# Automatic Reconstruction of Screen Capture Videos by Removing Unwanted Objects

Hrishikesh Pardeshi
Adobe Systems Pvt Ltd
No. 5, Salarpuria Infinity
Bangalore -29 India

## ABSTRACT
Screen capture videos (recording computer screen) can contain unwanted objects hereafter referred to as "popups". Removing such unwanted objects becomes a challenge with existing techniques available for video inpainting. Screen capture videos are characterized by sharp edges and huge amount of text in the background. Hence, techniques involving intra-frame interpolation/extrapolation of pixels fail to produce good results.

In this paper, a multi-stage algorithm is proposed to detect and remove popups from screen capture videos. The steps involved are determining the duration of popup, exact shape of popup and source region for replacement. The algorithm does not perform any kind of interpolation/ extrapolation throughout. Instead, it finds the region/patch of pixels in the previous or future frames to replace the popup region (inter-frame replacement).

## General Terms
Image and Video Processing, Computer Vision

## Keywords
Popup removal, Reconstructing screen capture videos, Iterative edge detection

## 1. INTRODUCTION
Removing unwanted objects from videos and reconstructing specific areas (video inpainting) has been a major research area for long. However, inpainting frames that contain sharp edges and text is a difficult problem to solve. Intra-frame interpolation/extrapolation of pixels fails to produce good results.

Screen capture videos contain unwanted objects in the form of popups (notifications, alerts, dialogs etc.) which are captured unintentionally during screen recording. Removing such popups becomes difficult with existing techniques due to sharp edges and text in the background.

This paper intends to solve this problem by inter-frame replacement of pixels. The algorithm first determines the total duration of the popup. It then determines the exact shape of popup. It searches for a region/patch of pixels in the previous and future frames to act as replacement for the popup region.

## 2. PROBLEM STATEMENT
The algorithm in this paper intends to solve the problem of removing popups from screen capture videos. A rectangular region surrounding the exact shape of the popup is provided as input. The algorithm then determines the following (not in the same order):

1. Determine exact shape of popup.
2. Determine the duration for which the popup appears in the video.
3. Determine a source region of pixels to replace the popup region for every frame. This source may/may not be different for every frame.

## 3. RELATED WORK
Video inpainting has been a major research area. However, the specific problem of removing popups from screen capture videos is tackled by Camtasia freeze[4]. It simply freezes the first frame for the duration of the popup, set by the user.

Content aware fill in Adobe photoshop[3] was tried on various inputs. However, significant artefacts were observed indicating that directly image inpainting techniques proves unsuccessful.

## 4. ALGORITHM

### 4.1 Input
1. Screen recording.
2. Rectangular region containing the popup (selected by user).

### 4.2 Broad stages in popup removal
1. Determine start and end
   Output – start and end frame timestamp
2. Determine source region/frame for every frame
   Output – Vector indicating the choice of source region for every frame
3. Determine if popup has moved
   Output – Boolean value
4. Determine the popup mask
5. Perform actual replacement

### 4.3 Helper module
Finding the shape of the popup

An iterative algorithm is used combining edge detection and contour detection to determine the shape of the popup. The exact details are as follows:

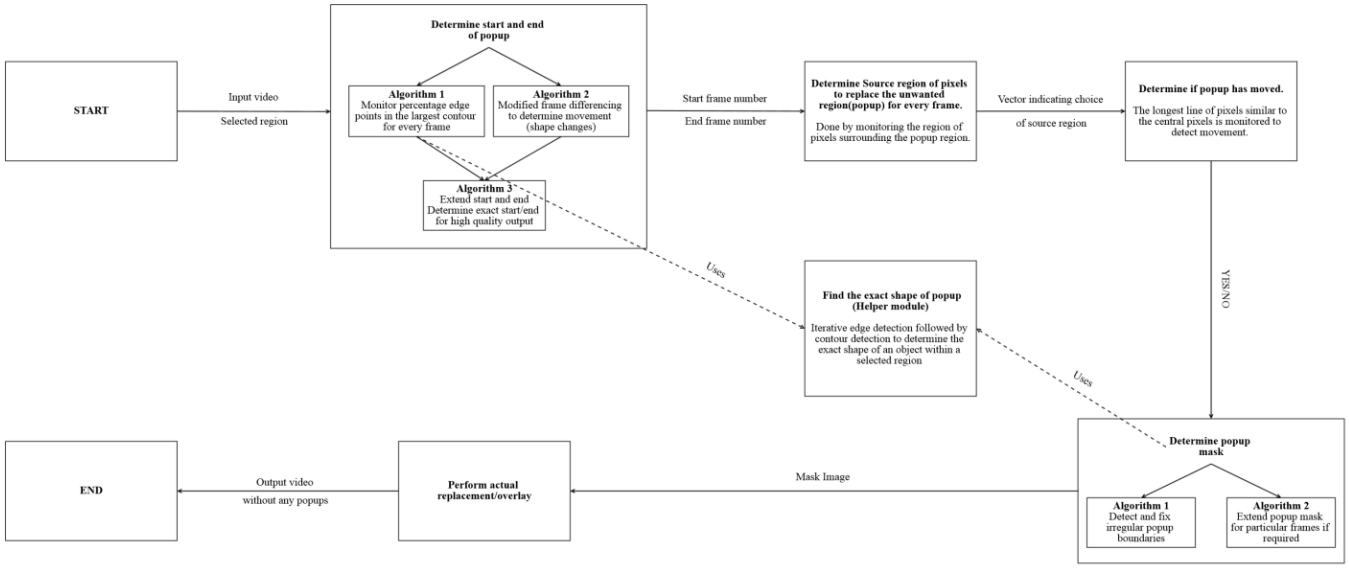



**Figure 1: Flowchart explaining various staged in the algorithm**

Find popup contour

1. Use canny edge detection to find the edges in the image.
2. Dilate the edge image mask.
3. Find the largest external contour in the edge image mask.
4. If the area of contour found in step (3) is less than 50% of the area of the image, relax canny threshold and go to step 1.
5. If area of contour > 50% of area of image, go to step 7.
6. If area of contour < 50% and the canny threshold falls to zero, popup contour could not be determined, return false.
7. If number of points in the contour is greater than 75, it indicates the contour shape is malformed, return false. Else if number of points is less than 75, return true.

OpenCV is used to perform canny edge detection and finding contours. The openCV method takes in parameters low_threshold and high_threshold for canny edge detection. the aperture size used is 3. The algorithm starts with a canny_threshold =32 and keeps decrementing the threshold. The low thresholds and high thresholds are computed as follows:

low_threshold = canny_threshold * aperture_size * aperture_size;

high_threshold = 3 * canny_threshold * aperture_size * aperture_size;

Thereby, a high:low hysteresis ratio of 3:1 is used. In the following discussion, threshold refers to canny_threshold.

In the following examples (and the rest of the examples), red colored region indicates the region of interest.

In examples 1 and 2, algorithm starts with a threshold of 32 and keeps moving down. In example 1, it terminates when threshold = 8 and in example 2, it terminates when threshold =24.

Example 1:


**Figure 2a: Source Image**


**Figure 2b: Threshold =13**


**Figure 2c: Threshold =12**


**Figure 2d: Threshold =11**


**Figure 2e: Threshold =10, 9**

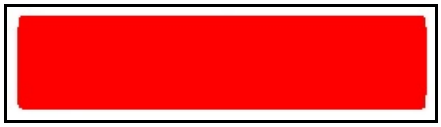


**Figure 2f: Threshold =8**

Example 2:

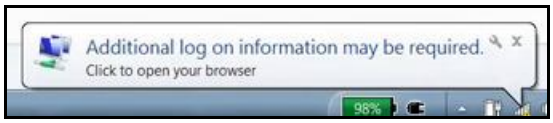



**Figure 3a: Source Image**

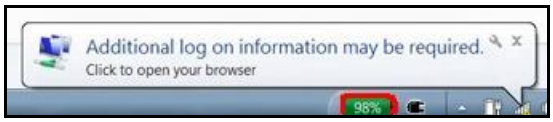



**Figure 3b: Threshold =27, 26, 25**

**Figure 3c: Threshold =24**

Example 3:

The following image (figure 4) shows an example of a malformed contour. It contains 190 boundary points which invalidates our condition. Hence, algorithm returns false in this case.

**Figure 4: Malformed contour**

## 4.4 Determine start and end

Determining the start and end of the popup is not trivial. Comparing frame differences doesn't help because each popup can appear/disappear in different ways. Example, Windows notification fades in/out, Skype alert moves from bottom to top, a tool tip/dialog box just appears. And even the effects don't have a specific pattern/uniformity. Example, Windows notifications does not fade in over every frame uniformly. It fades in, stays the same over 2-3 frames (depending on the frame rate of the video), fades in more and so on.

