Exploded Images

Arcady Goldmints-Orlov
Brown University

December 2010

1 Introduction

The Exploded Images program offers a novel approach to image manipulation, wherein objects in the image are automatically detected[2] and cut out from the background. The "exploding" is similar to the notion of an exploded-view diagram [3], except it is performed for a photograph rather than a diagram. The holes in the background that are left from cutting out the objects are filled using an inpainting algorithm[1]. A user interface is then presented that allows the user to move the objects around the image independently. Both the object detection and inpainting use existing algorithms, and indeed largely use existing code: the novelty is in combining them in this fashion. This combination can provide new ways of presenting and manipulating images, by allow the user to move or remove individual objects in them.

2 Boundary Extraction

The boundary extraction algorithm is an implementation of [2], which attempts to recover occlusion boundaries from an image, that is, it tries to classify the image pixels into objects delimited by boundaries where one object is occluding another. The algorithm works by first constructing an over-segmentation of the image into considerably more segments than the final result, then iteratively merging them based on various cues that include properties of the boundaries, regions, and the geometric context. The geometric context includes classifying objects as one of five categories (horizontal support, vertical planar, vertical solid non-planar, vertical porous, and sky), and reasoning about the juxtaposition of different categories of objects at boundaries and boundary junctions. The geometric context is also used to derive depth estimates, by finding where the lowest point of an object touches a horizontal support. That point may be occluded, in which case a range of possible depths is estimated.

The result is a labeling of each pixel with the region it belongs to, as well as the probability that each region belongs to each of the five geometric categories. The highest-probability "ground" and "sky" regions are then combined to form the background layer, and each other region is taken to be a foreground object. The holes in the background that are left by cutting out the foreground objects are then filled using the inpainting algorithm described in the next section. No depth ordering information is used, but it could be estimated using techniques similar to those in the boundary extraction algorithm. Even with depth ordering, filling in occluded regions in foreground objects would be considerably more difficult, as that would require estimating the occluded portion of that object's boundary.

3 Inpainting

The inpainting algorithm is an implementation of "Region Filling and Object Removal by Exemplar-Based Image Inpainting" by Criminisi et. al.[1]. The algorithm takes an image and a mask that defines which pixels are to be filled. We shall refer to these pixels as the "hole," though they do not need to be contiguous. The hole is then progressively filled with square patches of pixels taken from the source image. The patch to be filled is selected using a priority function that combines a confidence term C, which tends to fill in "bays" that are mostly surrounded by filled areas, and a data term D, which tends to extend linear structures into the filled region. The algorithm is similar to exemplar-based texture synthesis, except with one image acting as both the exemplar and texture being synthesized.

The algorithm first calculates the perpendicular gradient $\nabla I_p^{\perp}$ for each pixel in the source image, and assigns a confidence value $C(p) = 1$ for each pixel not in the hole. The algorithm then iterates until the hole has been filled entirely. On each iteration, the current boundary of the hole $d\Omega$ is computed. For each pixel $p \in d\Omega$, a confidence term term $C(p)$ and a data term $D(p)$ are calculated and then multiplied to obtain that pixel's priority $P(p) = C(p)D(p)$. The confidence is defined as

$$C(p) = \frac{\sum_{q \in \Psi_p} C(q)}{|\Psi_p|}$$

that is, the sum of the pixel confidence values from the part of the patch that is not in the hole, divided by the total area of the patch. Thus, the $C(p)$ will be higher the more pixels in the hole are already filled, and will tend to decay further from the original hole boundary. The data term is defined as $D(p) = \frac{|\nabla_p^{\perp} \cdot \mathbf{n}_p|}{\alpha}$, where $\mathbf{n}_p$ is the normal of the hole boundary at p and α is a normalizing factor for the pixel values. Thus, is large where strong gradients of color are perpendicular to the hole boundary. The target patch that is to be filled is the one centered on the pixel with the highest priority value.

Having found the patch we want to fill, the algorithm then finds the most similar patch in the source image. Similarity is defined by taking the sum of squared differences between pixels that are already filled in the target patch and the corresponding pixels in every other patch in the image. Our implementation rejects source patches that would not result in all the corresponding hole pixels in the target patch being filled. The hole pixels in the target patch that are then simply copied from the source patch, along with the isophote values. The value of $C(p)$ assigned to the filled pixels in the target patch is the value that was found for the center pixel earlier. A pseudocode version of the algorithm is presented below.

