Image Search and Replace

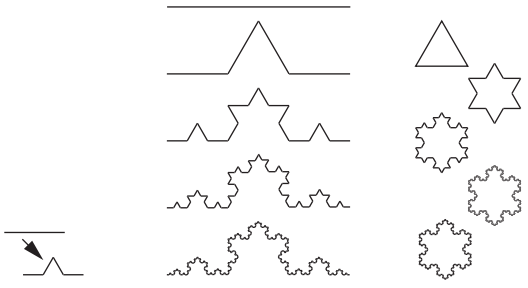
Andrew Glassner

Our world is imperfect—not everything is exactly the way we wish it would be. Perhaps this is a good thing, as improving the world gives us something to strive for.

One way to better the world is to edit it. We start with something that's close to what we want and then improve on it. Editing is a normal part of almost every creative process, whether we're starting with something made by someone else or our own creation.

A common way to improve something by editing is to find a feature we don't like and replace it with something we prefer. Text editors universally provide this ability with some kind of search and replace command. So if you're reading a big document that discusses the adventures of a character named Fred, but you think it would be better if his name was David, you can simply tell the system to replace all occurrences, or instances, of Fred with David.

Most programs let you apply the change as a global substitution, automatically altering each instance of the target (the name Fred) with the replacement (David). Usually you can also tell the system to let you preview each substitution before it's committed, to ensure nothing goes astray in the process. If you forego this manual confirmation step the substitution process goes much faster, but you might end up (if your search is case insensitive) with some character in your story inexplicably ordering pasta with aldavido sauce.



1 Creating one edge of a Koch snowflake. (a) The rule is that we replace each straight line by a line of the same length, but with a point in the middle. (b) Applying that rule four times to a single straight line creates a crinkly shape. (c) Applying the rule to the three sides of a triangle makes a Koch snowflake.

Search and replace is too good an idea to limit to text. In 1988, David Kurlander and Eric Bier showed how to use this idea in vector graphics, or drawings made out of lines.[1] As Figure 1 shows, this is a great way to build up a Koch snowflake: just replace each straight line with a new line with a point on it and then do it again. In this example, the program looks for straight lines at any position, angle, and length and replaces them with the replacement pattern transformed so that the endpoints match.

Here we can see the basic ideas behind all search and replace methods: search for the target, remove the target, and insert the replacement properly transformed to match the target.

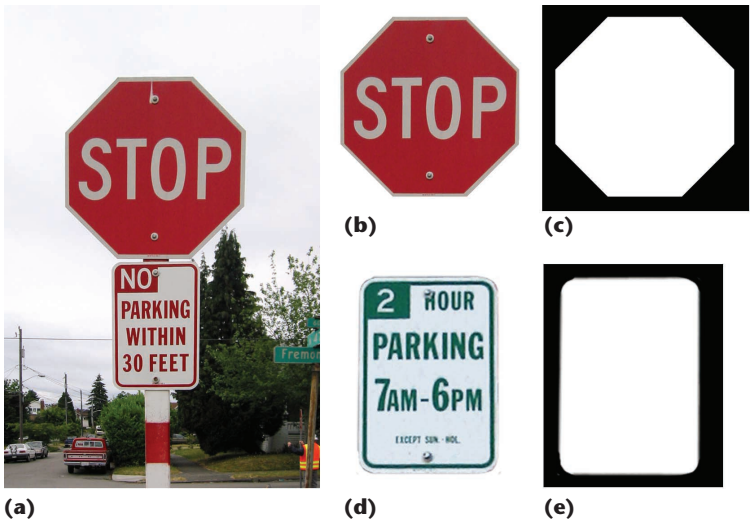
Some interesting variations on this idea have appeared on television in recent years, particularly during sports broadcasts. It's now common for broadcasters to replace the advertising signs in a stadium with different signs, so home viewers and fans at the game see different ads.

Broadcasters use other techniques to enhance the game's visual presentation. For example, some systems augment the field's video image in a football game with a synthetic yellow line that represents where the offensive team must advance the ball to secure a first down.[2] Rather than actually painting the line, they insert it electronically on top of the video signal somewhere between the camera and the transmitter. The illusion is convincing because the system is sophisticated enough to recognize when a player or official on the field has moved into a position where he would be blocking our view of the line, were it really there. When a person on the field obscures our view of the line, the system suppresses drawing it into the image, so that it appears naturally obscured.

Note that this system isn't actually a search and replace technique (there's no searching or removal, and they just add the yellow line to the video). The technique belongs to the field now called augmented reality.

