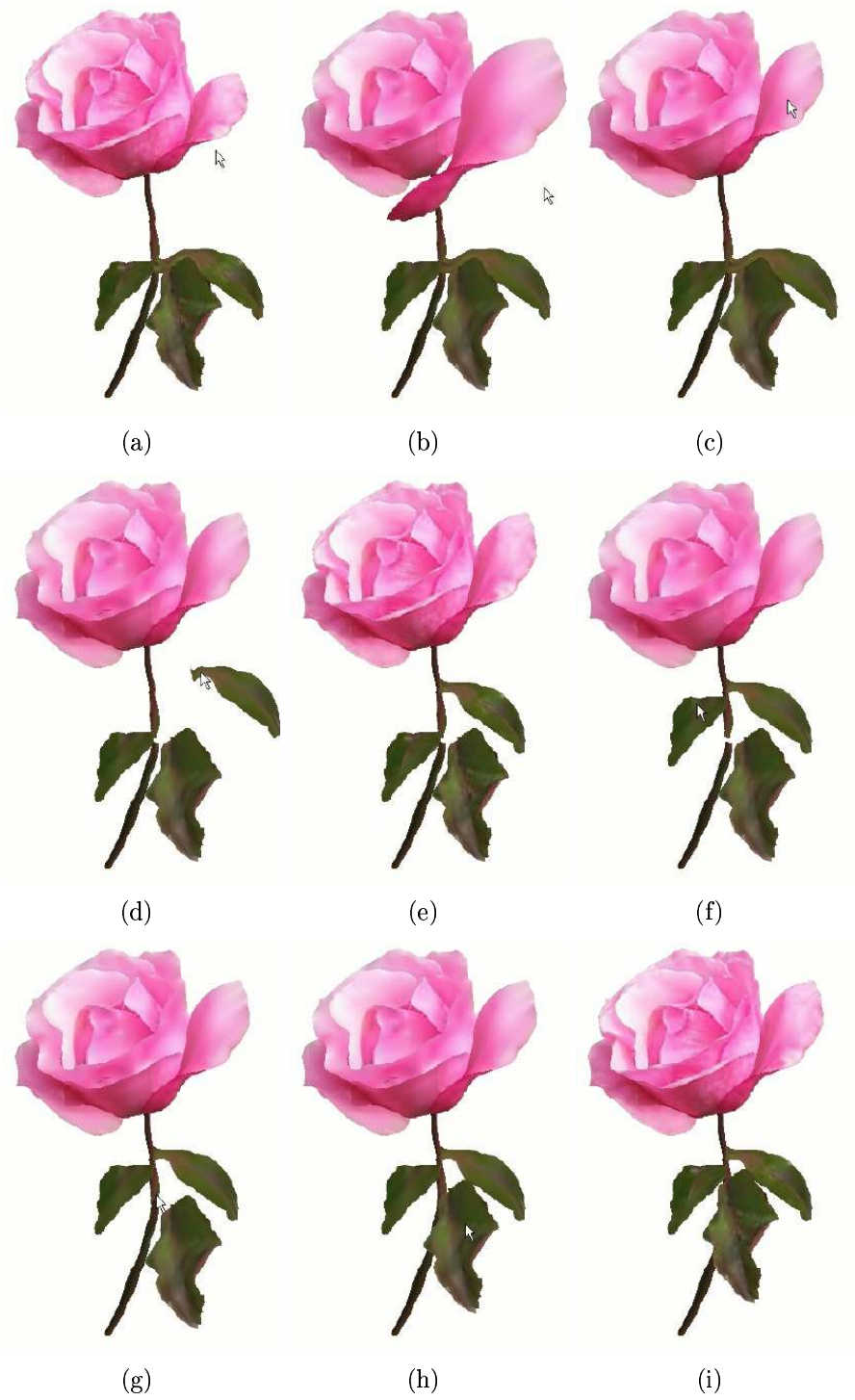


Figure 4.15: A sequence of edits on the rose is shown. (a) Original. (b) Petal is scaled. (c) Petal is moved. (d) Right leaf is warped. (e) Right leaf is moved onto stem. (f) Left leaf is moved higher up the stem. (g) Stem is warped to cover hole. (h) Center leaf is moved up the stem. (i) Final image. Some of the images were captured mid-edit, and so are rendered at a lower level of resolution.

(a) (b) (c) (d)

Figure 4.16: The words represented on the chalkboard as vector graphics are erased. Original image in Figure 4.3(a).

(a) (b)

Figure 4.17: The pyramid in (a) is shrunk and the sky filled in through background completion(b).

(a) (b)

Figure 4.18: "Desktop Editing." Items from the computer desktop and the real desk top from (a) are swapped in (b).

(a) (b) (c) (d)

Figure 4.19: (a) The original has been (b) downsampled to 25%, then scaled back up using (c) bicubic interpolation and (d) Image Vectorization. (c) has an average error of 1.58 per pixel while (d) has an average error of 4.34 per pixel.

