Getting the Picture

Andrew Glassner

Most of the time we work hard to create sharp and clear images. After all, we make our pictures to communicate our message as clearly as possible.

Pictures meant for utilitarian purposes should be simple and direct. For example, if you need to change a photocopier part, you want the repair diagram to be uncluttered and comprehensible. There are also pictures that have a more human purpose, perhaps to help tell a story. The intention of such images isn't to convey information, but share a sense of mood and feeling. There are also pictures that are almost all mood. These are abstract or impressionistic types of images.

We've seen computer graphics used for all of these purposes in recent years, and the styles continue to proliferate. From an early focus on a kind of photorealism, the kinds of pictures we create with computers have evolved to embrace a wide variety of styles, some of which seem to look like traditional media.

Creating images that are deliberately imprecise images provides another way to express visual ideas.

The technique of *successive approximation* helps us see a rough version of a picture quickly, with details filling in over time. Many rendering systems first create and display a low-resolution version of an image and then gradually add in details. If the user doesn't like the way the picture is shaping up, he or she can stop the program and fix things before trying again. This leads to faster turnaround time than waiting for the complete final image to render before seeing anything at all.

Successive approximation is also useful when we want to transmit models or images over low-bandwidth connections. Most browsers can show GIF images by first drawing only a small number of scan lines, replicating each line downward. New lines fit in between the old ones, slowly building up the image, as in Figure 1. Some programs that display 3D models use a similar technique, transmitting a low-resolution version of the model so that the person on the other end has something to look at immediately, and then sending more detail over time.

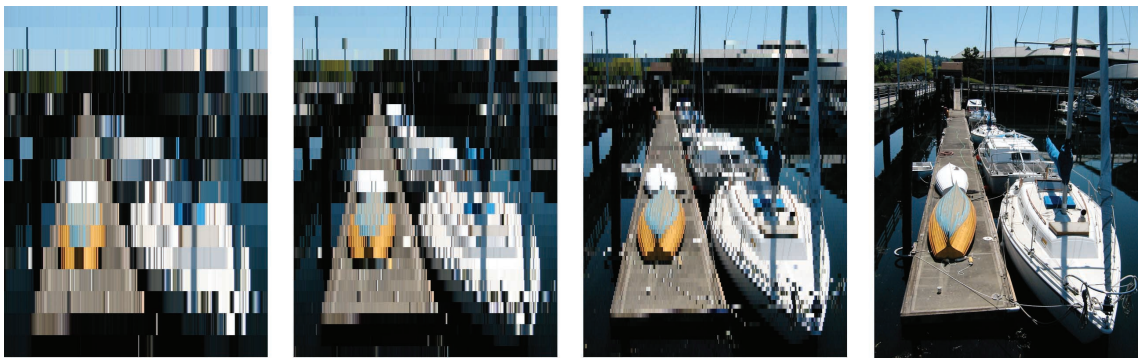
There are lots of interesting ways to make approximate images. Here I'll look at some ways to create approximate images using relaxation and optimization techniques.

Approximately right

We can use two general techniques for making approximate images: transformations and algorithms.

Transformations take an image into another representation, operate upon it there, and then turn it back into an image. The most popular transformations for this sort of thing are Fourier and wavelet analysis. Using Fourier methods we can, for example, take a picture, lop off the high frequencies, and then recreate a new image from what's left. The result is often an image that looks blurrier than the original. Wavelet transforms are similar. When the higher-order wavelets are removed, the reconstructed image can look a little blocky, like an image that has gone through JPEG compression.

Algorithmic techniques encompass just about every-



1 Four images demonstrating the window shade effect as a browser loads an image in progressive-GIF format. The scene is a boat dock at the Port of Edmonds, Washington.

2 Blurring and pixellation. (a) A street scene in Edmonds, Washington. (b) A blurred version. (c) A pixellated version.

thing else. Perhaps the most popular forms of creating approximate images algorithmically use *blur* and *pixellation*.

Blur is pretty simple—just take an image and smooth it. Every image-processing program has some form of blur filter built into it. Figure 2a shows an image, and Figure 2b shows a blurry version. Pixellation involves creating a grid of large blocks over the image, and filling in each block with the average color of the image beneath it, as in Figure 2c. This technique is used in television and video to obscure details such as faces and license plate numbers.

We can create approximate images in other ways. Most image-editing programs contain a wealth of built-in filters to apply special effects to images, and several companies sell packages of additional filters as plug-ins. Some of these filters are simple and run with a single touch of a button, while others are complex with many user-interface controls.

An interesting approach to creating an approximate image was presented in a 1990 paper by Paul Haelberli (see the "Further Reading" sidebar for details). Among other approaches, he mentioned using "relaxation" to create an image and showed two example images. Although

Further Reading

The original inspiration for this column came from a couple of pictures at the end of the paper "Paint by Numbers: Abstract Image Representations," by Paul E. Haeberli.[1] Those figures show a picture of a man sitting in a chair approximated by relaxed boxes and a woman standing up approximated by a relaxed Voronoi diagram.

Adobe's Photoshop has a built-in filter called Crystallize that appears to build a Voronoi diagram and then colors each cell with the average color under the cell. The filter doesn't seem to make any use of image information to place the points, and some cells have curiously soft or blurry boundaries.

One of the first papers to present a nice way to get nonphotorealistic filters to work across animated sequences is "Painterly Rendering for Animation" by Barbara J. Meier.[2]

Many excellent filters exist for both still and moving images. Some filters use image-processing techniques to track the changes between frames and minimize the boiling effect when the geometry maneuvers to accomodate the new image. For example, some of the filters created by RE:Vision Effects for the film *What Dreams May Come* (which are now available commercially) work this way.

Writing an efficient and stable program to compute Voronoi diagrams is no picnic. Rather than starting from scratch, I recommend building on C language source code written by Steve Fortune of Bell Labs, which he has made available at http://cm.bell-labs.com/who/sjf/. You can also find code for computing Voronoi diagrams and many other algorithms at the Computational Geometry Algorithms Library (CGAL) at http://www.cgal.org/.