In comparison, I'd like to discuss building a system that will let us perform a search and replace on raster images. If we provide a source picture, a target image, and a replacement image, the system can remove every instance of the target from the source and insert in its place a copy of the replacement. Of course, implementing this is a little more complicated.

 Published by the IEEE Computer Society



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

2 (a) Source image *S*, (b) target image *T*, (c) target mask T_M, (d) replacement image *R*, and (e) replacement mask R_M.

3 If we simply drop the replacement on top of the source, parts of the source might still be visible.

The big and small picture

Let's start with the big picture. The search and replace algorithm starts with three pictures, as Figures 2a, 2b, and 2d show: the source picture *S*, the target picture *T*, and the replacement picture *R*. Our goal is to find each instance of *T* in *S*, and replace it with *R*.

Because it's rare that the target and replacement images will rarely be rectangular, both *T* and *R* can each have associated masks, T_M and R_M. These masks indicate the opacity of their corresponding pixels in *T* and *R*. A value of 0 (or black) means the pixel is transparent, while a value of 255 (or white) means the pixel is opaque. Intermediate shades of gray represent intermediate amounts of transparency. This lets us create a smooth edge around the parts of *T* and *R*.

To build my prototype, I started with a simple searching technique that just marches through the source picture one pixel at a time, looking for copies of the target. This will be much too inefficient when we later start adding in other transformations like rotation, scaling, and color shifting. For simplicity, right now I'll stick to simple brute-force searching for copies of *T* that seem to be pasted right into the source picture without any geometrical or color changes. I'll return to this step later.

Using this automatic searching, we start by looking through the source image for *candidates*. These are regions of the source that might be instances of the target. Unlike text processing, we often can't be sure exactly when we've found a match. Small variations in color or shape can make it hard for a computer to detect a match even when the general features look similar. Rather than making a final decision at this point, when we think we've found a region that could be a copy of the target, we call it a candidate and add it to a growing list of possibilities for later consideration.

Then we'll rank those candidates with some measure, so that the most likely matches will bubble up to the top. A simple sum of absolute color differences on a pixel-by-pixel basis does a pretty good job of distinguishing the close matches.

Next, the algorithm presents each candidate to us one by one, in their ranked order, the best ones first. If we say that yes, this candidate should be replaced, the computer can't just drop the replacement over the source. Take a look at Figure 3. You can see if we simply drop a copy of the replacement *R* over the target *T*, bits of the target will still be visible. What, then, can we do?

The answer comes from a new class of texture-synthesis algorithms. These algorithms take some piece of reference texture and "grow" arbitrary amounts of that texture to fill any desired region. They can also blend that new texture into the boundaries of a region so that it smoothly merges with the parts of the image already existing.

We can see this in action in Figures 4a and 4b (next page). The target mask T_M tells us which pixels we need to replace. Now we can simply write the replacement pattern *R* on top of where *T* was, as in Figure 4c, and all is well. The replacement mask R_M tells us which pixels to copy out of *R* and how strongly to blend them into *S*.

By default, the algorithm positions the replacement over the target so that centers of both bounding boxes overlap. Of course, we can change this default alignment on both images and on a per-substitution basis.

In summary, the basic algorithm goes like this:

1. Look through *S* and find candidates, assigning each one a score based on the quality of the match.
2. Offer us a chance to accept or reject the highest-ranked candidate (we can skip this step if we feel brave). Once we make the choice, remove this candidate from the list. If we decide that this candidate shouldn't be replaced, jump to step 5, or else continue to step 3.
3. Remove this instance of *T* from the source image, and fill in the hole with synthetic texture.
4. Draw the replacement image *R* over the spot where the match was found.

4 (a) First we remove the source (blue indicates the removed pixels). (b) Then we fill in the source with a synthetic texture. (c) Now we can drop the replacement into the scene.

5 We have to be careful if we want to catch the upper copy of the sign, which is only partially visible.

5. If there are candidates yet to be considered, return to step 2, or else quit.

Let's look at the scoring and synthesis steps a little more closely.

Scoring

Probably the easiest way to determine if one picture resembles another is to compute a penalty or difference score. Add up the differences in the color values of the pixels, one by one. We find the absolute difference in each of the red, green, and blue components and add them together. Lower penalty scores indicate better matches.

There's a little gotcha with this technique: it has trouble if some of the target falls off the edge of the source, as in Figure 5. We might just ignore any pixels in T that fall outside of S, but then we'd only accumulate penalty scores for a smaller number of pixels, which would reduce the overall penalty and make this match look really good. In the extreme, there might be just one pixel of T that overlaps S, and if they happen to be the same color, then the penalty score would be 0, indicating a perfect match!