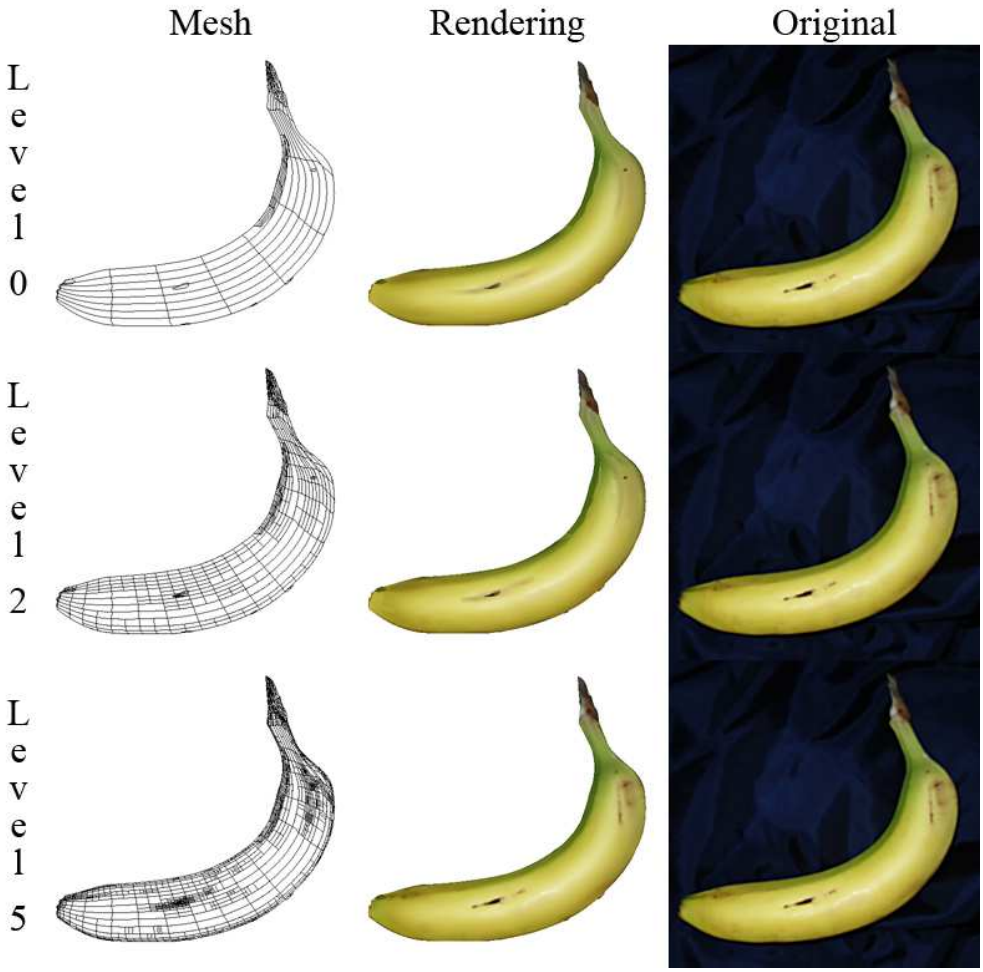


Figure 4.20: The grid and graphic of the banana shown at various levels of detail.

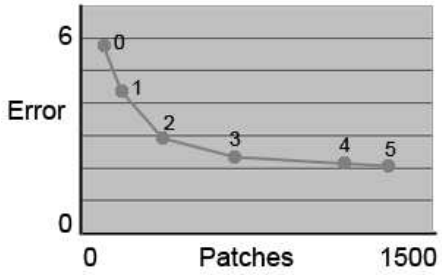


Figure 4.21: The error per pixel per RGB channel in Figure 4.20 given the number of patches. The number next to each data point indicates the iteration number which produced the given data.

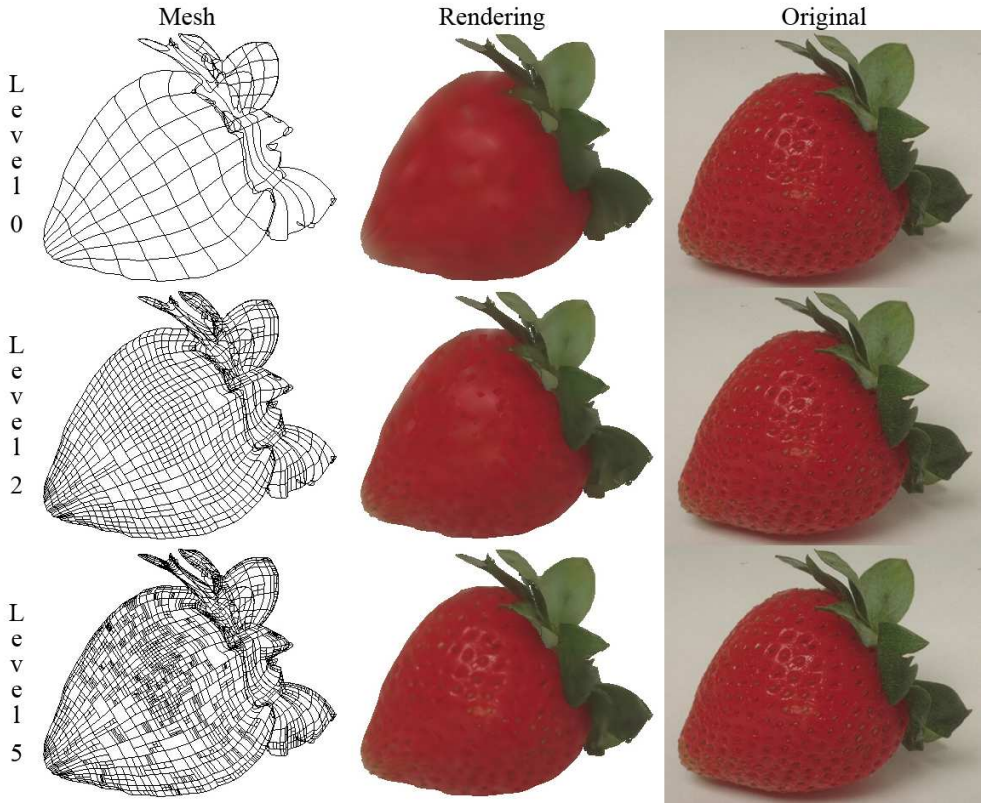


Figure 4.22: The grid and graphic of the strawberry shown at various levels of detail.