There are a couple of implementation details worth noting about this algorithm. The first is that the total run time depends on how many total patch fill operations are done, which depends primarily on the patch size, with larger patch sized resulting in much faster fills. The other is that the ssd operation of finding the most similar patch dominates the run time, taking between one half and three quarters of the total. The naive implementation of calculating the ssd for each patch is even slower, but an optimization can be made when calculating it for the whole image at once. The operation can be treated as sum of convolutions and these convolutions can be computed using a two dimensional Fast Fourier Transform (FFT), which speeds things up considerably.

4 Results

The segmentation and texture filling algorithm were combined in a pipeline together with a small UI that shows the segments The segmentation and texture filling algorithm were run on a test set

$C(\Omega) = 0$
$C(\Phi) = 1$
while $\Omega \neq \emptyset$ **do**
  for $p \in d\Omega$ **do**
    $C(p) = \frac{\sum_{q \in \Psi_p} C(q)}{|\Psi_p|}$
    $D(p) = \frac{|\nabla_p^{\perp} \cdot \mathbf{n}_p|}{\alpha}$
  end for
  $t = \arg\min_{p \in d\Omega} C(p)D(p)$
  $s = \arg\min_{p \in \Phi} \mathrm{ssd}(\Psi_p, \Psi_t)$
  for $p_{i,j} \in \Psi_t \cap \Omega$ **do**
    $p_{i,j} = s_{i,j}$
    $C(p) = C(t)$
  end for
  $\Omega \leftarrow \Omega \setminus \Psi_t$
end while

of 21 photographs from the Flickr photo-sharing site (`http://www.flickr.com`). Each image was scaled down to a maximum dimension of 512 pixels and used to produce four sets of results, as the segmentation algorithm has some randomized parts and produces different results on each run. The segmentation algorithm performs with varying degrees of success by subjective standards, but the four results for each image generally contain at least one "good" segmentation and at least one "bad" one that misses key objects. The inpainting tends to work well for certain cases, especially large patches of grass, but fail in others.

Figure 1: A simple exploded image

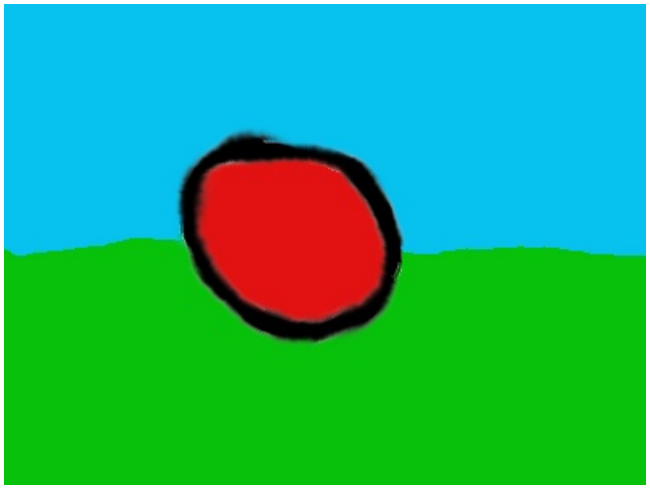

Figure 2: The original image from which Figure 1 was computed

A simple example of an exploded image is given in Figure 1, with the original image in Figure 2. In this rather simple synthetic test case, the segmentation algorithm performs quite well, identifying the background and single object quite accurately. The inpainting algorithm works reasonably well, extending the horizon most of the way into the occluded area, but as it gets further from the original image, it gets less accurate, resulting in the protrusion of green into the sky. Note that the inpainting algorithm operates purely on the local scale of a patch (in our case, 15×15 pixels), and has no knowledge of global image structure, so such structure is less likely to be preserved further away from the edge of the hole.

A more complex and realistic example is given in Figure 3, with the original image given in Figure 4 for comparison. The background with the holes filled is in the center, and around it are the individual objects identified by the boundary extraction algorithm. The algoritm does identify the tombstone that is in the center of the image fairly accurately. The other tombstone is also identified, but that segment includes some spurious background. There are also a few segments that subjectively appear to be background, but were picked up as discrete objects. There are also a few very small patches of sky that were identified as objects. A potential future improvement would be to find a way to more accurately label multiple sky and ground segments and distinguish them from the foreground objects we're interested in.

The inpainting algorithm has filled most of the background with a plausible leafy texture, which merges fairly nicely with the grass. However, there are very noticeable artifacts where the grass meets the sky and in the sky itself. The sky artifacts are likely due to the fairly small area of sky in the original image, which leads few choices of sky patches to copy, and leads to the artifacts. An image with more sky or use of a smaller patch size would have fewer artifacts in the sky region. The grass-sky boundary artifacts are due to the fact that the original background image contains no patches with both grass and sky texture, so the inpainting algorithm can find no good patches from which to copy such a boundary. Each patch on the boundary ends with either grass or sky, not both, and the boundary shows artifacts of the fill order.