References

1. P.E. Haeberli, "Paint by Numbers: Abstract Image Representations," *Computer Graphics* (Proc. Siggraph 90), ACM Press, New York, vol. 24, no. 4, 1990, pp. 207-214).
2. B.J. Meier, "Painterly Rendering for Animation," *Computer Graphics* (Proc. Siggraph 96), ACM Press, New York, vol. 30, no. 4, 1996, pp. 477-484)

that paper doesn't discuss how the pictures were made, relaxation is a well-known method for optimization, so it's not hard to guess what he was up to. In the next section I'll talk about writing a simple optimizer and then present the results of using that system with a variety of geometric elements to make approximate images.

Around the loop

Depending on the complexity of the problem you want to solve and the sophistication of your solution, writing an optimization program can mean a few hours of low-stress programming or become a life's work. Of course, for this column I'll take the former approach. Specifically, I'll look at what's sometimes called a *random-walk relaxation* routine.

The general idea behind this kind of program is that we have two basic creatures: a piece of data called a *candidate* and a routine that evaluates a *fitness function*. The fitness function looks at the candidate and assigns it a numerical score. Typically a fitness function will assign a higher score to a better candidate than to a less one. Thus, we want to maximize the score for maximum fitness.

If instead a better candidate gets a lower score, the fitness function is sometimes called a *penalty function*, because we want to minimize the penalty.

In this column, our candidates will be collections of flatly colored geometric objects that we'll distribute over the original picture to create a new image. The fitness function will measure the total error in color space. Since we want this error to be as small as possible, this is technically a penalty function.

The process will be to generate an initial candidate, which I'll call the *best* candidate, and score it. Then I'll generate another candidate by randomly *perturbing* the best candidate, which just means changing it in some way. If the new candidate has a worse score than its predecessor, we throw it away. If its score is better, then it becomes the new best candidate. Then we perturb and score again. Many tests exist for determining when to stop this loop, which I'll discuss a little later.

Because I wanted to try out a bunch of different geometries, each with their own descriptions and parameters, I thought it would be easiest to write a general-purpose routine to control the optimizing process.

```
1.  Candidate PerturbStep(double *bestScore,
        Candidate bestC, Procs procs)
2.  {
3.      Candidate newC = procs->perturb(bestC);
4.      double newScore = procs->score(newC);
5.      if (newScore < *bestScore) {
6.          procs->save(newC);
7.          bestC = newC;
8.          *bestScore = newScore;
9.      }
10.     return(bestC);
11. }
```

3 Pseudocode for the basic perturbation routine.

One step at a time

This project is a natural for class-oriented programming languages like C++, though using function pointers you can write it up in other languages as well. For the little fragments of pseudocode in this article, I'll use traditional C-style notation.

Each type of candidate is implemented by exposing four functions to the system. For simplicity's sake I'll abstract them here by ignoring the important but cumbersome details of bookkeeping, memory management, error-checking, and so on. Of course in any real system you'll need to manage those issues, but they're not central to the ideas of the algorithm. The four functions are

- *Candidate init()*. Called before the optimizer runs, this sets up any required storage, and generates and returns the first candidate.
- *Candidate perturb(Candidate c)*. Perturb the input and return the perturbed version.
- *double score(Candidate c)*. Compute the penalty score for this candidate.
- *void save(Candidate c)*. Save the image specified by this candidate.

Figure 3 shows the basic loop of the perturbation algorithm. I'll assume that we've already initialized the geometry (giving us the original best candidate) and scored it (giving us the original best score). The basic idea is that the routine takes as input pointers to the current best score and candidate, as well as a structure with the procedures for the current type of geometry. On line 3 we perturb the input candidate to create a new candidate, and on line 4 we get its score. If the score is better than the best score we have so far, then we save the new candidate, put its score into the pointer for the score, and reassign the candidate pointer to the new structure. Then we return the candidate pointer. If the new version wasn't any better than what we had before, that pointer and the score will be unchanged. Otherwise they'll both have the new values.

When I save a candidate, I simply write out the image file that it describes.

The procedures that get called aren't hard-wired into the loop but get looked up on the fly through pointers in the structure that holds them. That makes it easy to write up and plug in new types of candidates: just fill the structure and the program runs as before, except it calls the new routines.

Note that in C a procedure can only return one value, and this routine might need to change two things—the score and the pointer to the best candidate. To manage that in this pseudocode, I passed in the current score as a pointer, so that the routine could change its value. In real code you'd want a more elegant solution.

Scoring

Coming up with a score is the fitness function's job. For this column I started with a simple function that took two inputs: the candidate image (*cImage*) and the reference image (*rImage*). The cImage is the rendered image produced by the geometry in the candidate, and the rImage is the original picture that we're trying to approximate.

My first metric was to run through all the pixels and compute the squared difference in each color component. If Δr is the difference between the red values in the two pictures, then the error at each pixel is $E_p = \Delta r^2 + \Delta g^2 + \Delta b^2$. The total error is the result of summing up this measure for all pixels in the image. I square the values so that they'll always be positive, and big differences will count for more than smaller differences.

This worked pretty well at capturing big blocks of color, but it has no special preference for preserving edges, which are visually important. It also doesn't let the image creator influence the process. I decided to add a simple importance measure to the error metric. When you start the program, you have the option of supplying an importance image (*iImage*) of the same size as rImage. You can also supply an overall scale factor *s*, which scales to the entire importance image. Writing *I* for the value of iImage at a specific pixel (scaled to the range [0,1]), the importance-weighted error is $E_i = sI E_p$. This means that pixels that have been painted white (or assigned a high importance) contribute more to the final score. For the figures in this column, I typically painted white around important edges and places where I felt details were important. Figure 4 shows an image with its hand-drawn importance image.

If you don't want to exercise personal control over the algorithm but would like to encourage it to preserve regions of high contrast, you could write a program to find edges in the original image and create an importance image where pixels near edges are shaded from black to white proportionally to the strength of their edge.

In addition to the importance-weighted color differences, I've included two more measures in my scoring function. The first is an overlap penalty. I count how many times a given pixel is written to over the course of rendering the image. Let's call this counter *c*. If $c > 1$, I multiply the excess by an overlap penalty value P_o and add that in to the total error. That is, I add in a value $(c - 1)P_o$, where P_o is a constant for the whole image. The larger the overlap value, the more the pieces will be penalized for landing on top of each other. Similarly, I include a gap penalty. If a given pixel has a count value $c = 0$, then I add to the score the value of a gap penalty value P_g, which again is constant for the whole image. The idea here is to try to get all the pixels covered by at least one object, so that most of the background is covered.

