Pacific Graphics 2008
T. Igarashi, N. Max, and F. Sillion
(Guest Editors)

Volume 27 (2008), Number 7

Shrinkability Maps for Content-Aware Video Resizing

Yi-Fei Zhang[1] Shi-Min Hu[1] Ralph R. Martin[2]

[1]Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[2]School of Computer Science, Cardiff University, UK

Abstract

A novel method is given for content-aware video resizing, i.e. targeting video to a new resolution (which may involve aspect ratio change) from the original.
We precompute *a per-pixel cumulative* shrinkability map *which takes into account both the importance of each pixel and the need for continuity in the resized result. (If both x and y resizing are required, two separate shrinkability maps are used, otherwise one suffices). A* random walk *model is used for efficient* offline *computation of the shrinkability maps. The latter are stored with the video to create a* multi-sized video*, which permits arbitrary-sized new versions of the video to be later very efficiently created in real-time, e.g. by a video-on-demand server supplying video streams to multiple devices with different resolutions. These shrinkability maps are highly compressible, so the resulting multi-sized videos are typically less than three times the size of the original compressed video. A* scaling function *operates on the multi-sized video, to give the new pixel locations in the result, giving a high-quality content-aware resized video.*
Despite the great efficiency *and* low storage requirements *for our method, we produce results of comparable* quality *to state-of-the-art methods for content-aware image and video resizing.*

Categories and Subject Descriptors (according to ACM CCS): I.4.10 [Computing Methodologies]: Image Processing And Computer Vision

1. Introduction

With the rapid growth of output devices with widely differing resolutions and processing power, adaptive resizing of images and video is of increasing relevance. A good resizing algorithm should be fast and preserve the important content in an image or video while retaining spatio-temporal coherence of video sequences.

When transferring video to a new resolution, simple methods have clear drawbacks. Scaling the output in x and y can be performed in real-time using linear or higher-order interpolation. However, doing so causes distortion if the output aspect ratio differs from the input aspect ratio. Furthermore, even if the aspect ratio remains unchanged, if the output resolution is much lower than the input resolution, important information may be lost if simple scaling is performed (e.g. text in the image or video may no longer be readable).

A second approach is to use some form of cropping. The simplest is to crop the output to the center of the input frame. This can be performed in real-time and is typically supported by modern wide-screen televisions. However, this fails when objects of interest are too far off-center. Cropping is also inappropriate if the output resolution is significantly lower than the input resolution, as too much information of interest may be discarded. More sophisticated cropping approaches, such as pan-and-scan methods, usually require human intervention to select the most appropriate portion of the scene, especially if high-quality results are to be produced. Such an approach is also usually targeted at a single specific output resolution, and again fails if the information of interest does not fit within the cropping rectangle.

Recently, Wolf et al [WGCO07] proposed a method for content-aware video resizing. It produce much more satisfactory results than cropping or uniform scaling methods, in terms of retaining the information of interest. However, a major disadvantage of this method is that it is slow. Speed is particularly important for video resizing algorithms, due to the large amount of data.

We consider the particular case where a video-on-demand server must supply video streams in real-time to devices of many different output resolutions, which cannot necessarily be predicted in advance. This precludes the precomputation of a fixed number of videos at known new resolutions, using e.g. high quality pan-and-scan techniques.

Two approaches could in principle provide the new video data at the required rate. Firstly, a sufficiently fast algorithm could resize video in real-time. This would seem difficult to achieve, given the amount of data to be processed, especially if a server is serving multiple streams of video; a variant of the approach in this paper might be feasible if the multigrid computations required were performed on a GPU.

More realistically, a video could be preprocessed offline to produce some supplementary data to be used in conjunction with the original video to enable video streams to be resized in real-time. The combination of the original video with such supplementary information can be called a *multi-sized video*. Clearly, the extra storage required must not be too great: it should be commensurate with the original video size.

While this paper was under review, Rubinstein et al [RSA08] also published a method which can produce multi-sized video, allowing real-time video resizing after precomputation. However, their method has several drawbacks: (i) the additional storage required in relation to the original compressed video is very high, (ii) it is extremely slow, and (iii) it only supports multi-sized video which allows resizing in width *or* height, but not both.

In this paper, we present an algorithm which, after preprocessing to produce a multi-sized video, allows for content-aware video resizing to be performed on-the-fly, even allowing running video to be *dynamically* resized if desired. Our preprocessing method is *efficient*, and our multi-sized videos take *little additional space*.