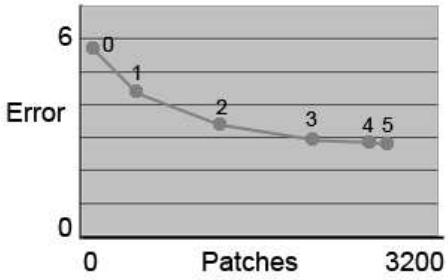


Figure 4.23: The error per pixel per RGB channel in Figure 4.22 given the number of patches. The number next to each data point indicates the iteration number which produced the given data.

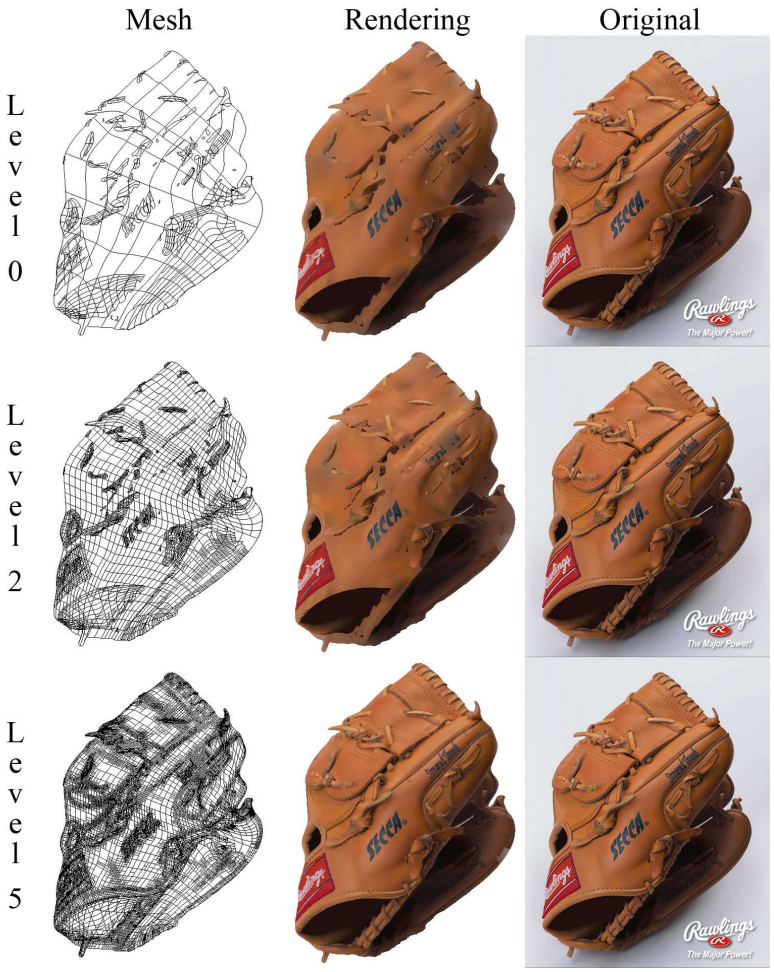


Figure 4.24: The grid and graphic of the baseball glove shown at various levels of detail.

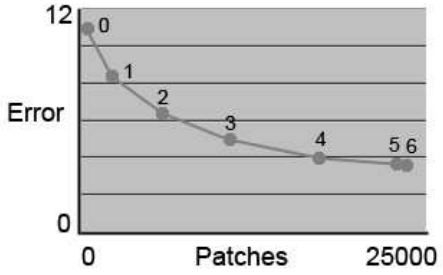


Figure 4.25: The error per pixel per RGB channel in Figure 4.24 given the number of patches. The number next to each data point indicates the iteration number which produced the given data.