Perturbing

The perturbation routine for each type of geometry is different. For example, circles change their center and radius, boxes change their length and angle, and so on. But a few things are common to all the perturbation routines.

Each candidate is a list of geometric objects: dots, boxes, and so on. The rendering algorithm draws these in order into a blank image, so that in effect they're rendered back to front.

When we call a perturbation routine, it first throws a random number. If that number is above a threshold, the routine swaps a random pair of objects and returns. So none of the shapes change, but their order is altered. I set the swap probability and leave it unchanged throughout the entire run. Generally I put the threshold at around .9 or .95, so pieces get swapped about one out of every 10 or 20 calls to the routine.

If there's no swap, the routine picks a piece and changes it. Because I don't want to pick pieces in the same order every time, when the perturb routine is called for the first time I create a *permutation array* (let's call it *P*). *P* is an array of integers that has as many entries as there are pieces of geometry (let's call that number *L*), which I initialize so that each $P_i = i$. Then I run through the array and for each element, I choose another element and swap them. That means that every element will get swapped at least once, and probably some will get swapped many times. The result is that the array still holds the numbers 0 through $L - 1$, but in scrambled order.

I then set a counter (which is persistent to the routine) to 0. The first time I enter the perturb routine, I choose the element from the list that corresponds to the value of P_0. The next time it's P_1, then P_2, and so on. I just keep bumping the counter each time. When the counter hits *L*, I rebuild the array and start the counter again at 0. This way the pieces get modified in a random order, but no pieces get skipped.

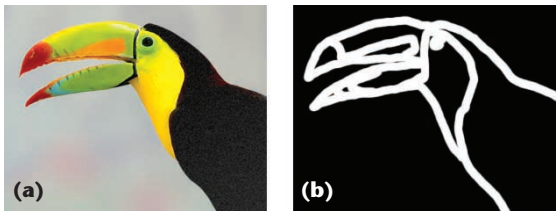
Now that we've selected a piece of geometry to modify, we need to decide what to do with it. Each piece of geometry has its own data. A circle has a center and radius, a triangle has three points, and so on. We usually want these values to vary a lot in the beginning simulation steps so that objects can try many different places to land and see how well they contribute there. As the simulation continues, the changes get smaller as pieces jostle into their best settings.

I manage this with a simple linear transition. I set a scale factor for the start of the simulation and one for the end. Because I know how many steps I'm going to take before I begin, I just interpolate based on the current step to come up with a current scaling factor. For example, I might say that the radius of a circle can change by up to ± 40 pixels at the start of the run, but only up to ± 5 pixels by the end.

In a more sophisticated technique, you might want to run the simulator until the changes become very small and save the result. Then reset the change factor to its maximum value and try again. This way the system can explore a variety of different starting conditions and follow each one to its best conclusion.

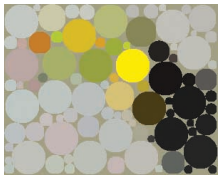
Toucan play that game

Let's look at a bunch of different geometric objects

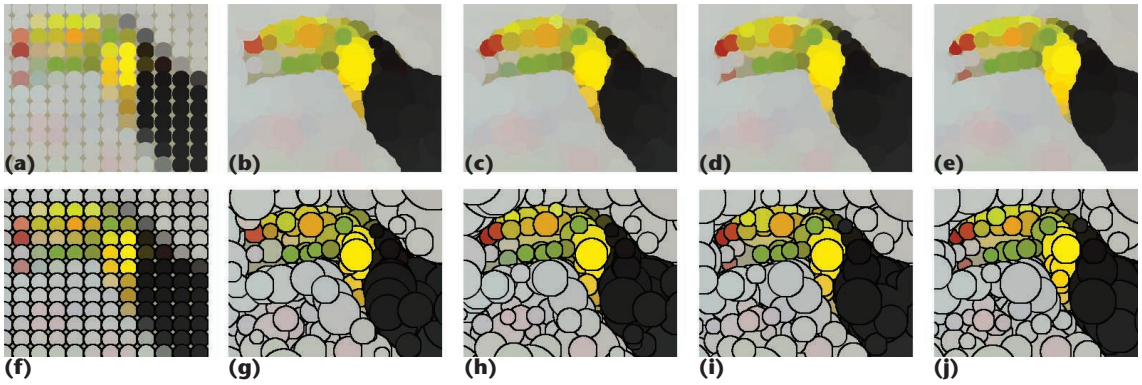


4 (a) An image of a toucan. (US Fish and Wildlife Service/photo by Mike Lockart.) (b) The hand-drawn importance image. White means most important, black means least.

5 The 102 boxes produced by the scaling-box algorithm for the toucan image.



6 The 144 dots produced by the scaling-dot algorithm for the toucan image.



7 Using relaxed dots to approximate the toucan. For each image, the first number for each part refers to how many steps the algorithm took to produce that picture. The second number identifies how many times a new candidate replaced an old one since the start of the run. (a) The original 12 × 12 grid of dots. (b) 6,000 / 485. (c) 12,000 / 680. (d) 18,000 / 834. (e) 25,000 / 1,093. (f–j) Figures 7a through 7e with outlines.

applied to an image of a toucan. This is a good test image because it has just a few regions of consistent color, a few details, and sharp, high-contrast edges. Figure 4a is the original, and the important image is Figure 4b.

In each of the following techniques, I create an image from a candidate by starting off with a solid field that's set to the average color of the entire original image, and then add regions of constant color to it.

In each method except for the first two, I made the images with 144 geometric elements and 25,000 steps of relaxation (of course, not every step of relaxation was accepted by the optimizer as an improvement). The original formation was a 12-by-12 regular grid of elements of identical shape and size.

Scale boxes

I'll start off with two approximations that aren't really generated by relaxation, but they give us a nice starting point.

Suppose that in the original picture you try to find the best location to place a box of a given side length. That is, you center the box on every pixel of the image and determine whether it overlaps any other boxes. If it doesn't, then compute the error you'd have if you replaced all the pixels under the box by their average color. Repeat this for every possible location.