One way to fix the score for partial overlaps is to divide the total penalty score by the number of pixels actually compared, creating an average per-pixel difference. But we'll want to still make sure that there are enough pixels compared, so that we're considering a significant overlap. The easy way here is to set a user-adjustable threshold. For example, at least 40 percent of the pixels of T must overlap with S to even consider that position as a candidate. If we set this percentage to 100, only copies of T completely contained within S are candidates. Smaller values will catch instances of T partially over the edge, but will also start picking up more noise and bogus instances.

Synthesizing

Once we accept a candidate for T, the system needs to get rid of it. The first part is easy—just erase pixels into the source, modulated by the mask T_M, as in Figure 4a. The second part requires filling in those erased pixels with something that looks good.

In recent years we've seen numerous algorithms that can create arbitrary amounts of seamless, synthetic texture in an image, such as those presented by Heeger and Bergen[3] and Portilla and Simoncelli.[4] These are sometimes called texture-expansion algorithms, because rather than creating texture from first principles, they start with a piece of reference image and then generate a new texture that looks like the reference.

It's easy to get going with basic texture expansion algorithms, even if you don't want to write the code

yourself. For example, free source code is available for a plug-in for the GIMP image editor,[5] and Alien Skin Software's Image Doctor is available commercially as a plug-in for editors such as Adobe's Photoshop. For my prototype, I wrote a simple texture generator based on Wei and Levoy's technique,[6] along with a robust substitution method that Igehy and Pereira[7] designed for masked images.

Reference samples

Texture expansion algorithms that create texture work by generalizing from a reference sample. Where that sample comes from makes a big difference to the image. For example, in Figure 6b I deleted a patch of water. In Figure 6c I selected a reference from a patch of the background greenery, and as you can see the results are more surreal than realistic. By contrast, in Figure 6d I told the system to create new texture based on some nearby water, and the result looks much better.

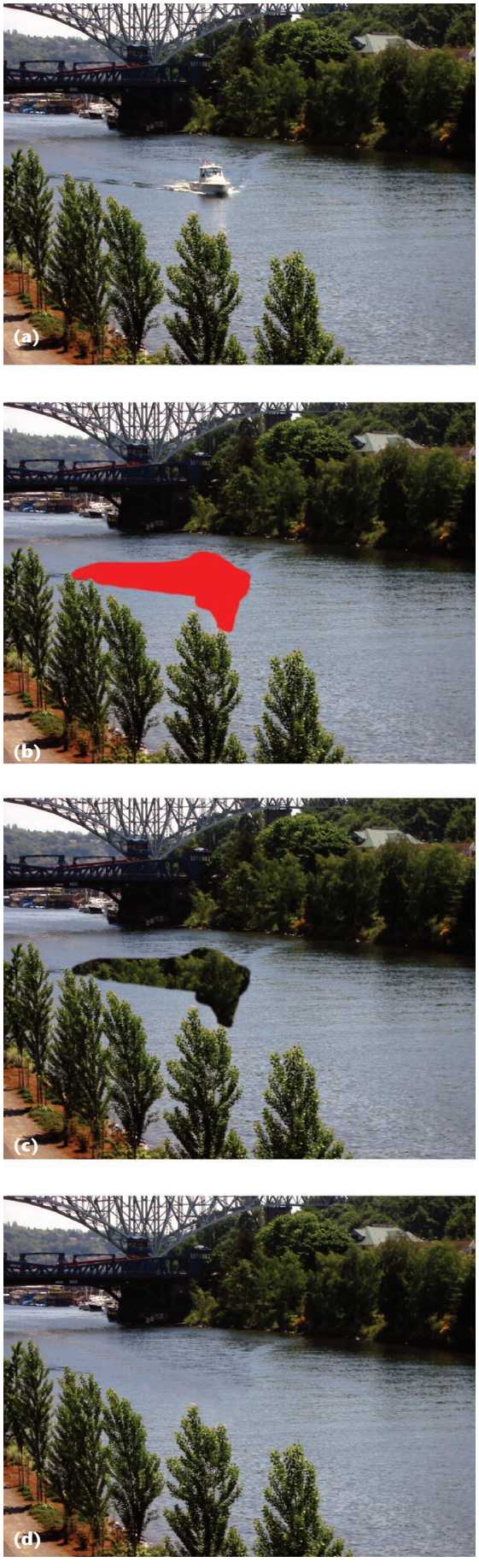
This leads us to the question, When we replace a candidate, where should we go looking for new texture? There are a few answers, depending on how hard we want to work. I think the simplest answer is to look in the neighborhood of the pixels we're replacing. For example, if we draw a bounding box around the patch we're replacing and then expand it, then we can pull texture from the region between the two boxes.