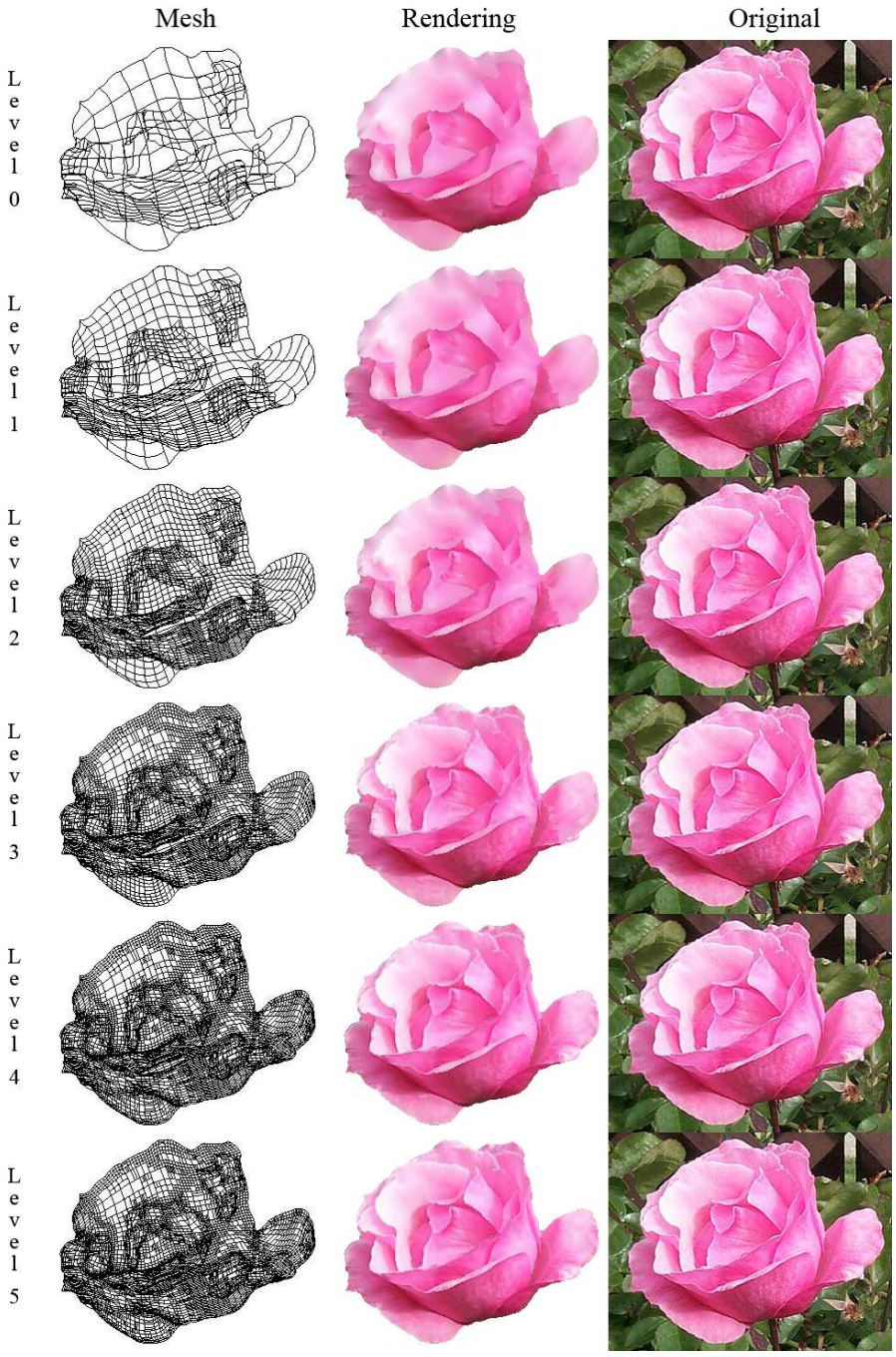


Figure 4.26: The grid and graphic of the rose shown at various levels of detail.

Chapter 5

Limitations and Future Work

One challenge faced by this system is its ability to handle complex textures. Most natural scenes contain textures that are not easily modeled using a smooth-shaded vector approach. These textures pose a problem both for our automated subobject selection, where textures make it difficult to find true edges and thus lead to incorrect segmentations, and for representation, where many small subobjects or excessive grid refinement are required to accurately model them. Complex image textures can lead to vectorizations heavier than the original pixel grid. However, such textures also pose similar problems to humans modeling the same objects by hand. Indeed, generally vector graphics appear smoothly shaded because such textures are noticeably absent. Their presence indicates either excessively heavy vector meshes or texture mapping. Since this work specifically focuses on creating shaded graphics without texture mapping, our results are similar to those which would be produced manually by graphic artists.

Nevertheless, to address these problems, additional work could focus on allowing

the system to better handle textures. This would include both better segmentation of textured regions and better representations of them. The ability to locate and distinguish different textures of an image would facilitate correct segmentation. Improved representation could be achieved by procedurally generating the desired textures. Procedural textures would allow for image zooming without the artifacts seen in texture maps, and would not require the storage of the original image. Another possible approach to improve texture representation is using the built-in error metric to identify when a particular patch is being subdivided too much, causing the mesh to become too heavy, and then reverting to an OBIE-style texture map to represent that patch.

Our algorithm also has difficulty with complex object shapes. We assume relatively convex shapes, and although our algorithm works quite well on many shapes with concavities, even multiple concavities, many images contain objects with shapes too complex to accurately represent. These objects must either be subdivided into more managable shapes, or our approach must be modified to handle boundaries of arbitrary complexity. Additionally, we have difficulty representing extremely thin (pixel-wide) objects, due to our dependence on the pixel grid to calculate the graphic mesh. Future work could focus on better representing such objects.

Additional work could focus on improving the automatic segmentation algorithm. Not only may textured areas hamper the segmentation, but also large shifts in object shading and shadowing may confuse the algorithm, such as those often found near the boundary of curved objects as the surface normals of the object fall into the image plane. Superfluous subobjects are also occasionally found. Improvements in the subobject selection would not only improve the vectorization algorithm, but could also

be used to provide object-level information to a variety of computer vision algorithms.

Image vectorization could be applied to image resampling problems in order to produce more accurate or visually-appealing image zooms. Our ability to capture hard edges and scale smooth surfaces could potentially improve the scaling of such features.

A logical extension of image vectorization is its application to video. Video vectorization is the process of vectorizing a particular object or the entire scene in a video sequence. Stylizing video is an active field of research, due to its applicability in advertising and film making. Video vectorization would not only allow for efficient video segmentation, but would also allow for hierarchical object-based editing to be applied to video sequences.