When you're done, you'll either have found that you can't place the box anywhere, or you'll have the location of the box that introduces the least error.

If you found a spot for the box, add it to the list of boxes, creating the newly "perturbed" candidate, and return it. If you couldn't place the box, decrease the side length by a given fraction and start again. Repeat the process until the box's side falls below some minimum size.

Figure 5 shows the result. In this example, the boxes started with a side length of 1/10 the length of the smaller dimension of the image and continued until they were 10 pixels on a side. My system was able to squeeze in 102 boxes until they got too small.

Scale dots

The approximation of scaled boxes in Figure 5 isn't much to write home about. What if we replace the boxes with dots? Then the dot's diameter replaces the box's side lengths. Figure 6 shows the result, using the same sizes for the radius as for the boxes in the previous figure. I stopped the run when it packed in 144 dots because all the other simulations use 144 elements and I didn't want this one using more.

This image also isn't that much to cheer about. Let's see if we can improve things by including relaxation into the process.

Dots

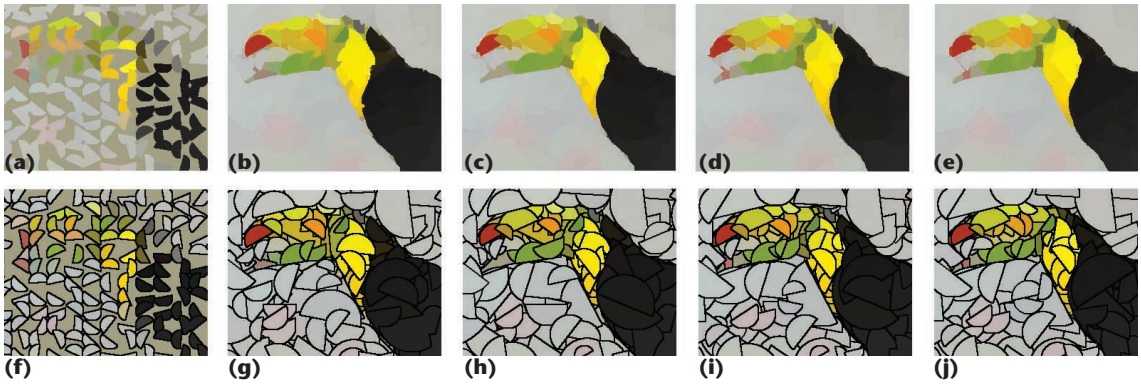
As with all the examples to follow, I'll begin with 144 elements laid out on a regular grid, as in Figure 7a.

Because we're using dots, each element is described by three numbers–one each for the x and y coordinates of the center and one for the radius. Each step of the perturbation either swapped two dots or changed the center and radius of one of them.

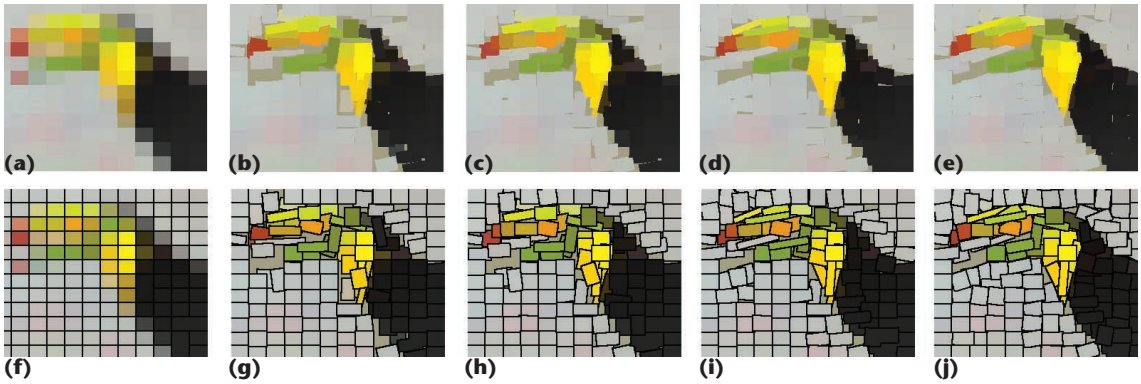
The rest of Figure 7 shows the original dots and the results after 6,000, 12,000, 18,000, and 25,000 steps, respectively. By the last image, the system had accepted a new candidate 1,093 times. So that you can better see the way the dots are stacked, I've also provided a version of each figure where the dots are outlined.

Half dots

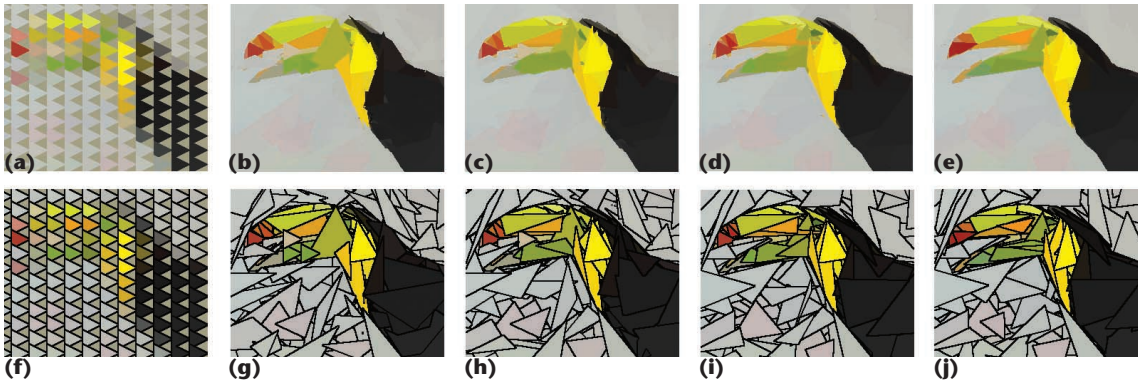
Dots look pretty good. What if we instead used half



8 Using relaxed half dots to approximate the toucan. (a) The original 12 × 12 grid of half dots. (b) 6,000 / 670. (c) 12,000 / 844. (d) 18,000 /1,026. (e) 25,000 / 1,242. (f–j) Figures 8a through 8e with outlines.