This solution works pretty well in general, and it serves as a good starting point for many replacements, but it won't always work. Each time the system offers us a candidate for replacement, it can also show the region from which reference texture will be drawn for that replacement. If we don't like the choice of region, we can identify a new region somewhere else in the image.

Obviously this solution won't always work. One common problem is when there just isn't a big enough patch of clean texture available. For example, if we want to replace a sky full of hot-air balloons, there might not be much blue sky visible between them. So we can also point the system to another image for reference. In this case, all of our balloons might get erased by using reference texture taken from another picture of an empty blue sky.

Another problem with using nearby pixels arises when those pixels also contain pieces of other objects, including other target copies. For example, we might want to get rid of a computer mouse sitting on a wooden tabletop, so we fill in the region under the mouse with wooden grain. However, in the box around the mouse we might accidentally catch the corner of the keyboard, and then that little corner will be replicated as part of the texture. Choosing as a reference a piece of the table that shows nothing but wood grain does the trick.

A really troublesome situation can occur if we're replacing an object that straddles two types of background, as in Figure 7 (next page). In this example, we should replace the truck's bottom half with texture from the street and the top half with the greenery behind it. Sometimes the texture generator does pretty well with a situation like this, sometimes it doesn't. For the times when automatic methods fail, it would be useful to have a manual tool to break up the region under the candidate and apply different textures to different regions.



6 (a) Source picture. (b) I manually identified the region I want to remove. (c) Filling in this region with a synthetic texture based on the background trees doesn't look good. (d) Basing the synthetic texture on the nearby water looks much better.

7 If we want to remove this truck, we should generate a texture for the upper part based on the greenery behind it and a texture for the lower part (and shadow) from the road surface.

Matchmaker, matchmaker

So far I've limited the search to simple copies of the target image that we haven't changed in any way except for position. Let's see what happens when we relax that restriction.

The first things we'll want to include are of course rotation and scaling. We should be able to match the target no matter what size it is, or how it's been rotated.

We do need to be careful that the target isn't too much smaller than its largest appearance in the source. If we have to scale up the target too much to test it against a region of the source, the target is going to get pretty blurry, and it might not match very well. By the same token, the source instances we're trying to find can't be too small. If a candidate is only a few pixels large in the source, then we're going to have a lot of trouble ensuring that it's a match.

We can include some other important geometric transformations as well. Perhaps the most important is perspective, since that's going to be present in almost any photograph we want to manipulate, and most computer-generated images as well.

We'll also need to include some image-based transformations. Suppose our source image is a photograph taken on a hazy day. Instances of the target that are far away will have less contrast than those that are closer, and their colors will be shifted a bit. Suppose we have a picture of a bunch of cars parked outside, and we want to replace all of last year's models with shiny new ones. If some of those cars are partially or completely shadowed by clouds or buildings, then those regions of the cars will be darker than a uniformly lit car. As human observers, it's perfectly obvious to us that they still should be replaced.

Shadows result from the absence of light, but we can also have problems if our images contain different kinds of light. Suppose we're working with a clothing store that's invested heavily in lots of in-store signs, each of which carries a large and prominent copy of the store's logo. They're now considering changing their logo, and they want to see how the store will look when all the signs have the new logo on them. They bring us a photograph of the store that they've shot from the front door, a picture of their new logo, and ask us to show them how the store would look with the updated signs.

A potential problem here is that lighting could influence the store photograph. There might be sunlight coming in through the big front windows, fluorescent bulbs illuminating most of the store, and halogen bulbs highlighting some the merchandise. Each of these lights has a different white point and a different illumination spectrum, which will affect the color of the logos on the signs. The logo might be red-shifted in one place and blue-shifted in another, even if to the human eye all the logos appear pretty much the same. For the computer to find these matches, we'd want our search to include accommodation for a range of tonal and color shifts.

We have other reasons for accommodating such shifts. To just scratch the surface, we might want to match items in the presence of

- color bleeding (as when a bright red carpet casts a red tint on objects near to it),
- local discolorations (as when a shiny object has a highlight that's missing in the target image), and
- aging (when some dyes get old or are exposed to the sun they tend to fade).

It's important, then, that we detect regions of the source that are close to (but not exactly the same as) the target.

Of course, loading in all these geometric and color transformations makes the searching problem more expensive. Just how much more expensive might come as a surprise.

Better search

Now we've got a lot of dimensions to consider: There's two axes of translation, one for rotation, two axes of scaling, perspective (which we might model as a one-point perspective characterized by amount, and the horizontal and vertical location of the vanishing point), overall brightness, and color shifting (which we could model as shifts in red, green, and blue). That's 12 so far.