Our approach precomputes and stores a *cumulative shrinkability map*, which takes into account both the importance of each pixel and the need for continuity in the resized result. (If both x and y resizing are required, a separate shrinkability map is needed for each direction, otherwise one will suffice). The importance of pixels comes from an importance map, which for simplicity we compute in the same way as [WGCO07], as it is not the main concern of this paper. In real-time, a *scaling* function takes the cumulative shrinkability map, and the target video size, to derive final pixel positions and sizes in the target video. These are then used to control a texture mapping process, generating output pixels from the source frames, following the approach in [Wol90]. The novelty of our paper lies in the shrinkability map concept, and the scaling function. Use of a *random walk* model allows *efficient* computation of the cumulative shrinkability map. Shrinkability maps are also highly compressible, and our multi-sized videos are typically less than three times the size of the original video (using both x and y direction cumulative shrinkability maps). Scaling in real-time has very low computational cost, allowing a video server to readily generate multiple different resolution streams of a video-on-demand for transmission.

2. Related Work

The range of devices capable of displaying video has increased rapidly over recent years, from mobile phones and portable media players to high-definition televisions, and beyond. Thus, resizing videos both in dimension and aspect ratio is an important topic. (We only consider *spatial* resizing, not adjustments to the *running time* of the video).

Simply *uniformly* scaling video to the target size is not 'content-aware', ignoring the varying importance of different areas, and their changes over time.

A second class of approach is based on *cropping*, or pan-and-scan. Such papers as [LG06, TJS07] consider the problem of selecting an optimal rectangle from an input image or video, where optimality is defined in terms of an importance or salience function; continuity requirements must also be considered. Unfortunately, cropping alone often loses too much of interest, particularly when important detail is located towards more than one edge of the scene.

Thirdly, Wolf et al [WGCO07] give a video retargeting method which is constrained to preserve the shapes of important regions. It does so by using a non-uniform global warping. However, it use Cholesky decomposition to solve a large sparse least-squares problem, which can be done in real-time only for low-resolution video. Wolf et al do not consider creating multi-sized video, and they only show results which *reduce* video size.

Very recently, Rubinstein et al [RSA08] proposed a video retargeting algorithm which works by removing 2D seam manifolds from 3D space-time volumes. They use a graph-cut approach to find these seam manifolds. Compare to Wolf et al's approach, the quality of the results is not improved, but they do support the creation of *multi-sized video*. However, their method only supports multi-sizing video in one dimension, and both the additional storage and computational time requirements are very high.

Our method is similar in spirit to Wolf et al's but is faster, and it supports the creation of multi-sized video. Our random walk model allows the resizing problem to be solved efficiently using a multigrid algorithm, and in principle it could readily be parallelized for GPU implementation, although we have not yet done this. We can create multi-sized videos by precomputing a cumulative shrinkability map, after which the video may be resized with very little computational effort in real-time, using a scaling function. Our cumulative shrinkability maps are very smooth, so can be highly compressed, meaning that our multi-sized videos are typically less than three times as big as the input video (when using both x and y shrinkability maps).

Random walk models have been used to solve a variety of problems in image processing such as interactive image segmentation [Gra06] and real-time high quality matting [WAC07]. Problems formulated as random walks require the solution of a sparse, symmetric, positive-definite system of linear equations. This may be done efficiently using a variety of methods, as discussed later. Here, we use a random walk model because it is very suitable for modeling the constraints on the shrinkability map, and allows rapid calculatation of the cumulative shrinkability map.

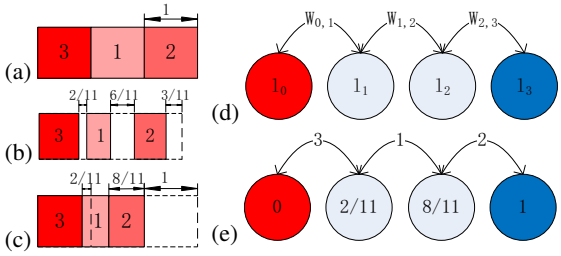


Figure 1: *Reducing the width of a* 3×1 *image by 1 pixel. (a) An assumed importance map E. (b) Shrinkability of pixels according to importance. (c) Repositioning pixels according to target size. (d) Random walk model with red and blue terminal nodes. (e) Weights on edges come from the importance map. Node labels are the computed new positions of the left hand edges of pixels (the cumulative shrinkability map).*