9 Using boxes to approximate the toucan. (a) The original 12 × 12 grid. (b) 6,000 / 228. (c) 12,000 / 318. (d) 18,000 / 435. (e) 25,000 / 602. (f–j) Figures 9a through 9e with outlines.



10 Using triangles to approximate the toucan. (a) The original 12 × 12 grid of triangles. (b) 6,000 / 635. (c) 12,000 / 882. (d) 18,000 / 1,079. (e) 25,000 / 1,412. (f–j) Figures 10 through 10e with outlines.

dots? These have a center and radius but also a fourth number: an angle that locates a diameter through the dot. Only pixels on that diameter's positive side get drawn.

Figure 8 shows the results. After 25,000 steps there were 1,242 steps of improvement. Note that the little pointy bits of the half-dots are useful for filling in crevices.

Boxes

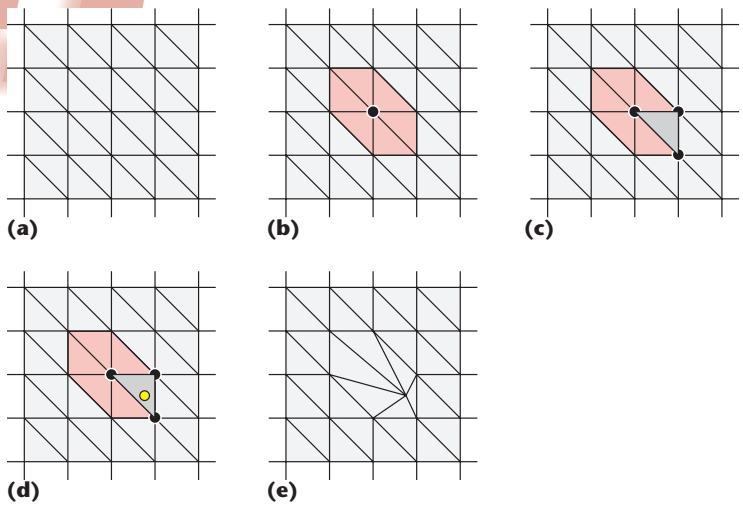
Let's use boxes as our geometric elements. An easy way to define a box is with five numbers that specify its center, width, height, and an angle of rotation about its center. Whenever it's time to perturb a box, all these parameters get tweaked.

Figure 9 shows the results. After 25,000 steps, there were 602 steps of improvement.

Triangles

It's a small step from boxes to triangles. A triangle is described by six numbers, corresponding to the locations of each of its three vertices.

Figure 10a shows the results. After 25,000 steps, there were 1,412 steps of improvement.



11 Geometry for the mesh. (a) The basic mesh. (b) Each vertex can move within the hexagon formed by its six closest neighbors. (c) A triangle formed by the vertex and two adjacent neighbors. (d) Finding a point in that triangle. (e) Moving the vertex to that point.

Mesh

Rather than letting the triangles roam free, let's tie them down to a mesh, like that of Figure 11a. The relaxer will then just move around the inner vertices of the mesh (the vertices on the border will remain fixed). My perturbation procedure starts with a hexagon made of the points' nearest neighbors, as Figure 11b shows. To move a vertex, I randomly select an adjacent pair of these neighbors, which gives us a triangle (Figure 11c). I pick a random point in the triangle (Figure 11d) and move the vertex to that point (Figure 11e). I try to pick a point that's not too close to the edges so that the triangles don't get too long and skinny right away. Writing $?(a, b)$ for a random number in the range (a, b), I compute three numbers $a_0 = ?(.2, .7)$, $a_1 = ?(.1, 1-a_0)$, and $a_2 = 1-(a_0 + a_1)$. I then randomly apply these weights to the triangle's three vertices to get the new point. Because the weights are all positive and they sum to 1, the new point is guaranteed to lie inside the triangle.

Although the connectivity of the triangles (that is, the mesh's topology) doesn't change over the course of the simulation, the mesh can flop over on itself. So, some triangles can land on top of others. The overlap penalty helps reduce this from happening too much.

Figure 12 shows the results after 25,000 steps and 616 updates.

Voronoi cells

A *Voronoi diagram* contains a set of convex regions called *cells*. Each cell contains a single *input point* and all the points that are closer to that input point than to any other.

You can use that description to write a brute-force Voronoi renderer that fills each cell with the average color of the original pixels under it, using a list of N 2D input points to define the cells. Let's do it in two passes.

Before we start, though, we'll create an auxiliary grayscale image called V, where each pixel is initially set to -1. What we want to do is to fill each pixel of V with a number from 0 to $N-1$, indicating which input point it's closest to.

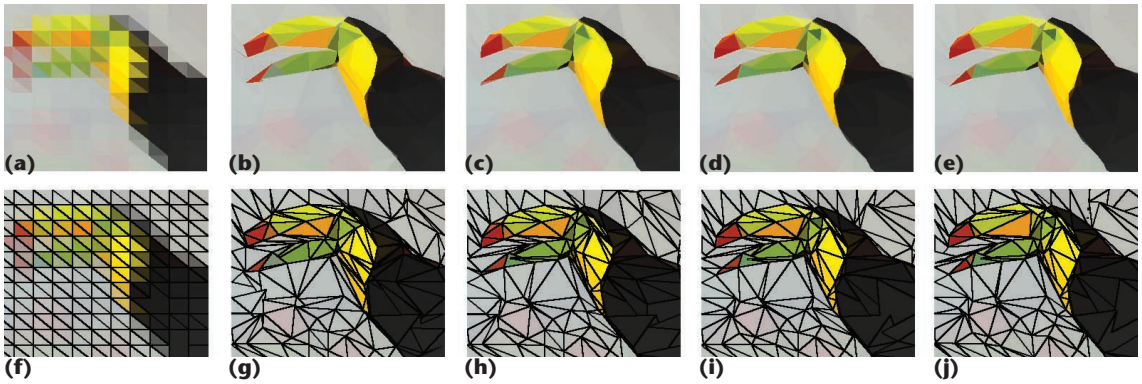
We'll also create two 1D arrays. First, we'll make an array C of N colors, and set each to $(0, 0, 0)$. We'll also create a counting array K of N integers, which we'll set to 0.