The brute-force way to do this search would be to make 12 nested loops. Let's get a rough handle on the numbers. Suppose we have a 512×512-source picture, the color ranges from 0 to 255, and we'll consider 360 degrees of rotation, scaling from 0.5 to 1.5 in each direction (in 100 steps of 0.01), and brightness and perspective measures from 0 to 100. If we take eight steps between values for each measure along each range, then we'll have to make around 2.8×10^{39} tests. Let's assume we have a really fast computer and it can do 100 tests per second; this search would take around 1.5×10^{28} years. Astronomers are estimating the age of the uni-

verse at somewhere around 12.5 billion years (give or take 3 billion). Taking the high-end estimate of 15.5 billion years, this says that our one little search would take about 9.6×10^{19} universe lifetimes to complete. Of course, we could complete this search in only 96 years using parallel computing if we could assign an equal-sized chunk of the search to each of about a billion billion different universes, but since we only have the one universe at the moment, that seems just a bit impractical. A billion billion computers running in parallel would also impose some difficulties; just logging in to each one would take quite a while. Of course, we could hand-optimize the code a bit, but to get this search to run in an hour would require speeding things up by a factor of 460 trillion trillion billion (or is that 460 billion trillion trillion?), which would require a really good programmer.

If brute-force at full resolution doesn't work, let's try working with a simpler problem. Probably the easiest way to get out of this computational nightmare is to chop down all those numbers.

We can make some progress computationally. Suppose that we're looking for red balloons against a blue sky. We don't need to take lots of steps when we're looking at regions with no red in them; in fact, we can skip these regions altogether.

One way to figure this kind of thing out is to simplify both the target and the source and do a series of matching steps at different resolutions, using a multiresolution algorithm. We'd start with simple versions of both images, perhaps just by scaling them way down, and then search those much smaller images for matches with a somewhat looser tolerance. If we find a match, then we use that as the starting point for a more careful search at a slightly higher resolution, and so on, working our way up until we reach the original images at their original resolution.

Other ways exist to reduce the data size describing the image but still capture some of its important features. Both Fourier analysis and wavelet decomposition are mathematical methods that let us compare images at a variety of ranges of scales.

However, I think such a computational approach is all wrong because we're using the computer to do something that people are good at: pattern matching. If I'm looking at a picture and looking for a target image as complicated as the faces of a couple of friends, I can instantly spot them with no difficulty. Waiting for more than a billion billion lifetimes of the universe to pass for the computer to come to the same conclusion using brute force seems like a poor use of resources.

A better matching algorithm, then, is human driven. We can do extremely well with just one piece of human input. Ask users to tap once with their pen or mouse in roughly the center of each instance of the target in the source. Then we can do a more localized search using that as a starting point. Again, we can use multiresolution methods to start the search coarsely at first and then refine our match with a series of ever-finer searches.

This is a huge step forward, but with a little more effort we can do even better. Users can provide a complete multidimensional starting point. To make a match, they can drag a copy of the target over the source, drop it in about the right place, quickly scale and rotate it into position, and then do a little color shifting if necessary to make it look like the source. The computer can then refine that guess with a fine-resolution search in the neighborhood of the initial input, tweaking each of the parameters for the best fit.

We don't need to sit and wait for each refinement—we can give the machine a bunch of starting points, go off to lunch or do other work, and let the machine crunch away.

The big value here is that we actually know that there's a match to be found, so we're never wasting time in completely unproductive searches. We can use a simple and greedy maximum-descent algorithm, repeatedly looking for parameter tweaks that make the matches better and accepting them, rather than looking all over each one's entire range for the best place to start looking.