3. Algorithm

3.1. Overview

We now outline our algorithm. We treat resizing of images, or video, as a warping problem, using four steps. The first two steps are carried out as preprocessing, and the second two at run-time. First, an *importance map* is computed indicating the significance of each pixel. Next, using the importance map for each frame, we precompute separate *cumulative shrinkability maps* for the x and y directions and store them, after lossy compression, with the input video, giving a *multi-sized video*. Thirdly, at run-time, a target resolution is input. A *scaling function* uses this and the multi-sized video to assign a new position and a new size for each pixel within each target frame; again, x and y directions are processed independently. Finally, standard *texture warping* maps the image to the target size using the new positions and sizes. The novelty of our algorithm lies in the second and third steps.

In the rest of this section, for clarity of exposition, we consider the case where only a reduction in width is to be performed. If a new height is also desired, two passes of scaling are made, first using the x direction shrinkability map to compute new x positions for each target pixel, then using the y direction shrinkability map to compute new y positions for each target pixel. (If only width reduction is required, pixel y coordinates are unchanged.)

In the first step, for simplicity, we use the same *importance map* as Wolf et al's method, although we note that if high quality resizing results are required (e.g. for a Hollywood movie), it may be desirable, and indeed necessary, to hand-tune or hand-specify the importance map. The importance map assigns a value between 0 and 1 to each pixel where 1 indicates highest significance. The importance map E takes into account the local saliency (L_2- norm of the gradient), any areas detected as faces (faces often convey the most significant information in a scene), and moving areas (moving objects are important since they draw the viewer's attention). Further details can be found in Wolf et al's paper [WGCO07].

We now consider the second and third steps. Earlier methods directly calculate new pixel positions (and implicitly, pixel sizes) based on the importance map and the new target size, whereas we use a two-step approach. Initially, as a preprocessing step, we calculate *shrinkability* for each pixel, to give a cumulative *shrinkability map*. The aim is to *selectively* reduce the width of each pixel. The cumulative shrinkability map gives the new pixel locations in the particular case of reducing the width of the image by *one pixel*. As cumulative shrinkability maps are smooth functions, they can be highly compressed. By storing the compressed cumulative shrinkability maps with the original video, we create a *multi-sized* video, which can then be resized to any target size on the fly.

Subsequently, at run-time, given initial and target widths, we compute the desired reduction in width k, and then a *scaling function* is used to compute new pixel positions from k and the cumulative shrinkability map. Note that k is an input to the scaling function: because the multi-sized video is valid for any target size, and the scaling function is cheap to compute (as is the final texture warping step), we can simultaneously generate multiple streams of video-on-demand with little computational effort required by a server. We may even change the new width as the video is playing, if desired.

Once the new pixel positions are defined, texture warping is performed by well known image resampling techniques [Wol90].

3.2. 1D shrinkability

We now explain the concept of shrinkability maps and their use. We start with a 1D case, an image comprising a single row of pixels (see Fig. 1), and assume that the width of this $w \times 1$ image I is to be reduced by 1 pixel.

The basic idea is to shrink different pixels by different amounts, according to their importance given by the importance map. We define the *1D-shrinkability* of a pixel as the desired reduction in that pixel's width when reducing the 1D image width by 1 pixel. If $s(i)$ is the shrinkability of pixel i, the concatenation of these values is the shrinkability map.

For this simple 1D case, we can analytically solve the problem to find the $s(i)$:

$$s(i) = 1/(E(i)\sum_{j=1}^{w} 1/E(j)), \quad (1)$$

where $E(i)$ is the importance map value for pixel i.

However, we now consider an alternative approach using a random walk model to find shrinkability, as this extends more readily to the 2D (image) and 3D (video) cases.

A general random walk model can be formulated as follows. Suppose G is a weighted graph with certain designated terminal nodes (see Fig. 1(d)). Each node i has an associated label $l(i)$. The labels of terminal nodes are predefined in some way, while labels for other nodes are computed by random walks. Suppose a walk starts at a non-terminal node. At each step, it randomly moves to a neighboring node according to given probabilities, until it reaches one of the terminal nodes. The probability of the walk going from node i to node j is given by $p(i,j) = W(i,j)/\sum_k W(i,k)$, where $W(i,j)$ is the weight on edge (i,j). If many random walks start at node i, each reaches some terminal node with value $T(i)$. We set the label $l(i)$ for the starting node to the average value of $T(i)$ over all walks, in the limit that the number of walks tends to infinity. To compute the $l(i)$, we note that

$$l(i) = \sum_j p(i,j)l(j) \quad (2)$$

which leads to a linear system; its solution is considered in Section 3.4.