For the first pass, we'll visit each pixel (x, y) in the original image, compute its Euclidean distance to each of the input points, and find the closest one. (Because we're only comparing distances, we can work squared distances and save ourselves an expensive square root.) Let's call the nearest input point v. We'll set the corresponding cell of V to v. We'll also add that pixel's color from the original image into C_v and increment K_v by one.

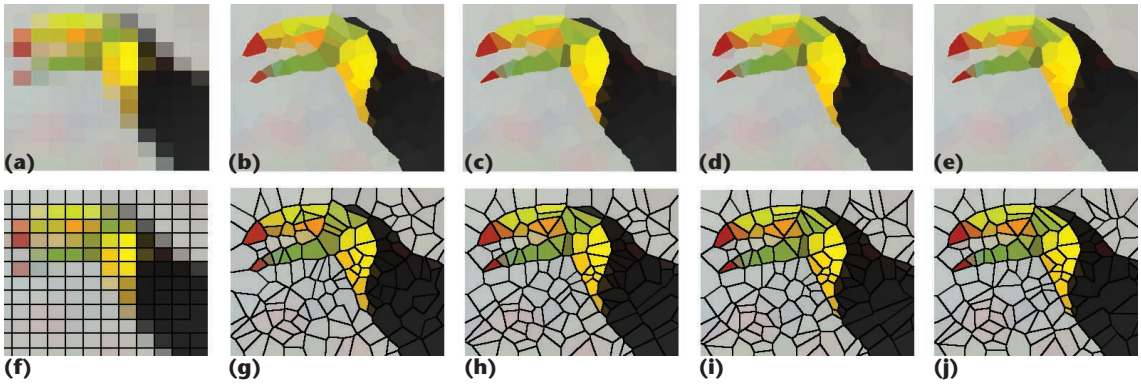
When we're done with the first pass, each element C_v contains the sum of the colors of all the pixels that are closer to input point v than to any other, and K_v tells us how many of them exist. We'll divide each color C_v by K_v to get C'_v, the average color of the pixels in that cell.

Now comes the second pass, which is easy. We scan through V, get the index v, and we place the color C'_v into the corresponding pixel in the output image.

This algorithm works just fine, but the first pass is slow. It also doesn't scale well. If we use N input points, then every pixel has to compute its distance to all N points. If we double the input points to $2N$, it takes twice as long. The easy way to speed things up is to actually compute the Voronoi diagram using an efficient algorithm. To get going, I recommend using one of the free implementa-



12 Using the mesh to approximate the toucan. (a) The original 12×12 grid of boxes. (b) 6,000 / 374. (c) 12,000 / 486. (d) 18,000 / 571 (e) 25,000 / 616. (f–j) Figures 12 through 12e with outlines.



13 Using a Voronoi diagram to approximate the toucan. (a) The original 12 × 12 grid of cells. (b) 6,000 / 369. (c) 12,000 / 505. (d) 18,000 / 599. (e) 25,000 / 769. (f–j) Figures 13a through 13e with outlines.

tions available on the Web (see the "Further Reading" sidebar).

Figure 13 shows the results. After 25,000 steps, there were 769 improved images.

Measure for Measure

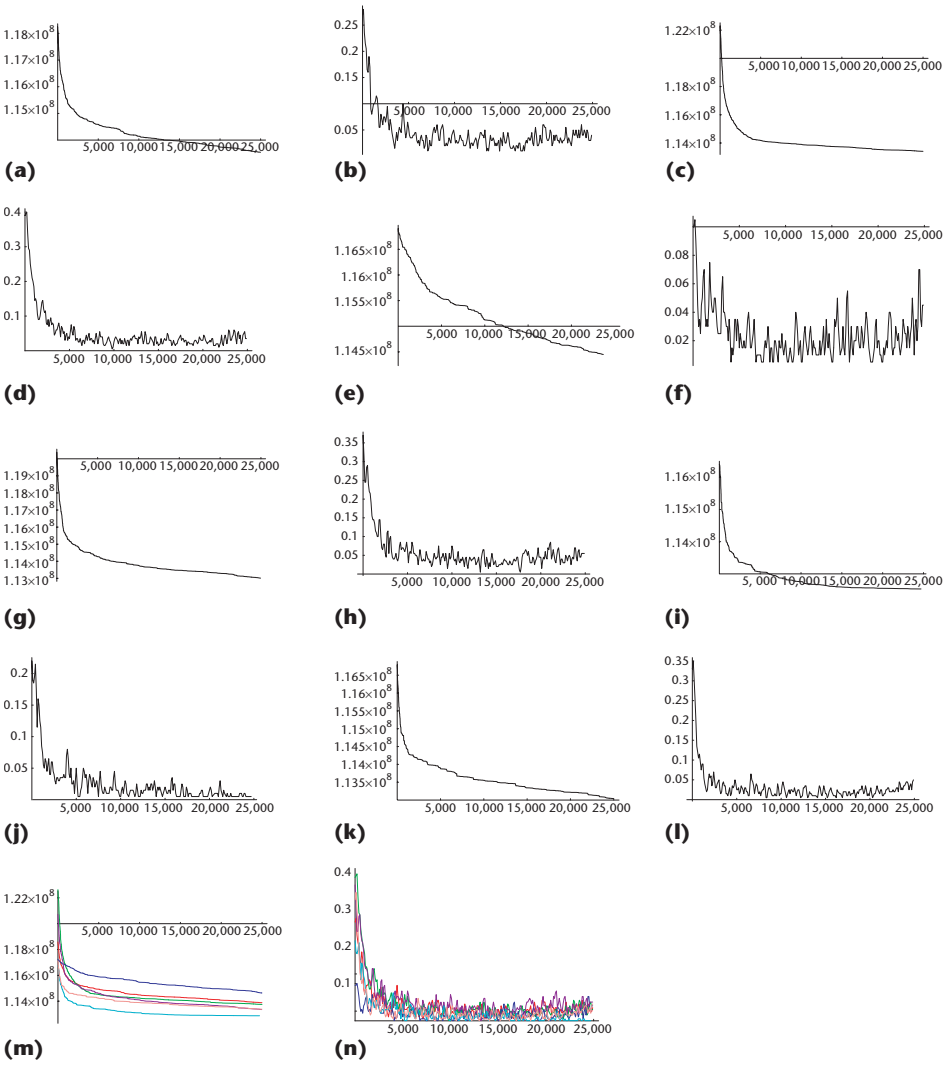
Let's take a look at the performance of these different filters. In Figure 14, I show numerical data gathered from the runs that produced the images in Figure 13.