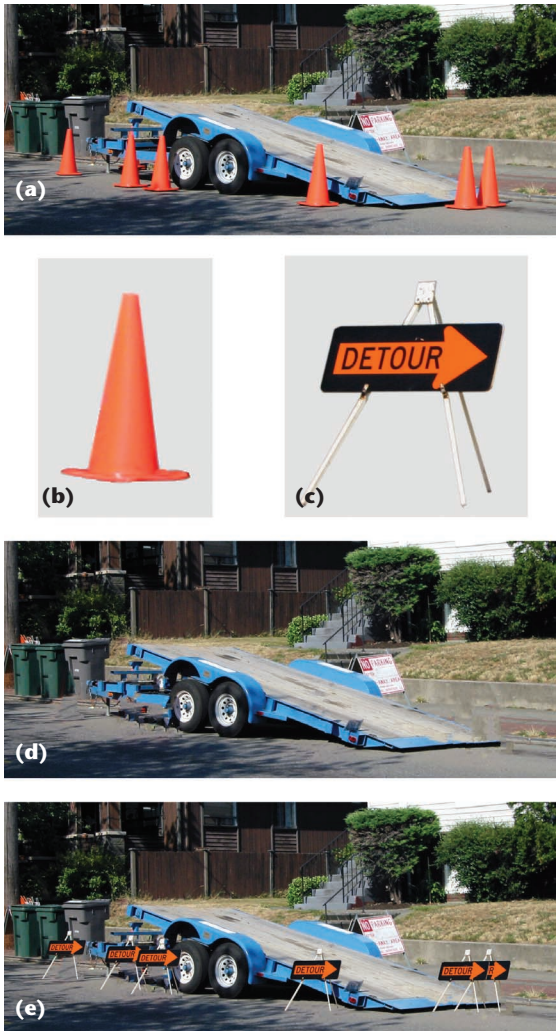
This human-seeded approach works well, and I've found that even with my unoptimized code I can zero in on a good match in just a minute or less given a good starting hint. When using the program, we don't need to sit and wait for each refinement—we can give the machine a bunch of starting points, go off to lunch or do other work, and let the machine crunch away.

You can also specify the quality of the match you require on each hint. For example, if you're replacing a logo on a department store sign, it's probably important to have correctly rotated and sized high-quality matches. However, if you're doing something more casual, like replacing round red balloons with long yellow ones, then you might be willing to accept a much cruder match. As long as the new balloon roughly matches the old one, that's good enough. Such a match can of course be found much faster. In fact, you can lower the search quality to zero, which means that your hint is used immediately as the match.

While you're identifying candidates, you can also provide the region of the source image (or another image) that should be used as the reference texture for filling in the hole.

The human-driven version of the algorithm goes like this:

1. Create a list of seeds by dropping copies of the target on the source, applying geometric and color transformations as desired to make the instances match the source.
2. Tweak a user-provided seed to get a match that's as good as the user requested.
3. Remove this instance of T from the source image and fill in the hole with a synthetic texture.



8 (a) Original image of the trailer, (b) orange cone target, (c) detour sign replacement, (d) cones erased, and (e) final replacement.

4. Draw the replacement image *R* over the spot where the algorithm found the match.
5. If there are seeds yet to be handled, return to step 2, or else quit.

Some examples

Let's look at a few examples.

In Figure 8a I took a picture of a trailer with a few orange traffic cones around it. Let's isolate one of those cones, as in Figure 8b, and replace each instance of it with the detour sign of Figure 8c. You can see the result of the erasure in Figure 8d. In general, this looks pretty good, but you can see artifacts in the texture replacement if you look closely. This is largely due to the simplicity of my texture synthesis routine. A more sophisticated algorithm could probably do a better job of filling in the holes. Once it fills in the holes, the algorithm scales the detour signs and places them at the appropriate locations, giving us Figure 8e.

In Figure 9a I started out with a candle display from a local store. I want to replace the medium-sized green candles of Figure 9b with the soft panda heads of Figure 9c. You can see the result of the removal in Figure 9d. This task stresses my texture synthesizer pretty badly, but we'll cover up most of the problem areas when we insert the pandas. Figure 9e shows the new store display.

9 (a) Original candle display, (b) candle target, (c) panda head replacement, (d) removal of the candles, and (e) final replacement.

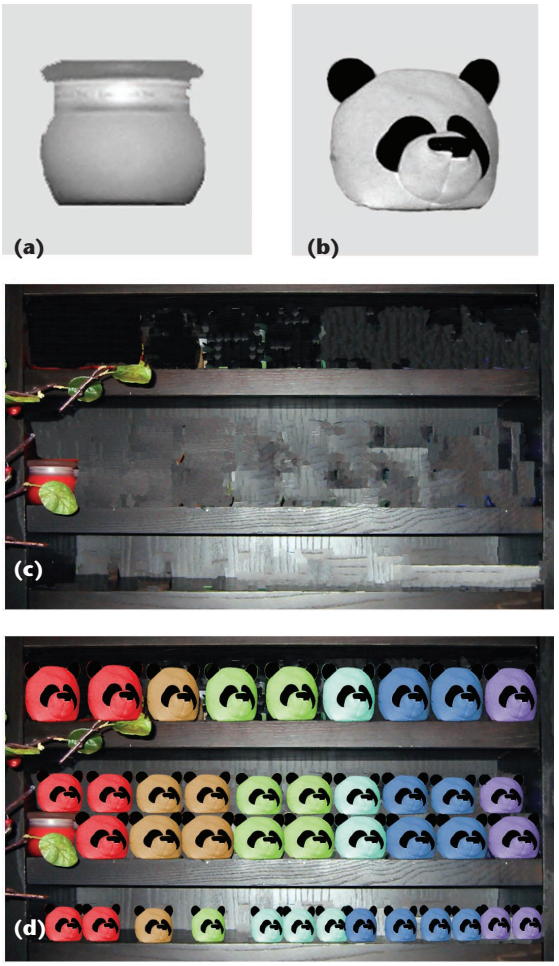
We can do a little better with this by adding the ability to handle color shifts. In Figures 10a and 10b I converted the green candle and the panda's head into black-and-white versions. The search algorithm looks for this black-and-white candle in a black-and-white version of the source.

