A Better Method of Telecine Removal

(Draft)

Kevin Atkinson
kevin at atkinson dhs org

29th July 2003

In this paper I will present a better method to remove telecine. My method will combine telecine fields and decimate the frame rate of the same time. Other methods generally remove telecine in a two step process. First telecine fields are matched up but the frame rate is not changed. Then duplicate frames are removed. This method however tends to lead to imperfect results with some frames being skipped and others being duplicated. This method will also not work well on Hybrid clips, that is clips that contain some telecine and some truly interlaced material.

My method, on the other hand, will generally lead to near perfect results if the material is telecine and as a bonus will reduce the frame rate of truly interlaced material with out excessive jerkiness or blurring of motion there for making it perfect for hybrid clips.

The latest version of this document and some sample code can be found at http://kevin.atkinson.dhs.org/tel/

1 Method

The basic idea is to double the frame rate of the interlaced source than intelligently select to best match the target frame rate. That is to go from 3:2 pulldown to 24 progressive the video will first be bob deinterlace which will give a frame rate of 60 fps, than frames are chosen in a 3:2 pattern to get the target frame rate of 24 fps. The frames are carefully chosen so that there would not be any duplicate frames in the final 24 fps video.

More specifically my filter goes through the following steps:

1. Bob deinterlace the video and compare each frame of this new video with the next frame. A dumb bob method is used here which simply interpolates.
2. Classify each field difference as either identical, similar, or different
3. Based on this classification, intelligently select fields to match the target frame rate.
4. Determine which field should be combined with the selected field to get a full resolution frame and how it should be combined.
5. Render the final frame.

1.1 Comparing Fields

Logically fields are compared using the following method:

1. Bob deinterlace the video using simple interpolation.
2. Blur the new frames using an even 3x3 kernel.
3. Compare every other pixel horizontally and every third line by summing up the square difference of the selected pixels.
4. Normalize the result.

The blurring helps to eliminate the effect of noise and insures that every pixel is accounted for in step 3.

However, I don't actually perform each of those steps separately as that will be too slow. The actual method is:

```
void subsample_line(byte * x, short * o)
{
  byte * stop = x + width;
  *o = x[0] + x[1];
  x += 2;
  ++o;
  while (x < stop) {
    *o = x[-1] + x[0] + x[1];
    x += 2;
    ++o;
  }
}

float compare (byte * top, byte * bottom)
{
  int line = 0;
  int i;
  byte * s_l[2] = {top, bottom};
  int s_height = height/3;
  int s_width  = width/2;
  short ad[s_width],bd[s_width],cd[s_width],dd[s_width],ed[s_width];
  short * a = ad, * b = bd, * c = cd, * d = dd, * e = ed;
  subsample_line(top, d);
  subsample_line(bottom, e);
  float accum = 0;
  while (line < s_height)
  {
    swap(a,d);
    swap(b,e);
    i = line & 1; // 0 if even, 1 if odd
    s_l[i] += width; subsample_line(s_l[i],c);
    s_l[i] += width; subsample_line(s_l[i],e);
    i = i ^ 1; // 1 if even, 0 if odd
    s_l[i] += width; subsample_line(s_l[i],d);

    // now compare lines
    for (int x = 0; x < s_width; ++x)
```

```
    {
      int v = a[x] + 4*c[x] + e[x] - 3*(b[x] + d[x]);
      v *= v;
      accum += v;
    }
    ++line;
  }
  static const float scale = 255* 6 * 3
  accum /= s_width*s_height*scale*scale;
  return accum;
}
```

1.2 Classifying Field Differences

After the fields are compared they need to be classified as either identical, similar, or different. The easy way to classify field differences is to have two thresholds. If the normalized value is less than the first threshold the two fields are considered to be identical, if the value is between the first and second threshold than the two fields are considered to be similar, otherwise the two fields are different. However this method has several problems, the first problem is obviously determining the value of these thresholds, the second is that, due to noise, the optimum value of these thresholds will be different for different videos.

So, instead I chose to automatically set these values by looking for peaks in the difference string. That is given three values representing the difference between four consecutive fields A B C D, if BC (ie the difference between B and C) > AB and BC > CD and AB and CD are close in value than the fields A and B are likely to be the same, fields B and C are likely to be different, and fields C and D are likely to be the same. If the peak is large than the first threshold is set to just above the difference of AB and CD thereby classifying AB and CD as the same and BC as different. If the peak is small the second threshold is set instead of the first thereby classing AB and CD as similar.