There is a significant difference between vectorizing an image and vectorizing a video sequence. An object must be segmented in each frame of the video sequence. Objects must "look right" in every frame of the sequence, meaning there are no visual discontinuities such as flickering or color changes, and thus the vectorization in each frame must be coordinated. The system must deal with objects that disappear and reappear through the sequence. Also an object which may look like a subobject in a single frame may be revealed to be an occluding object in a video sequence and so must be hierarchically arranged accordingly. Object editing occurs differently in that edits must be intelligently propagated through previous and/or subsequent frames as desired by the user. Each of these is a non-trivial problem requiring additional work.

Vector-based video generation techniques are another avenue of future research. Commercial applications such as Macromedia Flash are specifically designed to allow users to manually create and edit vector graphics in order to create animation se-

quences. Image vectorization allows these techniques to be applied to objects derived from photographs as well. It also opens the door to the development of more complex and powerful editing tools, allowing for the easy generation of video sequences which are difficult to create today.

Chapter 6

Conclusions

We have presented a method for creating editable vector graphics from objects in raster images. As demonstrated through the examples, object-based image vectorization works surprisingly well on a wide variety of naturally occurring and manmade objects, providing user-selectable levels of detail and rendering, ranging from photo-realistic to highly stylized. We have also demonstrated tools for editing and animating selected image objects in ways that are not possible with current pixel-based image editing tools or without resorting to manual creation of the vector graphic.

Our approach to image vectorization does not perform well on highly textured images such as trees or grass where object detail is often the size of a pixel or less. However, such cases can be detected from the error metric used in mesh refinement. In such cases we revert to Object-Based Image Editing [8] which uses texture mapping implemented on an editable, triangular network.

Object background completion facilitates automated segmentation of subobjects and allows objects to be edited and manipulated without leaving holes. However, we

believe that Image Completion with Structure Propagation [43] would be a powerful addition to this part of the work. In general, we find that automatic background initialization, coupled with 1-step watershed-to-pixel boundary localization, provides an efficient interface for object selection, while automated, recursive graph cut segmentation of subobjects makes complete vectorization of complex objects practical.

Appendix A

User Manual

A.1 Introduction

Image Vectorization (IV) is a software tool for converting raster images into vector graphics. The main steps in creating a vector graphic are:

1. Selecting an object from the image to vectorize
2. Creating a mesh for the object
3. Render the graphic
4. (Optional) Edit the graphic

This manual will explain how to perform each of these operations. The icons for the different tools are shown in Figure A.1.

A.2 Getting Started

A.2.1 Opening an Image

To open an image, go to the File menu, and choose Open. IV currently supports jpeg, png, and ppm files (and maybe some others).

(a) (b) (c) (d) (e) (f) (g) (h) (i)

Figure A.1: Image Vectorization icons for (a) Trap Select tool, (b) Graph Cut tool, (c) Mesh tool, (d) Move Pivot tool, (e) Scale tool, (f) Rotate tool, (g) Stretch tool, and (h) Bend-Stretch tool.

A.2.2 Saving an Image

To save an image, go to the File menu, and choose either Save or Save As. Save will save the image over current image, and Save As will let you choose a new name for the image. It can be saved as a jpeg, png, or ppm file. The image will be saved as it looks on the screen.

A.3 Object Selection

IV allows object-based vectorization, so accordingly objects must be selected before vectorization is possible. We use a selection tool based on min graph cut to allow for object segmentation. IV also contains an older tool, the trap select tool.

A.3.1 Trap Select Tool

The purpose of the Trap Select tool is to allow the user to select small regions called TRAPS and join them into objects, which can then be manipulated.

To start the Trap Select Tool, click the icon shown in Figure A.1(a). TRAP boundaries should then appear on the image. A sidebar with buttons specific to this tool should also appear to the right side of the main window.

To select a TRAP, either click on the trap or drag a box over it. If you drag the mouse an invisible bounding box is created, and any TRAP that is partially or completely inside the bounding box will be selected. Selected regions turn a blue-gray color. To deselect a TRAP, click on it again. You can also deselect all the TRAPS

by clicking on the clear button in the sidebar.

If the TRAP boundaries do not correspond to the object you want to select, you can move down in the TRAP hierarchy by clicking on the up or down arrows in the sidebar. TRAPS selected in a higher level of the hierarchy should be selected in lower levels, but TRAPS selected in lower levels will not be selected in higher levels. You can select any level of the hierarchy you want to work on. A useful strategy is to select as much of the object as you can on higher levels of the hierarchy, then to move to lower levels to pick up any additional pieces.

When the desired region is selected, click the "create object" button to create an object.

A.3.2 Min Graph Cut Tool

To activate the min graph cut tool, click the icon shown in Figure A.1(b). To select an object using min graph cut, simply drag over the object you desire. Seed points are then drawn on the object in yellow for foreground and blue for background. With each mouse movement the segmentation is recomputed. The object is highlighted in yellow and the background in blue.

To place foreground seeds, hold down the left mouse button while dragging. To place background seeds, hold down the right mouse button. To erase previously placed seeds, hold down the middle button. Alternately, the sidebar contains radio buttons which may be used to change the function of the left mouse button in selection.

The size of the paintbrush may be changed by clicking on the radio buttons on the sidebar. A large and small size are available.

The selection may be cleared by clicking the "Clear" button.