You'll notice in Figure 10c there's one candle I just couldn't match accurately because of the occluding leaf. The synthetic texture here leaves a lot to be desired, but I think a more robust implementation

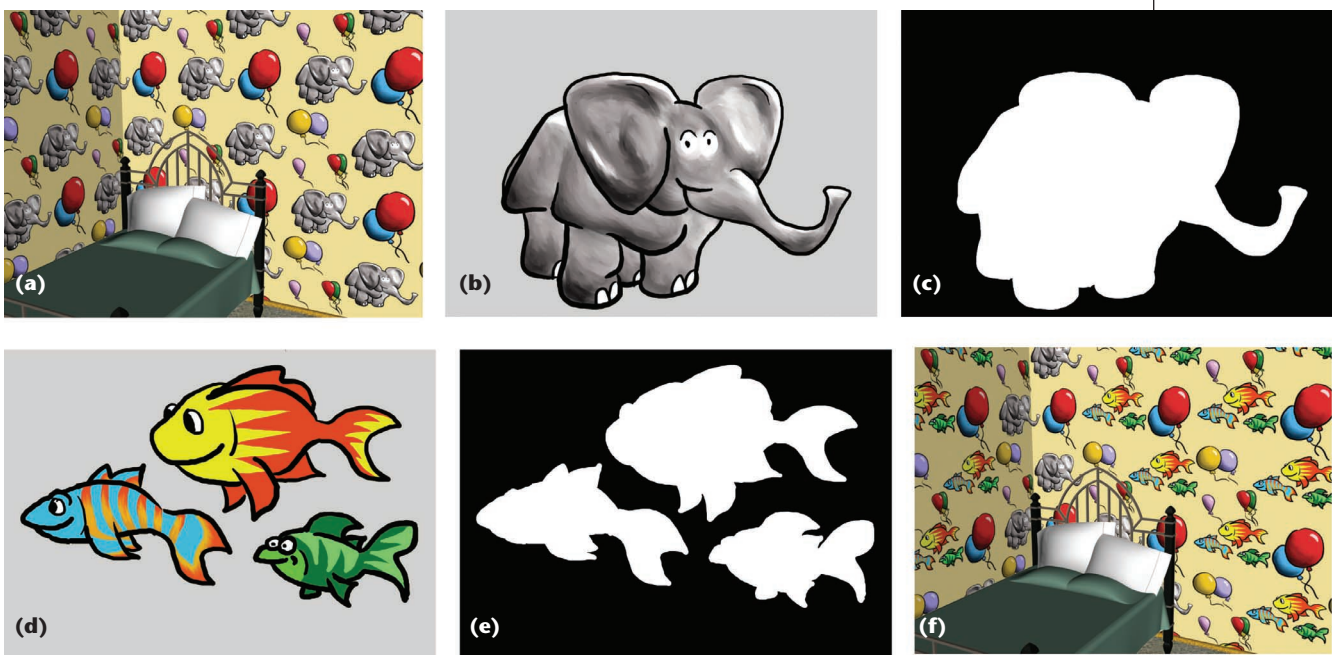
could do a far better job. Even so, once the panda heads are in position the whole thing works well enough to let us see the results. Figure 10d shows our new line of colored, scented panda heads.

Finally, let's look at a 3D rendered example. In Figure 11a I made a simple bedroom with a festive wallpaper pattern. Suppose we'd like to see what this looks like if we had fish instead of elephants. Of course, if we had the original 3D scene around we could just re-render it, but suppose we don't have the 3D model available. In Figures 11b and 11c I show just the elephant and his mask. Note that the elephant isn't distorted by perspective or rotation as he is in the wallpapered room. Because I still had the original elephant that I drew, I just grabbed that image and used it. It's possible with modern image-editing programs to correct for distortions like perspective, so even if I didn't have my original I could get a pretty good target working from the image. I could also use one of the elephants directly off the wallpaper with no correction. In that case I'd need to do a second search-and-replace to handle the elephants on the left wall, which are distorted somewhat differently. I drew the fish of Figure 11d to replace the elephant. Their mask is in Figure 11e. The final result is Figure 11f.

