Speed up video with composite merging frames (averaging // "motion blur")?

[+3] [1] sdaau

[2017-04-17 07:14:11]

[linux video ffmpeg open-source]

[https://superuser.com/questions/1200383/speed-up-video-with-composite-merging-frames-averaging-motion-blur]

Basically, I want to speed up a video (typically some math based stuff like Bouncing balls example in processing with source code - YouTube [1]); however, I don't want to just drop frames, but instead I want to "merge" (for lack of better term) frames, like shown on the diagram:

```
+----------+----------+----------+----------+----------+----------+
| Frame 01 | Frame 02 | Frame 03 | Frame 04 | Frame 05 | Frame 06 |
+---+------+-----+----+----+-----+-----+----+-----+----+----------+
    |            |         |           |          |
    |    +---+   |         |           |          |
    |    |   +<--+         |           |          |
    +--->+ + +<------------+           |          |
         |   +<-----------------------+          |
         ++--+<-----------------------------------+
          |
    +-----v-----+
    | NFrame 01 |
    +-----------+
```

In other words: if I want to speed up video 5 times, instead of just "decimating" by taking every 5th frame (so original Frame 01 is then followed by Frame 06 in the new output stream), I would like the new frame (in the new output stream) to be a "sum" of the frames from 01 to 05:

```
NFrame01 = k*(Frame01 + Frame02 + Frame03 + Frame04 + Frame05)
```

Since the color range is limited, I would need a constant k to control color values: say, we work with RGBA pixels, with ranges from 0.0 to 1.0; then, if at position x,y of each original Frame01-Frame05 is full red (1,0,0,1), I would have to multiply the alpha of each input pixel by 1/5 = 0.2, to ensure the output pixel (the sum) is also full red (1,0,0,1) without going over the color range; roughly speaking:

```
NFrame01(x,y) = [1.0, 1.0, 1.0, 0.2]*(Frame01(x,y) + Frame02(x,y) + Frame03(x,y) + Frame04(x,y) + Frame05(x,y))
```

(Alternately, assuming RGB pixels with no alpha, we'd have to multiply each of the RGB channels by 0.2)

If we have a mathematical video like the bouncing ball example, where there is no natural motion blur, I guess this would result with a "motion blur" of sorts (i.e. instead of one ball per frame, I'd have five balls on a frame, tracing the motion).

I guess I could make this by extracting the frames as images, and have my own custom code generate new frames, and finally make a new output video out of the new frames - but since that may take me "forever", I was wondering: can maybe `ffmpeg` (or other open-source tools) do this in a "one-liner"?

[1] https://www.youtube.com/watch?v=eSvrwO3MFiw

(1) See video.stackexchange.com/q/16552/1871 - **Gyan**

---

[0] [2017-04-17 11:05:53] sdaau

While the link from @Mulvya https://video.stackexchange.com/q/16552/1871 does indeed answer the question with `ffmpeg`:

```
ffmpeg -i input \
-vf "tblend=average,framestep=2,tblend=average,framestep=2,setpts=0.25*PTS" \
-r srcfps -{encoding parameters} output
```

... note that (https://ffmpeg.org/ffmpeg-filters.html):

> The tblend (time blend) filter takes two consecutive frames from one single stream, and outputs the result obtained by blending the new frame on top of the old frame.

So, it *only* blends two frames together, which means that for blending four frames, you have to repeat `tblend=average,framestep=2` twice, as in the example above.

But, I want to blend 700 input frame images per output frame image (and I doubt `tblend=average,framestep=2` repeated 350+ times will be parsed correctly by `ffmpeg`). So I decided to first unpack the frames, then do my own processing using Python. To unpack:

```
mkdir ofrs # original frames
mkdir outfrs # out frames
ffmpeg -i myvideo.mp4 ofrs/img-%05d.png
```

... and then I use this python script with `python blendManyImages.py`; since having each image with equal weight in the blend didn't result with the image features I needed, this scripts uses a formula that gives bigger weight to images earlier in the stream:

`python blendManyImages.py`:

```
# http://stackoverflow.com/questions/25102461/python-rgb-matrix-of-an-image
# http://stackoverflow.com/questions/40810716/how-to-get-a-list-of-float-rgba-pixels-values-using-pillow


from PIL import Image
import numpy
import math

# open an image, to get the data size:
im = Image.open('ofrs/img-00001.png')
#~ data = numpy.asarray(im)
data = numpy.array(im) # same as .asarray
print("Array dimensions: %s"%(repr(data.shape)))
data = data.astype(float)
print("[20, 30]=%s"%(repr(data[20, 30])))
#~ print(data)
#[[[240. 240. 240.]
#  [240. 240. 240.] ...
#~ data = numpy.divide(data, 255.0)
#[[[ 0.94117647  0.94117647  0.94117647]
#  [ 0.94117647  0.94117647  0.94117647] ...
# erase data:
data.fill(0)
#~ print(data)

inputframes = 44100
outptframes = 60
decimate = inputframes/outptframes # 735
k = 1.0/decimate # 0.001360
print(decimate, k)
i = 1 # input frame counter
o = 1 # output frame counter
while i <= 44100:
  data.fill(0)
  for dcnt in xrange(0, decimate):
    ifname = "ofrs/img-%05d.png"%(i)
    #print(ifname)
    tdata = numpy.divide(numpy.array(Image.open(ifname)).astype(float), 255.0)
    # manually tuned formula: give more weight to earlier frames
    data += numpy.multiply(tdata, k*70*pow(math.e,-0.05*dcnt))
    i = i+1
  # data should be done here; save
  ofname = "outfrs/img-%02d.png"%(o)
  print(ofname)
  oim = Image.fromarray(numpy.multiply(data, 255).astype('uint8')).convert('RGB')
  oim.save(ofname)
  o = o+1
```

And once this output frame image sequence is calculated, one can make a video out of it, again using `ffmpeg`:

```
ffmpeg -framerate 60 -i outfrs/img-%02d.png output.mp4
```

1