We now return to the 1D shrinkability problem. To determine the cumulative shrinkability map, which corresponds to reducing the width w of the image by 1 pixel, we construct a linear random walk graph with $w+1$ nodes. Nodes 0 and w are the terminal nodes and are given values of zero and one, respectively. The values for nodes 1 to w correspond to the cumulative shrinkability u (see Fig. 1(c)) at pixels 1 to w respectively (see Fig. 1(e)); note that $u(0) = 0$ by definition. The cumulative shrinkability $u(i)$ is related to the shrinkability $s(i)$ by:

$$u(i) = \begin{cases} 0 & \text{if } i = 0, \\ u(i-1) + s(i) & \text{otherwise.} \end{cases} \quad (3)$$

Because node w is initially allocated a cumulative shrinkability of 1, the overall image width will be reduced by one; cumulative shrinkability of all other pixels is determined by the random walk.

We now consider choice of weights on the graph edges. To selectively preserve content, more important pixels should have lower shrinkability. In the 1D case we may simply set the weight $W(i-1,i)$ for the edge between nodes $i-1$ and i to the importance map value $E(i)$. This ensures that the larger $E(i)$, the closer the value of $u(i-1)$ to $u(i)$, and hence the lower the shrinkability of pixel i. Assigning the weights in this way leads to the same solution as in Eqn. 1. (See Fig. 1(e))). Having solved the random walk problem to give cumulative shrinkability values, the shrinkability map entries can be simply found by $s(i) = u(i) - u(i-1)$.

3.3. 2D and 3D shrinkability

We now turn to computing 2D (image) and 3D (video) cumulative shrinkability maps. We first consider resizing a 2D *image* in width by 1 pixel. We define the *x-axis-2D-shrinkability* $s(x,y)$ as the shrinkability of pixel (x,y) when reducing the image width by 1 pixel. We also define $u(x,y)$ to be the cumulative shrinkability of all pixels to the left of pixel (x,y) in the same row. If we simply built a 1D random walk model for each row of the image independently, it is clear that this would degrade the image content by creating a zigzag effect [AS07]. Thus, we make a 2D random walk model whose rows are those of the separate 1D random walks, but with vertical edges also added in order to preserve continuity as much as possible between rows. To do so, $u(x,y)$ should approximately equal $u(x,y-1)$. Distortion of more important areas should also be lower, i.e., the larger $E(x,y)$, the more similar $u(x,y-1)$ should be to $u(x,y)$. This can be achieved by setting weights for vertical edges in the random walk model to

$$W(u(x,y),u(x,y-1)) = K_1 + K_2 \times E(x,y) \quad (4)$$

where K_1 must be positive in order to meet the first requirement above and K_2 positive to meet the second.

We next additionally take time into account. We define *x-axis-shrinkability* $s(x,y,t)$ as the shrinkability of pixel (x,y) in frame t when resizing the video width by 1 pixel. A 3D random walk model is constructed based on the time varying importance map $E(x,y,t)$. A 2D random walk is separately built for each frame as before, and these are connected by edges between corresponding pixels of successive frames into a 3D random walk model (we assume the video has not changed to another shot, when coherence is irrelevant). Suppose P and Q correspond to pixels at the same (x,y) location in successive frames, and W_{PQ} is the weight of the edge joining P and Q. The weights of such edges are given by :

$$W(u(x,y,t),u(x,y,t+1)) = K_3 + K_4 \times E(x,y,t) \quad (5)$$

using the same reasoning as when extending from the 1D to 2D case.

Parameters $K_1,\ldots,K_4$ control coherence from row to row, and between frames. We suggest these values should be $K_1 = K_3 = 1$ and $K_2 = K_4 = 0.2$, which work well for normal pictures and videos. Setting $K_1 = K_3 = 1$ gives equal weights to providing spatio-temporal coherence, and to keeping the most important pixels. If retaining important regions is more important than coherence, K_1 and K_3 can be set lower.