There are a couple of things to note here. First, my input to the system was to drop the target elephant on each occurrence in the source. Then I manually gave the system a pretty good starting hint for the perspective distortion. Second, the system computed the average change in luminance over all the pixels that were being removed and applied that change to the replacements. Thus the fish on the left wall are a little darker than the fish on the right wall, as they should be to fit



10 Color shifting objects. (a) Black and white version of a green candle, (b) black and white version of a panda's head, (c) an undetected candle, and (d) panda heads with color from the green candle



11 (a) Wallpapered room, (b) target elephant, (c) target elephant mask, (d) fish replacement, (e) fish replacement mask, and (f) final replacement. Note that the fish are darker on the left wall, following the overall darkness there. Note also that some of the elephants weren't replaced.

the overall tone. Third, the elephants near the top of the room were present enough that I could match to them, even though in some cases not much of the corresponding fish replacements showed up in the result. Finally, some elephants didn't get matched. Three of these elephants are hiding behind the bed: one in the lower right, one above the bedspread on the left wall, and one behind the headboard. Two of the elephants cross over the corner of the room, so their body is on the left but a piece of their trunk is on the right. These didn't get replaced because the fish would need the same visible bend to look good. Finally, two of the elephants—just to the right of the corner—were also left alone. That's because although the elephant doesn't straddle the corner, the fish would have, and it would look wrong.

Wrapping up

One thing I'd like to add to this system is the ability to identify and replace obscured matches. For example, sometimes an instance is recognizable, even though it's partially hidden by some other object. Often even an obscured object is visible enough to be recognizable to the human eye, like the medium-sized candle at the far left of Figure 9. It would be nice to capture the match and then use the same occlusion on the replacement so that it's blocked the same way.

Shadows pose an interesting problem. Suppose that a shadow falls halfway across an instance of the target. We might be able to match this, but the algorithm I presented doesn't replicate the shadow in the replacement. With a little more work, I think we could analyze each match to find how it deviates from the source and then apply those deviations to the replacement. This way, we'd maintain not only the original shadows in source, but also highlights, reflections, and other surface variations.

Taking a cue from some modern image-editing programs like Procreate's Painter and Adobe's Photoshop, it would be fun to have a variety of replacement images for each target. For example, we might have several images of the panda head in Figure 9e, and have each replacement use one of those at random (or let the user choose them), so that the pandas aren't all identical to one another. And we needn't use just pandas; if we wanted to replace the candles with a variety of toys, we could fill the shelves with a collection of different toys simply by picking different replacement images for each candle.

I'm intrigued by how much speedup I got from moving the matching problem from the computer to the person. Of course, I'd like an automatic program that I could start and then ignore as you can do with slow Photoshop filters on big pictures, for instance. I like the idea, though, of harnessing the amazing power of the human visual system and brain to make a staggeringly slow and complex process relatively fast and easy. ■

References

1. D. Kurlander and E.A. Bier, "Graphical Search and Replace," *Proc. Siggraph 88*, ACM Press, 1988, pp. 113-120.
2. J. Flint, "TV Football's MVP—Yellow First-Down Line," *The Wall Street Journal Online*, 26 Jan. 2000.
3. D.J. Heeger and J.R. Bergen, "Pyramid-Based Texture Analysis/Synthesis," *Proc. Siggraph 95*, ACM Press, 1995, pp. 229-238.
4. J. Portilla and E.P. Simoncelli, "A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients," *Int'l J. Computer Vision*, vol. 40, no. 1, Oct. 2000, pp. 49-70.
5. P. Harrison, "A Nonhierarchical Procedure for Resynthesis of Complex Textures," *Proc. 9th Int'l Conf. in Central Europe on Computer Graphics, Visualization, and Computer Vision*, Winter School of Computer Graphics (WSCG), Feb. 2001, http://www.csse.monash.edu.au/~pfh/resynthesizer/.
6. L. Wei and M. Levoy. "Fast Texture Synthesis Using Tree-Structured Vector Quantization," *Proc. Siggraph 00*, ACM Press, 2000, pp. 479-488.
7. H. Igehy and L. Pereira, "Image Replacement through Texture Synthesis," *Proc. IEEE Int'l Conf. Image Processing* (ICIP 97), IEEE CS Press, 1997, pp. 186-189.

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