A.3.3 Creating Objects and Subobjects

When the desired region has been selected, click on the "Make Object" button to finish the selection of the object. This object will show up in the Objects tab of the Objects, Graphics, Hierarchy sidebar under a generic name.

You may also select subobjects of the main object that has been selected. To do this, select the subobject in the same way that you would select a main object. Then, click the "Make Subobject" button instead of the "Make Object" button.

A.4 Graphic Creation

Once an object is selected, it must be converted into a graphic. This consists of three main steps: selecting corners for the graphic mesh, creating the mesh, then creating and rendering the mesh. To begin Graphic Creation, click the icon shown in Figure A.1(c).

A.4.1 Choose Object

To choose which object to vectorize, go to the "Objects" tab in the Objects, Graphics, Hierarchy sidebar. Click on the name of the object you wish to vectorize.

A.4.2 Selecting Corners

To select the corners, simply click on the "Select Corners" button. To edit the corners, click on the "Edit corners" button. The button will remain down until you have finished editing the corners. To move a corner, click near the desired corner on the image, then move the mouse to where you wish the corner to be. The corner will remain on the object boundary. When finished, click the "Edit Corners" button to end.

You may turn off the display of the corners by clicking the "Show Corners" checkbox.

A.4.3 Making Mesh

To make the mesh, click on the "Build Axis" button. You may turn off the display of the mesh by clicking on the "Show Mesh" checkbox. This not only creates the mesh but generate a rough rendering of that mesh.

A.4.4 Render Mesh

To finish the graphic creation, click on the button "Subobject Extraction". This will create subobjects and refine the mesh as needed to create a rendering of the graphic.

To adjust the look of the graphic, you may change the connected component threshold, the edge accuracy threshold, the subobject accuracy and error thresholds. The "CC Threshold" is used to compute the initial connected component which determines the object's main color. The "Edge Accuracy" threshold sets how accurate the vector graphic boundary must adhere to the real object boundary. The "Subobject Accuracy" slider sets a threshold which is used to decide if a certain pixel is near enough to the main object color to be considered part of that object, or whether it should be used as a seed to select a subobject. The "Error Threshold" slider sets a threshold to decide whether a particular patch has an error high enough to require refinement. The Error Threshold spinbox determines the maximum number of refinements allowed.

You may also set the background color of an object rather than have the algorithm compute it by clicking the "Set Background" button, clicking on the image within the desired color, and then clicking the "Set Background" button again to deselect it.

An error map may be computed by clicking the "Compute Error" button.

A.5 Managing Graphics and Hierarchy

When a graphic is created, many subobject graphics are created as well. These form a hierarchy which can be seen in the "Hierarchy" tab of the Objects, Graphics, Hierarchy sidebar. Each main object will appear as with a plus next to it if it contains subobjects. By clicking on the plus, the subobjects can be shown. These subobjects can in turn have subobjects. The "Show Grid" checkbox allows you to see the grid for a given subobject.

To select an object or subobject, click on its name in the Hierarchy menu. This will be the object to which editing operations will be applied.

The graphics tab also lists the graphic objects, but does so linearly as opposed to in a hierarchy. On this tab, you may also save or load graphics. When saving a graphic, the select graphic and all its subobjects will be saved.

A.6 Editing Tools

Once an object is selected, there are multiple tools which can be used on it to change the look of the object. The following is a list of these tools and how they work. These tools based on those developed by Alan Cheney in [14] and [8].

A.6.1 Move Pivot Tool

The Move Pivot tool is an auxiliary tool that helps the other tools to perform. Many of the other tools require a pivot point about which they will do their work. The Move Pivot tool allows you to change the location of the pivot point.

To start the Move Pivot tool, click the icon shown in Figure A.1(d). A light blue dot should appear in the center of the selected object. To move the pivot point, simply click on the object where you would like the new point to be. To place the pivot point back it its original location, click on the Reset Pivot button in the Move

Pivot Tool sidebar located on the right side of the main window.

A.6.2 Move Tool

The Move Tool allows you to move an object around in the image. The shape and orientation of the object remain the same, only the location changes.

To start the Move tool, click the icon shown in Figure A.1(e). To use the tool, simply click on the object and then drag the cursor around. The object will move around accordingly.

To apply the tool to the selected graphic without propagating the edit to the children, deselect the "Edit Children" checkbox.

A.6.3 Scale Tool

The Scale Tools allows you to scale the selected object. This tool does not require that the aspect ratio is maintained, so you can scale it in various directions. The scaling happens with respect to the pivot point.

To start the Scale Tool, click the icon shown in Figure A.1(f). To use the tool, click anywhere on or near the object and drag the cursor. The object will scale according to the pivot point. Moving the cursor in a direction along the line between the pivot point and the point where the mouse was first clicked will scale the object in that direction. Moving the object in a direction perpendicular to that will scale the object in that direction.

To apply the tool to the selected graphic without propagating the edit to the children, deselect the "Edit Children" checkbox.

A.6.4 Rotate Tool

The Rotate tool allows you to rotate the object.

To start the Rotate Tool, click the icon shown in Figure A.1(g). To use the tool, click on the object and drag it around. The object will rotate around the pivot point.

To apply the tool to the selected graphic without propagating the edit to the children, deselect the "Edit Children" checkbox.

A.6.5 Stretch Tool

The Stretch tool allows you to stretch a portion of the object relative to the pivot point. The object will only be stretched to the side of the pivot point where you originally clicked. It also allows for implicit control of the stretch. For example, you can control whether most of the stretch happens near the pivot point or near the end of the object, and control how wide it stretches along the length of the stretch direction.