3.4. Solving the random walk model

The random walk model requires the solution of the linear system in Equ 2. For example, for the 1D random walk model, we have:

$$u(i) = \frac{E(i)u(i-1) + E(i+1)u(i+1)}{E(i) + E(i+1)} \tag{6}$$

for $i = 1, 2, \ldots (w-1)$, and $u(0) = 0, u(w) = 1$.

A more detailed discussion of the construction and properties of the linear systems associated with random walks can be found in [Gra06]. The fastest way to accurately solve such a linear system is to use a direct solver based on sparse Cholesky decomposition [BBK05]. However, since pixels are ultimately placed at integer locations, a high-precision solution is not required. Thus, instead we use an iterative multigrid solver which is faster but less accurate. Such solvers are the most effective iterative solvers for this kind of problem [Gra06].

Although we have not yet implemented such a scheme, multigrid solvers can be efficiently parallelized to use a GPU [BFGS03]. We note that such an approach provides the potential capability to achieve video resizing in real-time by computing cumulative shrinkability maps as required, rather than using a precomputed multi-sized video.

Directly solving the full 3D model in one go would require too much memory, so we in practice use an approximation to our 3D random walk model. A 2D random walk is initially solved for the first frame, treating it simply as an image. For all other frames, we treat the problem as a two-layer random walk, where the previous frame gives constraints on the current frame, determined as described; we ignore any links to the succeeding frame. This may be justified on the basis of causality. This simplified method requires much less memory and computational effort.

3.5. Scaling function

We now consider how to use the x cumulative shrinkability map to shrink a video of width w not just by 1 pixel, but by k pixels. Using the random walk framework, to do so, we would simply set the value of the right hand terminal nodes to k instead of 1. However, linearity of the random walk model tells us that the solution to this problem simply results in new cumulative shrinkability values which are each also multiplied by k. However, there is a small problem. Certain shrinkability values may now exceed 1. The interpretation of such values is that the right-hand edge of the pixel involved is moved leftwards past its left-hand edge. As a result, pixels can change order locally in the final output, potentially causing undesirable artifacts.

We can overcome this difficulty by using a *scaling function*. Let $s_k(x,y,t)$ be the shrinkage for each pixel when shrinking the video width by k pixels. It should depend only on $s_1(x,y,t)$ and k:

$$s_k(x,y,t) = S(k, s_1(x,y,t)). \tag{7}$$

The approach in the previous paragraph would result in

$$S(k,s) = ks. \tag{8}$$

To avoid pixel reordering, we should design the scaling function to ensure that no shrinkage exceeds 1. We have experimented with several ways of doing this, and have found that a simple and effective approach is to use:

$$S(k,s) = \min(k_0 s, 1) \tag{9}$$

where k_0 is found for a particular k by solving

$$\sum_{x=1}^{w} \min(k_0 s_1(x,y,t), 1) = k \tag{10}$$

by binary search.

This scaling function prevents the edge flipping problem, although at the expense of failing to fully preserve the vertical and time direction constraints. However, in practice, using this modified scaling function introduces negligible *additional* artifacts (most artifacts in our output come instead from insufficient continuity of the importance map).

It is also of interest that our concept of shrinkability can be used for *enlarging* or upscaling video too, if desired, simply by making the shrinkability negative. (This results in unimportant areas of the video being expanded to fill the extra pixels). Here the simple scaling function of

$$S(k,s) = -ks \tag{11}$$

can be used without restriction, as, while pixels cannot shrink in width by more than one unit, they can grow by an arbitrary amount.

3.6. Multi-sized video compression

Clearly, our approach of precomputing a multi-sized video is only useful if the multi-sized video is not too large—video files are already very large. The size of the multi-sized video clearly should be smaller than simply storing multiple (compressed) videos at all or many desired target resolutions.

The multi-sized video stores the x and y cumulative shrinkability maps with the video. Without compression, each has the same size as the original *uncompressed* video (i.e. much larger than the *compressed* video). However, each cumulative shrinkability map is a smooth function, because of its inbuilt continuity constraints for avoiding zig-zag artifacts and jitter. Because of this smoothness, they can be highly compressed. (Being based on differences, shrinkability maps might seem to require less storage than cumulative shrinkability maps. However, accumulated errors in using shrinkability values when constructing the output video negate any advantage due to ability to represent shrinkability maps in fewer bits).