The problem of boundaries between ground and sky is a significant one for photographs with most of the horizon obscured, which is a common condition in our dataset. In these images, most of the area around the horizon gets cut out from the background, leaving very few horizon patches for the inpainting algorithm to use, which leads to blocky artifacts on the horizon as patches are chosen

Figure 3: A more complex exploded image

Figure 4: The original image from which Figure 3 was computed

from either all-ground or all-sky. A potential solution to this is to use patches with irregular edges, rather than squares, which would at least make the artifacts look less blocky, though it would still mean a hard edge between the ground and sky patches.

Finally, in Figures 5 and 6 we present two exploded images generated from the same source image. Because the segmentation algorithm uses some randomization, repeated runs on the same input can produce different results. Often, only some of these results are "good" segmentations, while others, such as the example in Figure 5 end up breaking up single objects among different segments, or not including all of an object in a segment. In both cases, the inpainting performs fairly poorly. In this case, the problematic area is the smoothly shaded wall, which the inpainting handles poorly compared to a textured area like grass or a solid color. In this case, the blending that would have blurred the sharp texture of the grass would likely provide a better result, without the square patches being as visible.

The performance of the image processing is still slower than is desirable for interactive use, with even the scaled-down images taking on the order of 10 minutes for the segmentation, and another minute or two for the inpainting. However, once the segmentation is completed, the user interface allows the image segments to be manipulated interactively, by sliding them around the image. There is a big performance tradeoff in the inpainting algorithm between patch size and speed. A larger patch size leads to worse looking results, but is considerably faster, as the algorithm has to look up fewer patches, while a smaller patch size is slower, but potentially more accurate, with the authors of the algorithm recommending a 9×9 patch size.

References

[1] Antonio Criminisi, Patrick Perez, and Kentaro Toyama. Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on Image Processing*, 13(9):1200–1212, September 2004.

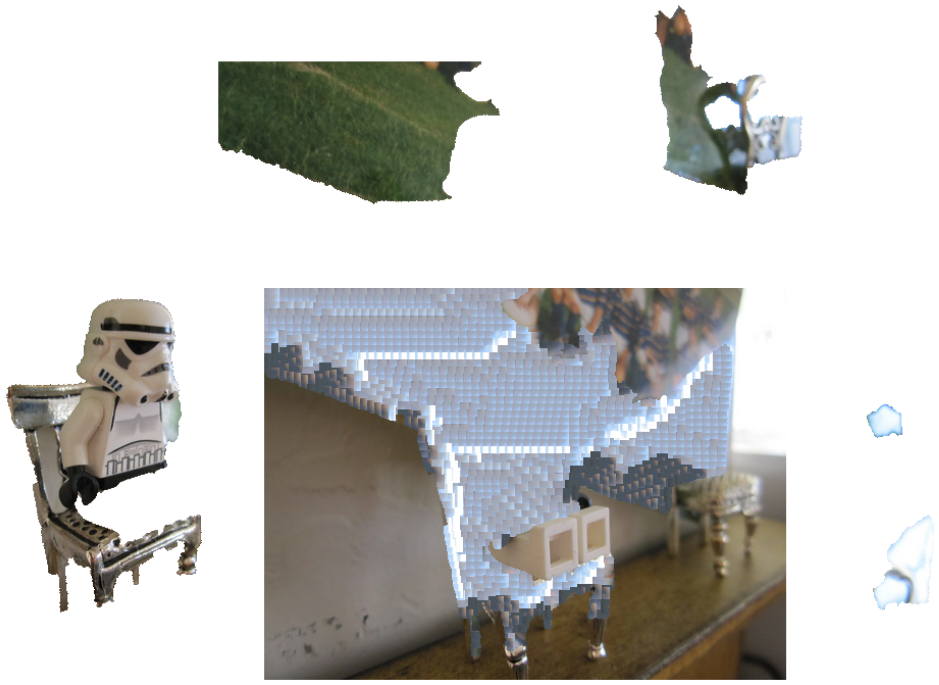

Figure 5: An example where the algorithms perform suboptimally

[2] Derek Hoiem, Andrew Stein, Alexei A. Efros, and Martial Hebert. Recovering occlusion boundaries from a single image. In *International Conference on Computer Vision (ICCV)*, October 2007.

[3] Wilmot Li, Maneesh Agrawala, Brian Curless, and David Salesin. Automated generation of interactive 3d exploded view diagrams. *ACM Trans. Graph.*, 27:101:1–101:7, August 2008.

Figure 6: Another exploded image produced from the same source image

Figure 7: The original image from which Figures 5 and 6 were produced.