To start the Stretch Tool, click the icon shown in Figure A.1(h). To use the tool, click on the object near the area you want to be stretched. Dragging the mouse along the line between the pivot point and the point where you originally clicked will stretch the object in that direction, and dragging the mouse perpendicular to that will stretch the object that way.

You can also implicitly affect the curve by changing the curves in the Stretch tool sidebar. Each curve has four control points which can be moved. The left side of the curve corresponds to the part of the object closest to the control point, while the right side of the curve corresponds to the part of the object farthest from the control point in the direction of the stretch. By moving the control points, you can change the curve and by so doing change the way that the object is stretched. By changing the Length curve, you change which areas of the object are stretched more. Initially, the curve is set so that the stretching occurs uniformly over the object. By changing the

curve, you can change whether the object is stretched more near the control point or near the border. Similarly, by changing the thickness curve, you can change whether the object is wider near the pivot point or the border. By clicking reset under either curve, you can reset the curve to its initial position.

You can also click the preserve area checkbox if you want the part of the object being stretched to have a constant area. This will cause the object to thin out as it is stretched.

To stretch a certain portion of the object, first move the pivot point so that all of the object you want to be stretched is to one side of the pivot.

To apply the tool to the selected graphic without propagating the edit to the children, deselect the "Edit Children" checkbox.

A.6.6 Bend-Stretch Tool

The Bend-Stretch tool is similar to the Stretch tool except that it can bend the object as well as stretch it. The bending and stretching occurs relative to the position of the control point, with only the parts of the object to one side of the control point being affected. Like with the Stretch tool, you can also implicitly control the change of the object.

To start the Bend-Stretch Tool, click the icon shown in Figure A.1(i). To use the tool, click on the object near the area you want to be changed. Dragging the mouse along the line between the pivot point and the point where you originally clicked will stretch the object out in that direction, and dragging the mouse perpendicular to that will bend the object in that direction.

You can also implicitly affect the curve by changing the curves in the Stretch tool sidebar. Each curve has four control points which can be moved. The left side of

the curve corresponds to the part of the object closest to the control point, while the right side of the curve corresponds to the part of the object farthest from the control point in the direction of the stretch. By moving the control points, you can change the curve and by so doing change the way that the object is stretched. By changing the Length curve, you change which areas of the object are stretched more. Initially, the curve is set so that the stretching occurs uniformly over the object. By changing the curve, you can change whether the object is stretched more near the control point or near the border. Similarly, by changing the rotation fallout curve, you can change whether most of the bending happens near the pivot point or the border. By clicking reset under either curve, you can reset the curve to its initial position.

To bend a certain portion of the object, first move the pivot point so that all of the object you want to be bent is to one side of the pivot.

To apply the tool to the selected graphic without propagating the edit to the children, deselect the "Edit Children" checkbox.

Appendix B

OpenGL Commands

The following OpenGL commands are used to render a patch.

For the arrays coords[i][j][k] and colors[i][j][k] where the i and j indices in both cases refer to the x and y index respectively into the 4x4 Bezier patch, the k index in coords refers to the x, y, or z position respectfully in the image (where z = 0), and the k index in color refers to the r, g, b, and alpha values respectfully in the image (where alpha = 1):

```
glEnable(GL_MAP2_VERTEX_3);
glEnable(GL_MAP2_COLOR_4);
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &coords[0][0][0]);
glMap2f(GL_MAP2_COLOR_4, 0, 1, 4, 4, 0, 1, 16, 4, &colors[0][0][0]);
glMapGrid2f(10, 0, 1, 10, 0, 1);
glEvalMesh2(GL_FILL, 0, 10, 0, 10);
glDisable(GL_MAP2_VERTEX_3);
glDisable(GL_MAP2_COLOR_4);
```

Bibliography

[1] S. Ablameyko, V. Bereishik, O. Frantskevich, M. Homenko, N. Paramonova, and O. Patsko. Automatic/interactive interpretation of color map images. In *Proc. 16th Intl. Conf. on Pattern Recognition*, pages 1269–1272, 2002.

[2] Adobe Systems Incorporated. Streamline at http://www.adobe/com/products/. 1997.

[3] Adobe Systems Incorporated. Adobe photoshop 7.0 user guide. 2002.

[4] Adobe Systems Incorporated. Illustrator at http://www.adobe/com/products/. 2005.

[5] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen. Interactive digital photomontage. In *Proceedings of ACM SIGGRAPH 2004*, pages 294–301, 2004.

[6] M Ashikhman. Synthesizing natural textures. In *ACM Symposium on Interactive 3D Graphics*, pages 217–226, 2001.

[7] AutoTrace. Autotrace at http://autotrace.sourceforge.net/. 2004.

[8] W. Barrett and A. Cheney. Object-based image editing. In *Proceedings of ACM SIGGRAPH 2002*, pages 777–784, August 2002.

[9] M. Bertalmio, V. Luminita, G. Sapiro, and s. Osher. Simultaneous structure and texture image inpainting. In *IEEE Trasactions on Image Processing*, volume 12, pages 417–424, August 2003.

[10] M. Bertalmio, G. Sapiro, L. Vese, and C. Ballester. Image inpainting. In *Proceedings of ACM SIGGRAPH 2002*, pages 882–889, 2002.

[11] F. L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. In *IEEE Transaction on PAMI*, pages 567–585, 1989.