An effective way to compress a single cumulative shrinkability map u^t is to use a conventional video encoder. If we use a uniform scaling function, 12-bit fixed point representation is sufficient to store $u(x,y,t)$ values for videos of input size up to 4096×4096. Since after use of our chosen lossy video compression algorithm, DivX 6, the lower bits of a YUV color value may change, we encode each 12-bit $u(x,y,t)$ value into the highest 12 bits of a 24-bit color in YUV space. We store the highest 6 bits $u_{11\ldots6}$ in Y, the next 3 bits $u_{5\ldots3}$ in U and the last 3 bits $u_{2\ldots0}$ in V, using the following equation:

$$Y = 32 + 3 \times u_{11\ldots6} \quad (12)$$

$$U = 112 + 4 \times u_{5\ldots3} \quad (13)$$

$$V = 112 + 4 \times u_{2\ldots0} \quad (14)$$

Converting u to YUV space in this way prevents quantization errors due to lossy YUV compression from affecting u significantly. After conversion, standard video compression technology (e.g. the DivX 6 codec) can take advantage of u^t's inter- and intra-frame smoothness to achieve high rates of compression. Using this method and the DivX 6 codec at high-quality settings to compress u^t, we achieve a somewhat smaller size than the compressed input video. Any artifacts specifically *arising from compression* have very little visual effect on the output resized video.

4. Comparison and Experimental Results

4.1. Time and storage comparison

We compare our methods to Wolf et al's non-homogeneous retargeting algorithm [WGCO07]. Although his paper did not address the creation of *multi-sized* videos, we note that his algorithm allows the possibility to precompute and store the sparse Cholesky decomposition for the linear system involved, to give a different representation for multi-sized videos, again allowing the least-squares problem for different target sizes to be efficiently computed.

Theoretically, both Wolf et al's algorithm and our approach need to solve a linear system $AX = b$ of the same size. As noted earlier, we reduce the problem to one of solving m linear systems of size N where m is the number of frames and N is the number of pixels in each frame. Each matrix A has only $O(N)$ non-zero values. The time complexity of solving such a linear system is $O(N)$ [BBK05]. However, compared to Cholesky decomposition which is used in Wolf et al's algorithm, our approach uses a multigrid algorithm, which is much faster than Cholesky decomposition if a high-precision solution is not required, the reasons for which were noted earlier. Furthermore, multigrid algorithms are amenable to parallelization for GPU implementation, although we have not yet done so. An important point is that, while Wolf et al's algorithm could also be accelerated by replacing the Cholesky decomposition by a multigrid algorithm, this could only by done for a *fixed* target size.

Turning now to storage requirements, for each frame, our approach needs to store N cumulative shrinkability values, while Wolf et al's method would need to store the sparse Cholesky decomposition of A. Even for sparse matrix input, this decomposition need no longer be sparse, although reordering can be done to reduce the number of non-zeros values after decomposition. We compared our method with a variant of Wolf et al's method adapted to give multi-sized images, using an approximate minimum degree ordering algorithm for this purpose [ADD04]. It resulted in about $34N$ non zeros; it was also necessary to store the position of each in the decomposed matrix. Thus, in practice, our approach needs much less storage than the sparse Cholesky decomposition approach used in Wolf et al's method. In addition, our cumulative shrinkability map is very smooth and can be highly compressed, as we note later.