To make the error plots, I drew a line of constant error from each iteration where a new image was produced to the right, until I reached the next iteration with a new error value. This is sometimes called a *sample and hold* plot.

The update plots try to capture the relative frequency with which new candidates are accepted. For each point on the plot, I looked at a window of 100 steps centered at the point in question and counted up how many times a new candidate was accepted. Then I divided that count by 100. That gives us a rough measure of the probability that a new candidate would be accepted as the simulation chugs along.

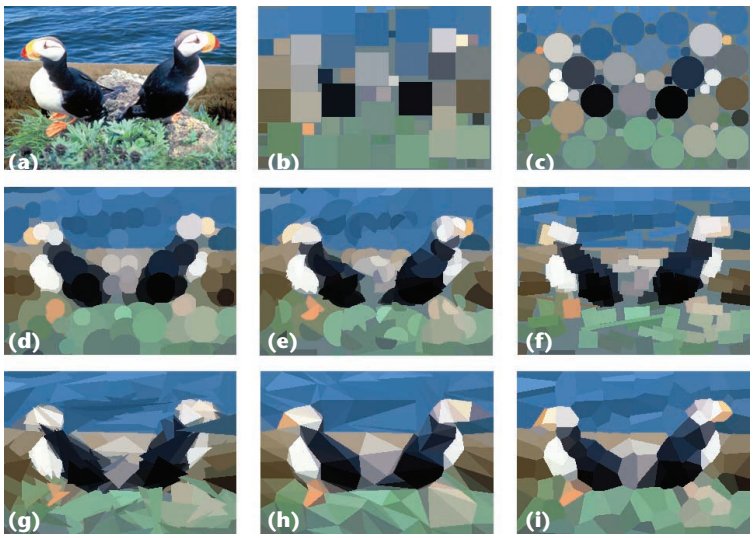
Figures 14g and 14h show these results superimposed. The point isn't to actually read off the data, but to see how the different algorithms line up. Surprisingly, though the mesh is the most constrained geometry, here it's the best at minimizing the error.

Of course, we don't want to go too far in using these error values to measure the quality of the results. After all, if we wanted an error of zero, we could have it just by using the original image and spare ourselves all this effort! The real test comes from how



14 Data for the toucan. For each type of geometry, the first plot shows the error over time. The second one shows the probability of finding a better candidate over time. (a) Dots error. (b) Dots probability. (c) Half dots error. (d) Half dots probability. (e) Boxes error. (f) Boxes probability. (g) Triangles error. (h) Triangles probability. (i) Mesh error. (j) Mesh probability. (k) Voronoi error. (l) Voronoi probability. (m) Combined error plots, all scaled to the same range. Dots are in red, half dots are green, boxes are blue, triangles are violet, the mesh is dark cyan, and Voronoi cells are pale red. (n) Combined update plots, using the same color scheme.

15 Applying the filters to a picture of a Black-Eyed Susan. (Photograph by the US Fish and Wildlife Service.) The number in parentheses reflects how many times over the run of 25,000 iterations the perturbation was an improvement and became the new best image. (a) The original image. (b) Scale boxes (98). (c) Scale dots (144). (d) Dots (1,005). (e) Half dots (1,128). (f) Boxes (749). (g) Triangles (1,219). (h) Mesh (560). (i) Voronoi (540).



16 Applying the filters to an image of two puffins sitting on a rock. (US Fish and Wildlife Service/photo by Richard Baetsen.) (a) The original image. (b) Scale boxes (66). (c) Scale dots (91). (d) Dots (917). (e) Half dots (1,100). (f) Boxes (810). (g) Triangles (1,265). (h) Mesh (475). (i) Voronoi (575).

the images look—the error is just to get a feel for how the algorithms proceed with their work.

More examples

I've run these eight algorithms on a few other images to show you how they look.

Figure 15 shows an image featuring a prominent Black-Eyed Susan, Figure 16 shows the results for a pair of puffins sitting on a shoreside rock, Figure 17 shows the filters applied to a hummingbird feeding at a yellow flower, and Figure 18 shows what happens to an image of a child's soft toy train.

It's fun to watch these algorithms do their stuff, shuffling the pieces around from their starting shape and size to their final position. You can find animation files for each of the examples in this column at http://computer.org/cga/cg2002/g5toc.htm.

Because of their low resolution, these filters work best on images with large regions of similar color, where what really matters are the shapes and colors, and not the fine details. Running these filters on an image of a zebra is likely to replace its beautiful black-and-white stripes with a gray mush. Of course, if you like those stripes, then there are other ways to keep them while compressing the image. My goal here wasn't compression of the original image, but nice low-fidelity approximations that look interesting.

Discussion

The filters presented in this column aren't fast. On the other hand, I paid no attention to efficiency when I wrote the programs and ran them in debug mode. Just a little profiling and tuning would doubtlessly improve performance. On images of about 300×200 pixels, my 1.7-GHz home PC can deliver about 2 or 3 iterations per second for most of these filters. Making the 40 examples for this column took about 120 hours of steady computing. Although these filters are impractical for routine use today, I think that would be easy to change.

First, you could write better code and then compile it with optimizations. Second, I started each run by placing my geometry in a regular grid. Putting the elements down in a distribution that even roughly corresponds to the importance diagram would probably give the program an enormous head start. Third, I evaluate each image from scratch for each iteration. Because only one or two geometric elements change per iteration, you could find the bounding boxes for those elements and just redraw the pixels in those boxes, which would mean only revisiting the geometric objects that fall into those boxes. Fourth, when I perturb my objects, I do so randomly. Again, using information from the importance diagram could go a long way toward getting them into good configurations quickly, rather than waiting for them to try many alternatives before lucking into a good spot. Fifth, one could use an estimate of the gradient of the error to nudge pieces in the direction where they'll do the most good. And finally, computers are getting faster all the time, so what's prohibitively slow today will certainly be very fast in the future. With enough speed, it would be fun to have a slider that set the number of steps to take, or the number of improvements desired. Then by moving the slider back and forth, you could see the image interactively at different qualities of approximation and pick the one you like the best.