The relative code is as follows (The “vals” array contains the values AB BC CD):

```
if (vals[0] < vals[1] && vals[2] < vals[1]) {
  float p = vals[0] / vals[1];
  float n = vals[2] / vals[1];
  if (vals[0] < T1_ABS && vals[2] < T1_ABS &&
      p < T1_REL && n < T1_REL && abs(p - n) < T1_DIFF)
  {
    last_set = 0;
    thres1 = (max(p,n) + T1_T1)*vals[1];
    thres2 = (max(p,n) + T1_T2)*vals[1];
  }
  else if (vals[0] < T2_ABS && vals[2] < T2_ABS &&
           p < T2_REL && n < T2_REL && abs(p - n) < T2_DIFF)
  {
    last_set = 0;
    thres2 = (max(p,n) + T2_T2)*vals[1];
    if (thres1 > thres2) thres1 = (min(p,n) + T2_T1)*vals[1];
  }
}
if (last_set > MAX_LAST_SET)
  thres1 = THRES1_DEF;
  thres2 = THRES2_DEF;
  last_set = -1;
}
```

```
if (last_set >= 0)
  last_set ++;
```

The constants are currently set as follows:

```
static const float THRES1_DEF = 0.00010;
static const float THRES2_DEF = 0.00025;
static const int MAX_LAST_SET = 20;
static const float T1_ABS = 0.0025, T1_REL = 0.65, T1_DIFF = 0.07;
static const float T1_T1 = 0.05, T1_T2 = 0.15;
static const float T2_ABS = 0.0040, T2_REL = 0.92, T2_DIFF = 0.10;
static const float T2_T2 = 0.03, T2_T1 = -0.10;
```

However these are far from optimal and can probably use a good deal of tunning.

*_ABS are there to help avoid incorrectly classifying truly interlaced material as telecine and MAX_LAST_SET is there to reduce the damage in case it does.

1.3 Selecting Fields

After the fields are selected the fields need to be selected in a way to match the target frame rate while keeping motion as smooth as possible. The easy way to chose what frames to selected is select frames in a regular pattern to match the output frame rate. For example to go from 60 to 24 select frames in a 2:3 pattern. However, this method may lead to duplicating frames if the material was telecided and the selection pattern is out of phase with the telecide pattern.

So, instead a more complicated method is used which takes into account how similar the frames are as given by step 2. The selection logic is rather complicated and difficult to explain in words so I will instead simply present you with the code (prev is the previous chosen field number):

```
float ideal = prev + accum + ratio;
int     cur = (int)ideal;
int p = prev - cur;  // note p does not nessary have to equal -1
float frac  = ideal - cur;
enum Chosen {Either, Cur, Next};
int chosen = Cur;

if (first || last)                                  chosen = Cur;
else if (same_group(0, 1))                          chosen = Either;
else if (same_group(p, 0))                          chosen = Next;
else if (similar_group(p,0) && similar_group(-1,0)) {
  if (similar_group(0, 1) && !same_group(1, 2))     chosen = Either;
  else if (diff_group(0, 1))                        chosen = Next;
} else if (diff_group(p, 0) && diff_group(-1, 0)
           && diff_group(1,2))                      chosen = Either;

if (chosen == Either) {
  field = frac < round_div ? cur : cur + 1;
} else if (chosen == Cur) {
  if (round_div < frac) round_div = frac + DELTA;
  field = cur;
} else if (chosen == Next) {
  if (round_div > frac) round_div = frac - DELTA;
```

```
  field = cur + 1;
}

accum = ideal - field;
```

All numbers are relative to “cur”.

See A for the logic used to determine the value of chosen.

The accum value is used to maintain the current selection pattern even when the material no longer appears to be telecine. For example:

```
a-a-a b-b c-c-c d-d e-e-e f-f g h i j k l m n ...
|   |     |   |     |   |     |   |     |     ...
^     ^     ^   ^     ^   ^     ^   ^     ^
```

The ’|’ is the rigid 3:2 pattern. The ’^’ is the frame chosen. Notice how the 3:2 pattern is maintained even when the material fails to be telecine. Without the use of accum it would be:

```
a-a-a b-b c-c-c d-d e-e-e f-f g h i j k l m n ...
|   |     |   |     |   |     |   |     |     ...
^     ^   ^     ^   ^     ^   ^   ^     ^
```

This can be important when the material is still 3:2 pulldown but it is failed to be detected due 60fps overlays or the like.

1.4 Choosing The Other Field to Combine With

After the best field is chosen it needs to be combined with another field to get a full resolution frame. Rather than attempt to describe the logic used in English I will, once again, simply present you with the code:

```
if      (same_group(0, matching))   use_f = Matching;
else if (same_group(0, other))      use_f = Other;
else if (aggressive) {
  weave = false;
  if (similar_group(0, matching))   use_f = Matching;
  else if (similar_group(0, other)) use_f = Other;
  else                              use_f = Matching;
} else {
  weave = false;
  int other_n = -(matching + 1);
  if (!same_group(other, other_n))  use_f = Matching;
  else                              use_f = Neither;
}
```

Zero here represents the current field. Given the fields P C N, where C is the chosen field, either field P or N can be chosen. The “matching” field is the field that was originally part of the same frame: if C is odd than that will be P, if C is even than that will be N. The “other” field is the field in the opposite direction of the matching field: if C is odd than that will be N, if C is even than that will be P. “Neither” means to only use the selected field and to interpolate to get the full frame size.