Three algorithms running independently are used to determine the start and end. The complete process is as follows:

Algorithm 1 builds up on the helper module algorithm. It is the primary algorithm to determine the duration (start and end) of the popup.

Selection_Rect – The rectangular region bounding the popup selected by the user

Pivot_Frame – frame in which the user has selected the popup region

Algorithm 1

1. Find popup contour (helper module), in Pivot_Frame.
2. If step 1 returns false, Algorithm 1 failed, return false.
3. Store all the edge points of the contour determined in step 1.
4. Start_Frame =Pivot_Frame, End_Frame =Pivot_Frame;
5. Start_Frame =Start_Frame-1
6. Find popup contour (helper module), in Start_Frame
7. Determine the percentage of edge points stored in step 3 which are also edge points in step 6.
8. If the percentage > 1%, go to step 5.
9. Start_Frame determined.
10. End_Frame =End_Frame +1
11. Find popup contour (helper module), in End _Frame
12. Determine the percentage of edge points stored in step 3 which are also edge points in step 11.
13. If the percentage > 1%, go to step 10.
14. End_Frame determined.

Output – Start_Frame, End_Frame

Algorithm 1 fails in case of moving popups like the Skype popup. Here the shape of the popup region itself changes while it appears on screen. A modified frame differencing algorithm is applied to account for this case. Note that here the algorithm first moves to a frame where the appearance/disappearance movement is about to start and then apply the frame differencing logic.

Algorithm 2

1. Start_Frame =Pivot_Frame, End_Frame =Pivot_Frame.
2. Start_Frame =Start_Frame -1.
3. If Start_Frame and Pivot_Frame are similar (90% of the pixels match with an absolute threshold of 8 pixels), go to step 2.
4. End_Frame =End_Frame +1.
5. If End_Frame and Pivot_Frame are similar (90% of the pixels match with an absolute threshold of 8 pixels), go to step 4.
6. Decrement Start_Frame until you get a continuous patch of 4 similar frames.
7. Increment End_Frame until you get a continuous patch of 4 similar frames.

Output – Start_Frame, End_Frame

Start_Frame =minimum( Start_Frame from Algorithm 1, Start_Frame from Algorithm 2 )

End_Frame =maximum( End_Frame from Algorithm 1, End_Frame from Algorithm 2 )

Algorithm 3 uses the inputs from Algorithm 1 and 2 and tries to extend the start and end.

The algorithm needs to check if it can extend the start and end because of the way semi-transparent popups like the mail notification appear as in the following example. The fade in

effect is extremely light in the initial frames, which goes undetected in algorithm 1 and 2.

Algorithm 3 – Extend Start And End

1. Start vertical rays of pixels from x=width/3 to 2*width/3, y =0 to height in Start_Frame.
2. If the pixels have same values, the ray continues. It stops on finding a dissimilar pixel.
3. If 75% of such vertical rays do not pass through the upper boundary of the reference contour of Pivot_Frame determined in Algorithm 1, decrement Start_Frame and go step 1.
4. Start_Frame determined.
5. Start vertical rays of pixels from x=width/3 to 2*width/3, y =0 to height in End_Frame.
6. If the pixels have same values, the ray continues. It stops on finding a dissimilar pixel.
7. If 75% of such vertical rays do not pass through the upper boundary of the reference contour of Pivot_Frame determined in Algorithm 1, Increment End_Frame and go step 1.
8. End_Frame determined.

Output – Revised Start_Frame and End_Frame

If Start_Frame from Algorithm 3 and Start_Frame from Algorithm 1 & 2 differ by 10 frames or more, discard the Start_Frame from Algorithm 3.

If End_Frame from Algorithm 3 and End_Frame from Algorithm 1 & 2 differ by 10 frames or more, discard the End_Frame from Algorithm 3.

Example:

The green line in the following images (figure 5) corresponds to the upper boundary in the above algorithm. The red lines correspond to the vertical ray of pixels.