If you run one of these filters on one frame from an animation and then run it again on the next frame, the geometry will probably pop around quite a bit. The "boiling" image look is sometimes desirable, but usually not. We can generally avoid it by using the final candidate for one frame as the starting candidate for the next, (see the technique in "Painterly Rendering for Animation" in the "Further Reading" sidebar). As long as the two

images are somewhat similar (that is, they have some image coherency), the geometry shouldn't have to move too much to accommodate the changes.

You could actually make sure of this by using ideas from image processing and optical flow to deliberately move the geometry along paths (smooth or ragged, as desired) from one frame to the next.

As with any program that takes a long time to run, it's a good idea to save periodic checks as you go, in case the program needs to be stopped or the computer crashes. The easiest checkpoint for these filters is a list of the current geometric elements and the current iteration number. The program just takes the input as the best candidate and continues until it completes the desired number of steps. Using the final checkpoint of one frame of an animation as the starting state of the next is an easy way to reduce the boiling effect.

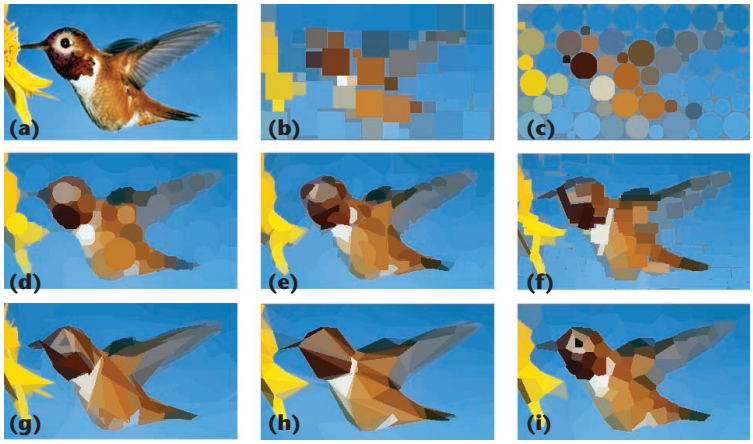
It's also fun to take one set of geometry and apply it to another picture. Sometimes you get an interesting merge where you can discern the first picture from the shape of the pieces, but it seems to be hiding among the colors that come from the second picture. This effect is particularly noticeable in animation when your eyes have a few frames to catch and track the contours.

I ran each filter in this column on each image for 25,000 iterations. The number 25,000 was arbitrary. It just seemed like the simulations were settling down around there in my tests. It would be much better instead to use an automatic error-based stopping condition. A simple but effective condition is to stop when the score for an image (that is, its total error) is below some numerical threshold.

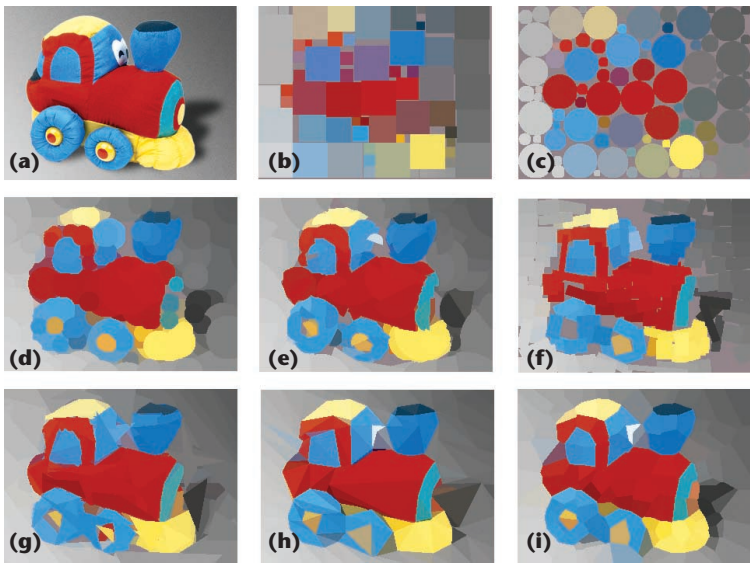
This is easy, but it has the problem that it can be hard to know what threshold to pick. After all, these images are approximations and will always have some error.

A better solution is to try to detect when the optimization has flattened out. In other words, you try to detect when the error appears to have stopped improving for a long period of time. An easy implementation of this uses two numbers. The first is a window width W, and the second is an error change threshold C. We apply the test every W iterations and find the difference between the score now and the score W iterations ago. Then, we divide that difference by the score at the end of the window. This gives us a measure of the relative improvement over the last W iterations. If it's less than the threshold C, we stop.

When C is small, the algorithm will continue to run even when the improvements are small in value. So you want C to be small enough to let the algorithm find a nice solution, but not so small that you wait endlessly while imperceptible changes accumulate. By the same token, the window W needs to be large enough that you don't stop when you've hit an unlucky sequence of perturbations. My rule of thumb was to set W to about $4N$, where the approximation has N elements. Then I set C automatically: I find the improvement for the first window of W elements, and I set C to $1/100$ of that value. It's not a perfect condition, but it's usually pretty close. If I need more iterations I can just read the output file where the program stopped and restart the system.



17 Applying the filters to an image of a Rufous hummingbird. (US Fish and Wildlife Service/photo by Dean E. Biggins.) (a) The original image. (b) Scale boxes (73). (c) Scale dots (80). (d) Dots (942). (e) Half dots (1,226). (f) Boxes (732). (g) Triangles (1,348). (h) Mesh (114). (i) Voronoi (593).



18 Applying the filters to an image of a child's toy train. (a) The original image. (b) Scale boxes (66). (c) Scale dots (84). (d) Dots (1,039). (e) Half dots (1,218). (f) Boxes (808). (g) Triangles (1,335). (h) Mesh (69). (i) Voronoi (757).

As with any stopping criteria, you'd probably also want to set a large upper limit on the number of steps so the program doesn't run forever.

Writing the filters in this column can be a lot of fun. There's something rewarding about successive approximation. ■

Acknowledgments

Thanks to Steven Drucker for encouragement and discussions on some of these filters.

Readers may contact Andrew Glassner by email at andrew@glassner.com.