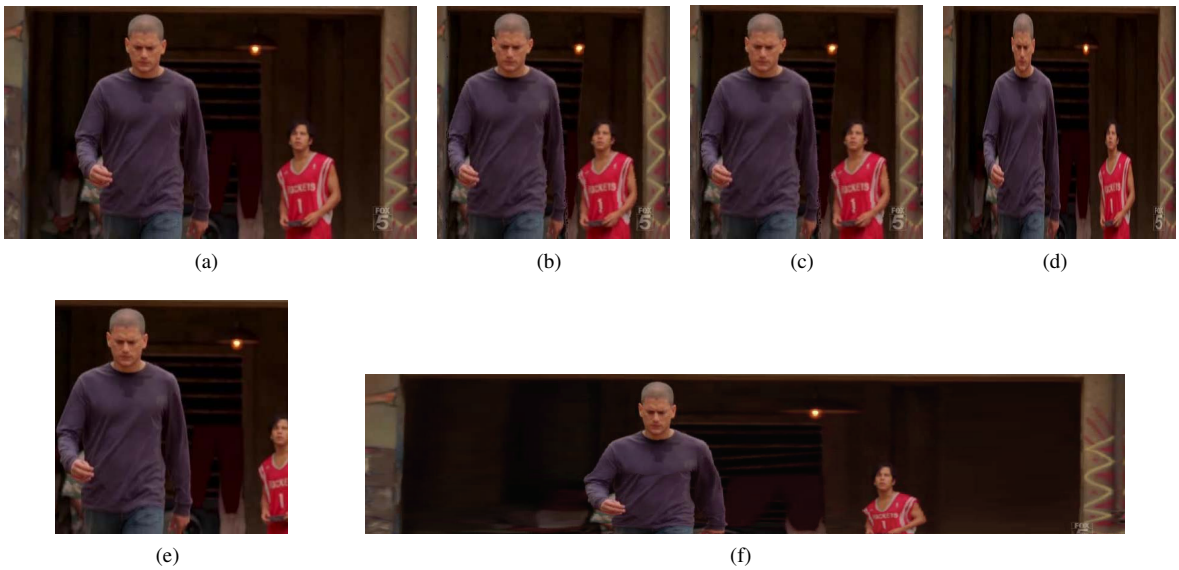
Thus, our shrinkability maps are useful because they support the creation of multisized videos which are highly compressible and can be computed efficiently using the GPU. The method in [WGCO07] also permits use of a multigrid method but it does not support multi-sized video. Storing the Cholesky decomposition would permit multi-sized video, but it would also require a huge storage space and would not permit multigrid acceleration.

4.2. Experimental results

Most multi-sized videos considered in this section only include a *single* cumulative shrinkability map for resizing the video's width, for simplicity, although we also show one example where both width and height are changed. Original videos and our cumulative shrinkability maps were compressed using the lossy DivX 6 encoder with 'Home Theater Profile'. All times were measured using an Intel 2.4GHz Dual Core Desktop with 2GB memory.

Table 1 gives the times needed by our algorithm and Wolf et al's algorithm to produce multi-sized video, for two input video clips taken from 'Prison Break' (see Fig. 2, $624 \times 352 \times 24$fps, PB for short) and 'Harry Potter and the Order of the Phoenix' (see Fig. 3, $720 \times 352 \times 24$ fps, HP for short). Table 2 gives the storage required by the corresponding multi-sized video. Clearly, our multigrid solver is much faster than Cholesky decomposition, and our cumulative shrinkability map needs much less storage than the corresponding Cholesky decomposition information. Our cumulative shrinkability maps, after lossy compression, are even smaller than the compressed input video, and the error caused by lossy compression has negligible affect on the output resized result, compared to not using compression. (See Fig. 3 for a comparison).

After creating the multi-sized video, we can decompress it and produce the resized result at over 100 frames per seconds with an unoptimised single threaded program, showing the potential to transmit simultaneous multiple video streams from a single server.



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

(e) (f)

Figure 2: *Resizing a frame from 'Prison Break, Season 3, Episode 10': (a) input frame, (b) our method, (c) Wolf et al's method, (d) uniform scaling, (e) cropping, (f) our method, enlarging the width while reducing the height*

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

Figure 3: *Resizing a frame from 'Harry Potter and the Order of the Phoenix': (a) input frame, (b) our method, uncompressed shrinkability map, (c) our method, lossy compression of shrinkability map, (d) seam carving method.*

Film clip	PB	HP
Input video running time	4s	4s
Cholesky decomposition	257s	297s
Multigrid Solver	14s	16s
DivX Compression	2s	3s

Table 1: *Precomputation time for multi-sized video.*

Film clip	PB	HP
Input video storage requirement	952KB	3.1MB
Cholesky decomposition	4.5GB	5GB
Uncompressed shrinkability map	64MB	73MB
Compressed shrinkability map	632KB	2MB

Table 2: *Storage needed for multi-sized video.*

Fig. 2 compares the quality of resized images obtained by our method and other resizing algorithms. In each case the same importance map was used. Uniform scaling makes people too narrow, while cropping loses context. Our approach produces results of a similar quality to Wolf et al's algorithm. Both approaches need to solve a linear equation; in fact, it can be shown that is possible to modify the weights in our random walk model to exactly reproduce Wolf et al's results. However, our approach is faster, and is amenable to lossy compression of the cumulative shrinkage map.