[12] Y. Boykov and M.-P. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images. In *Proceedings of IEEE International Conference on Computer Vision*, pages 105–112, 2001.

[13] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. In *IEEE Transactions on PAMI*, volume 26, pages 1124–1137, September 2004.

[14] A. Cheney. Object-based image editing. In *Masters Thesis, Department of Computer Science, Brigham Young University, Provo, Utah.*, 2002.

[15] J. P. Collomosse and P. M. Hall. Painterly rendering using image salience. In *Eurographics UK Conference, Proceedings*, pages 122–128, 2002.

[16] Corel Corporation. CorelTRACE at http://www.corel.com. 2005.

[17] A. Criminisi, P. Perez, and K. Toyama. Object removal by exemplar-based inpainting. In *Proceedings of IEEE CVPR 2003*, pages 721–728, 2003.

[18] Doug DeCarlo and Anthony Santella. Stylization and abstraction of photographs. In *Proceedings of ACM SIGGRAPH 2002*, pages 769–776, August 2002.

[19] Y. Deng, B. S. Manjunath, and H. Shin. Color image segmentation. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR '99,*, volume 2, pages 446–451, 1999.

[20] I. Drori, D. Cohen-Or, and H Yeshurun. Fragment-based image completion. In *Proceedings of ACM SIGGRAPH 2003*, pages 303–312, 2003.

[21] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001*, pages 341–346, 2001.

[22] A. A. Efros and T. Leung. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, September 1999.

[23] J. H. Elder and R. M. Goldberg. Image editing in the contour domain. In *Computer Vision and Pattern Recognition, Proceedings*, page 374381, June 1998.

[24] J. Fan and D. Yau. Automatic image segmentation by integrating color-edge extraction and seeded region growing. In *IEEE Transactions on Image Processing*, volume 10, pages 1454–1466, October 2001.

[25] H. Gao, Siu W.-C., and Hou C.-H. Improved techniques for automatic image segmentation. In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 11, pages 1273–1280, December 2001.

[26] P. Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of ACM SIGGRAPH 1998*, pages 207–214, 1990.

[27] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of ACM SIGGRAPH 1998*, pages 453–460, 1998.

[28] A. Hertzmann. Paint by relaxation. In *Proc. Computer Graphics International 2001*, page 4754, 2001.

[29] A. Hertzmann, C.E. Jacobs, N. Oliver, B. Curless, and D. H. Salsesin. Image analogies. In *Proceedings of ACM SIGGRAPH 2001*, pages 327–340, August 2001.

[30] F. Jing, M. Li, H.-J. Zhang, and B. Zhang. Unsupervised image segmentation using local homogeneity analysis. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 456–459, 2003.

[31] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. In *Proceedings of IEEE International Conference on Computer Vision*, pages 259–268, 1987.

[32] Highside (Takashi Kondo). http://homepage3.nifty.com/highside/. 2004.

[33] V. Kwatra, A. Schodl, I. Essa, G Turk, and A. Bobick. Graphcut textures: Image and video synthesis using graph cuts. In *Proceedings of ACM SIGGRAPH 2003*, pages 277–286, 2003.

[34] W. Li, M. Agrawala, and D. Salesin. Interactive image-based exploded view diagrams. In *Proceedings of Graphics Interface*, pages 203–212, 2004.

[35] Y. Li, J. Sun, C.-K. Tang, and H.-Y. Shum. Lazy snapping. In *Proceedings of ACM SIGGRAPH 2004*, pages 303–308, 2004.

[36] Macromedia Incorporated. Flash at http://www.macromedia.com/software/flash. 2004.

[37] E. N. Mortensen and W. A. Barrett. Intelligent scissors for image composition. In *Proceedings of ACM SIGGRAPH 1995*, pages 191–198, 1995.

[38] E. N. Mortensen and W. A. Barrett. Toboggan-based intelligent scissors with a four parameter edge model. In *Proceedings of IEEE Conference on Computer vision and Pattern Recognition*, pages 452–458, 1999.

[39] B. M. Oh, M. Chen, J. Dorsey, and F. Durand. Image-based modeling and photo editing. In *Proceedings of ACM SIGGRAPH 2001*, pages 433–442, 2001.

[40] L.J. Reese. Intelligent paint: Region-based interactive image segmentation. In *Masters Thesis, Department of Computer Science, Brigham Young University, Provo, UT.*, 1999.

[41] C. Rother, V. Kolmogorov, and A. Blake. Grabcut - interactive foreground extraction using iterated graph cuts. In *Proceedings of ACM SIGGRAPH 2004*, pages 309–314, 2004.

[42] Siame Editions. Vector eye at http://www.siame.com/. 2005.

[43] J. Sun, L. Yuan, J. Jia, and H.-Y. Shum. Image completion with structure propatation. In *Proceedings of ACM SIGGRAPH 2005*, pages 861–868, 2005.

[44] The Gimp. http://www.gimp.org.

[45] L. Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH 2000*, pages 479–488, 2000.

[46] G. Wolberg. Image morphing: a survey. In *the Visual Computer 14*, pages 360–372, 1998.

[47] B.S. Yu, X. Morse and T.W. Sederberg. Image reconstruction using data-dependent triangulation. In *IEEE Computer Graphics and Applications*, volume 21, pages 62–68, 2001.

[48] J.J. Zou and H. Yan. Line image vectorization based on shape partitioning and merging. In *Proc. Intl. Conf. on Pattern Recognition*, 2000.