**Figure 5: Set of images in which vertical ray of pixels is bombarded from top to bottom**

## 4.5 Determine source region/frame for every frame

There are two choices for the source frame – Start_Frame and End_Frame. For every frame in between the Start_Frame and End_Frame, the algorithm has to determine if the source will be Start_Frame or End_Frame. For this, it checks the number of pixels surrounding the selected region (by the user) in the current frame that match the corresponding pixels in the source frame. Whichever source frame gives a higher match is selected as the source for the current frame.

## 4.6 Determine if popup has moved

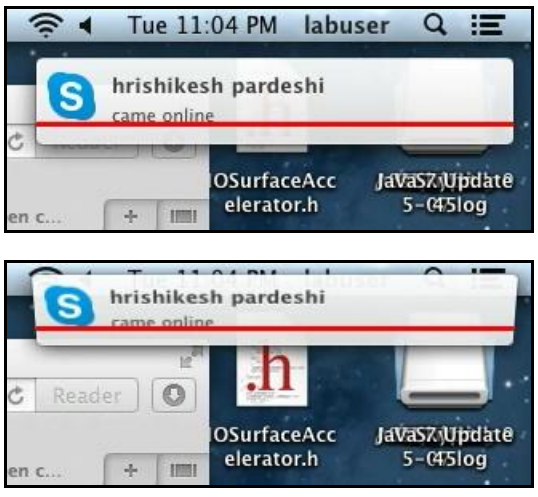
Mid_Frame =(Start_Frame + End_Frame)/2.
Centre_Pixel_Value =Value of Centre pixel of Selection_Rect in Mid_Frame.

1. Threshold the Mid_Frame based on Centre_Pixel_Value with an absolute threshold of 15 pixels. Create a mask.
2. Find the largest continuous line of white pixels (favorable pixels) in the mask. Store the height of this line (Y co-ordinate).
3. Decrement Mid_Frame till you reach Start_Frame. For every frame in between, perform steps 1 and 2.
4. Now we have a series of numbers representing the height of the favorable pixel line.
5. If the popup has moved, this series of numbers should form a continuously increasing or decreasing subsequence.
   Longest increasing/decreasing subsequence in the entire series was used in a specific implementation of the algoithm.

If the popup has moved, the algorithm sets the popup mask to be the Selection_Rect and go for the replacement of the entire selected region. However, an alternate approach to explore is to do a floodfill with the center of the favorable pixel line as the seed point. Floodfill itself was tested and it works well, however, the adjustment for edges and failure indications are yet to be explored.

Example of a moving popup:

The red line in the following images (figure 6) corresponds to the continuous line of pixels referred to in the above algorithm. It follows the movement of the popup as seen below.



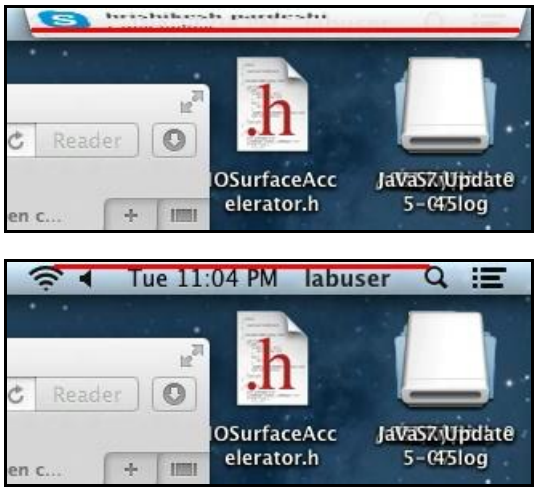



**Figure 6: Set of images demonstrating the detection of a moving popup**

## 4.7 Determine the popup mask

Mid_Frame = (Start_Frame + End_Frame)/2.

1. Find popup contour (helper module), in Pivot_Frame.
2. If step 1 returns false, the mask could not be determined, the entire Selection_Rect will be replaced.
3. If step 1 returns true, create the relevant popup mask.

Note here that the procedure above is similar to the one in Algorithm 1 to determine start and end. The popup mask could have been stored at that step itself and extra computation avoided. However, that popup mask would refer to Pivot_Frame while here it refers to Mid_Frame. Pivot_Frame could be very different from Mid_Frame and may not be the ideal choice to find the popup mask. Hence, this extra processing step.

The following additional algorithms are applied to tackle two specific cases:

1. Algorithm 1 ( for detecting irregular popup boundaries )

   There is a restriction in the "Find popup contour" helper module to have a maximum of 75 points in the contour. Otherwise, the contour is malformed. An additional algorithm is used to detect defects in a boundary of the contour.