In Fig. 3, our algorithm preserves the structure of the red pole on the right hand side much better than seam-carving. This is because the seam-carving algorithm is a discrete method while our method is continuous method. The reason for this improvement may be explained as follows. Con-

sider an image containing a single, important, sloping, thick line. The desirability of removing any vertical seam from top to bottom is identical (each seam enters and leaves the sloping line exactly once). In such cases, the seam-carving algorithm will choose some seam at random to remove (in practice, one determined by local noise), which will destroy the structure of the sloping line. In the continuous methods, different pixels will shrink by different amount depending on their importance, and it is not too difficult to see that in this particular case, this reduces to uniform scaling, which is a much better choice than deleting arbitrary seams.

For more complex images, it may be best to completely remove unimportant pixels, whereas in other cases, uniform or non-uniform scaling may be the best choice. Discrete methods always remove pixels, while continuous methods can remove pixels in some areas (i.e. by scaling them to zero size), and scale them elsewhere, automatically. (See for example Fig. 3, where pixels around Harry's head have been removed, while pixels belonging to the red pole have been uniformly scaled.)

Clearly, in order to fit all the objects of interest into the new size, content-aware resizing technologies must change distances between objects. While this flexibility is often an advantage, the consequence is that it is much harder to avoid jitter than when using cropping or scaling based approaches, such as pan-and-scan. For certain applications (e.g. high-quality movie production), these disadvantages may be considered to outweigh the advantages of content-aware resizing, although careful hand-tuning of the importance maps may alleviate these problems to some extent. Carrying out experiments of determine to what extent content-aware resizing is really useful would require a serious effort. Despite this issue, we believe that there are plenty of cases and potential applications for which our algorithm works well enough, and in such cases, our approach can give a solution for video resizing which is efficient in both time and storage.

5. Conclusion

We have given a novel method for performing content-aware dynamic video resizing. We define the shrinkability of each pixel to give a cumulative shrinkability map, which can reduce the width or height of an image by one pixel. We precompute cumulative shrinkability maps for x and y directions, and store them as a multi-sized video, after lossy compression, with the original video. At run-time, we then use a scaling function to generate a solution for reduction (or enlargement) by an arbitrary number of pixels, to give a high-quality content-aware resized video. Our approach is efficient both in time, and the size of the multi-sized video.

A *random walk* model is used to find the shrinkability of each pixel, leading to a linear system which can be solved by a multigrid solver, allowing for efficient implementation, which in principle could readily be adapted for use on a GPU. While being much faster, and more economical of storage, our approach produces results of a comparable quality to state-of-the-art methods for image and video resizing.

Acknowledgements

This work was supported by the National Basic Research Project of China (Project Number 2006CB303106), the Natural Science Foundation of China (Project Number U0735001), the Specialized Research Fund for the Doctoral Program of Higher Education (Project Number 20060003057), and an EPSRC Travel Grant. Illustrations in this paper and accompanying video were taken from the following films: Harry Potter and the Order of the Phoenix, and Prison Break Season 3 Episode 10.

References

[ADD04] AMESTOY P., DAVIS T., DUFF I.: Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software 30* (2004), 381–388.

[AS07] AVIDAN S., SHAMIR A.: Seam carving for content-aware image resizing. *ACM Trans. Graph 26* (2007).

[BBK05] BOTSCH M., BOMMES D., KOBBELT L.: Efficient linear system solvers for mesh processing. In *IMA Conference on the Mathematics of Surfaces* (2005), vol. 3604, Springer, pp. 62–83.

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph 22* (2003), 917–924.

[Gra06] GRADY L.: Random walks for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell 28* (2006), 1768–1783.

[LG06] LIU F., GLEICHER M.: Video retargeting: automating pan and scan. In *ACM Multimedia* (2006), pp. 241–250.

[RSA08] RUBINSTEIN M., SHAMIR A., AVIDAN S.: Improved seam carving for video retargeting. *ACM Trans. Graph 27* (2008).

[TJS07] TAO C., JIA J., SUN H.: Active window oriented dynamic video retargeting. In *Workshop, On Dymanical Vision, ICCV* (2007).

[WAC07] WANG J., AGRAWALA M., COHEN M. F.: Soft scissors: an interactive tool for realtime high quality matting. *ACM Trans. Graph 26* (2007).

[WGCO07] WOLF L., GUTTMANN M., COHEN-OR D.: Non-homogeneous content-driven video-retargeting. In *Proc.ICCV'07* (2007), p. 4409010.

[Wol90] WOLBERG G. (Ed.): *Digital Image Warping*. IEEE Computer Society Press, 1990.