The aggressive flag is described as follows:

Always match a field up with a frame and also allow a field to be in a frame out of order when deinterlacing needs to be done. Will lead to better results when the material is truly 3:2 Pulldown (or the like). Might cause strange results when the material is truly interlaced and deinterlacing is done by blending fields together. However, if interpolation is used on the moving areas of an image then this option is always safe to use and should lead to better results.

1.5 Rendering The Final Frame

The final step is to render the frame.

For best results smart bob deinterlacing tecniues should be used for interlaced frames. Unfortually my filter does not do so, it simily deinterlaces by throwing out the other field in intoplating which is far from optimial.

2 Results

I have tried my filter on several clips, including some hybrid clips and the results are very good. I could not find any duplicate frames in the clip, even during scene changes and switching from animation to film and back.

A Logic Used When Choosing the Best Frame

```
a b | c d e
```

The two choices for the current frame are 'c' and 'd'.

'a' is the previously field chosen

'b' is c - 1 (which may be the same as a)

'c'

'd' is c + 1

'e' is c + 2

ac difference between fields 'a' and 'c'

0 means almost identical fields

1 means similar fields

2 means different fields

- means either field 'c' or 'd' can be chosen. It doesn't really matter.

The following combinations are illegal:

ac	bc
0	1
0	2
1	2

Thus there are 54 possibilities left:

ac	bc	cd	de	
0	0	0	0	-
0	0	0	1	-
0	0	0	2	-
0	0	1	0	d
0	0	1	1	d
0	0	1	2	d
0	0	2	0	d
0	0	2	1	d
0	0	2	2	d
1	0	0	0	-
1	0	0	1	-
1	0	0	2	-
1	0	1	0	c
1	0	1	1	c
1	0	1	2	c
1	0	2	0	c
1	0	2	1	c
1	0	2	2	c
1	1	0	0	-
1	1	0	1	-
1	1	0	2	-
1	1	1	0	c
1	1	1	1	-
1	1	1	2	-
1	1	2	0	d
1	1	2	1	d

1	1	2	2	d
2	0	0	0	-
2	0	0	1	-
2	0	0	2	-
2	0	1	0	c
2	0	1	1	c
2	0	1	2	c
2	0	2	0	c
2	0	2	1	c
2	0	2	2	c
2	1	0	0	-
2	1	0	1	-
2	1	0	2	-
2	1	1	0	c
2	1	1	1	c
2	1	1	2	c
2	1	2	0	c
2	1	2	1	c
2	1	2	2	c
2	2	0	0	-
2	2	0	1	-
2	2	0	2	-
2	2	1	0	c
2	2	1	1	c
2	2	1	2	-
2	2	2	0	c
2	2	2	1	c
2	2	2	2	-

Add don’t cares (’-’)

ac	**bc**	**cd**	**de**	
-	-	0	-	-
0	0	1	-	d
0	0	2	-	d
1	0	1	-	c
1	0	2	-	c
1	1	1	0	c
1	1	1	1	-
1	1	1	2	-
1	1	2	-	d
2	0	1	-	c
2	0	2	-	c
2	1	1	-	c
2	1	2	-	c
2	2	1	0	c
2	2	1	1	c
2	2	1	2	-
2	2	2	0	c
2	2	2	1	c
2	2	2	2	-

Combine

ac	**bc**	**cd**	**de**	
-	-	0	-	-
0	0	1/2	-	d
1	0	1/2	-	c
1	1	1	0	c
1	1	1	1/2	-
1	1	2	-	d
2	0	1/2	-	c
2	1	1/2	-	c
2	2	1	0/1	c
2	2	1	2	-
2	2	2	0/1	c
2	2	2	2	-

Combine more

ac	**bc**	**cd**	**de**	
-	-	0	-	-
0	0	1/2	-	d
1	0	1/2	-	c
1	1	1	0	c
1	1	1	1/2	-
1	1	2	-	d
2	0/1	1/2	-	c
2	2	1/2	0/1	c
2	2	1/2	2	-

Avoid unnecessary tests:

’_’ means it doesn’t matter

(The first match is chosen)

ac	**bc**	**cd**	**de**	
-	-	0	-	-
0	_	_	-	d
1	1	1	1/2	-
1	1	2	-	d
2	2	_	2	-
_	_	_	_	c

DONE

- convert to if statements
- deal with border cases

B Copyright