   The algorithm is explained taking the example of the upper boundary. It can as well be used for any of the other boundaries.

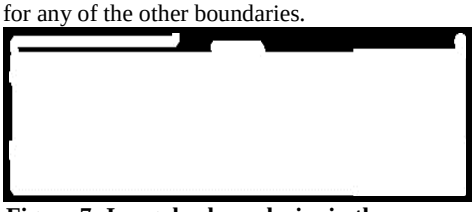


**Figure 7: Irregular boundaries in the popup mask**

In figure 7, the upper boundary of the popup mask is malformed. The following algorithm is used to detect this malformation:

   1. Start vertical ray of pixels from x= width/15 to 14*width/15, y =0 to height
   2. As soon a vertical ray hits a non-zero pixel (since this is a binary image there are only 2 values any pixel can have), stop the propagation of that ray and store the value of y.
   3. Plot a histogram of all the stored y values and ignore significantly small groups (ones that contain less than 6 pixels).
   4. In case of a malformed boundary like the one in figure 7, the number of groups in the histogram is greater than 4. In case of a well formed boundary (even if it has a beak), the histogram contains no more than 3 groups. Threshold based on the number of groups.
   5. Once we detect a malformed boundary, we can opt to straighten that particular boundary or even ignore the mask completely.

2. Algorithm 2 (Skype popup)


**Figure 8: Skype popup**

A closer look at the two images in figure 8 indicates that the left image has a slightly greater height compared to the right image. The image on the left is seen while the popup is appearing in the initial stages. The popup mask is calculated based on the Mid_Frame which is the image on the right. This implies that while replacing the pixels in the frame in which the left image occurs, we won't be replacing the extra height pixels (the difference in height between the 2 images). This prompts us to store separate masks for each frame. However, this would be expensive in terms of memory.

The following method is applied to handle this case:

1. Calculate the popup mask as per the original algorithm.
2. For every frame, Check if the pixels on the upper boundary (this can be applied to other boundaries as well) can be extended.
3. If step 2 is true, store the new extended mask. Else, we make use of the original mask.

## 4.8 Perform actual replacement

Inputs: Recording video, Selection_Rect, original popup mask, alternate popup mask vector, reference frame vector, Start_Frame, End_Frame.

For every frame between Start_Frame and End_Frame, do the following:

1. Copy the appropriate mask from original popup mask or alternate popup mask vector into Popup_Mask.

2. Find the Reference_Frame for the current frame from the reference frame vector.
3. Replace Selection_Rect in current frame with the corresponding region in Reference_Frame using the Popup_Mask.

## 5. RESULTS

Popups can be of varying shapes and sizes. Moreover, the background plays a pivotal role since it directly affects edge detection. Transparency, fade in/out and other effects also play a major role.

The algorithm has been tested on various inputs. The following class of videos with stated popups were successfully cleaned up:

**Table 1: Class of popups successfully cleaned up from screen capture videos**

| | Class of popups | Details |
|---|---|---|
| 1 | Windows notification popup | Classic windows popup with a beak |
| 2 | Windows notification popup in varying orientations | Position of the beak varies with respect to the rest of the shape |
| 3 | Skype popup | Moves from botton to top. Has sections of colours |
| 4 | Microsoft outlook mail notification | Semi-transparent popup |
| 5 | Windows dialogs | Classic windows dialogs e.g. download dialog |

## 6. FUTURE WORK

The algorithm stated in the paper is not restricted to only popups. It can be modified, reused and extended for any other sort of unwanted objects.

The algorithm works best when the selected rectangular region is as close to the exact popup boundaries as possible. However, there can be popups that move throughout the screen. Shadows and other artistic effects surrounding the popup also pose a risk to accurately finding the mask.

All of these areas can be explored by further research.

## 7. REFERENCES

[1] Learning OpenCV

Gary Bradski, Adrian Kaehler

[2] Canny John, "A computational approach to edge detection", IEEE Transactions on Pattern analysis and Machine intelligence, Volume PAMI-8, Issue: 6

[3] Content aware fill, Adobe Photoshop

http://www.photoshop.com/

[4] Freeze tool by Camtasia

http://www.techsmith.com/camtasia.html

[5] OpenCV, an open-source library for computer vision

http://opencv.org/