Student: BURGELIN Clément (249954)
Section: IN Ba6
Supervisor: Zhou Ruofan
Professor: Süsstrunk Sabine
Lab: Image and Visual Representation Lab (IVRL)

7th June 2018

Motion-Compensated Frame Interpolation

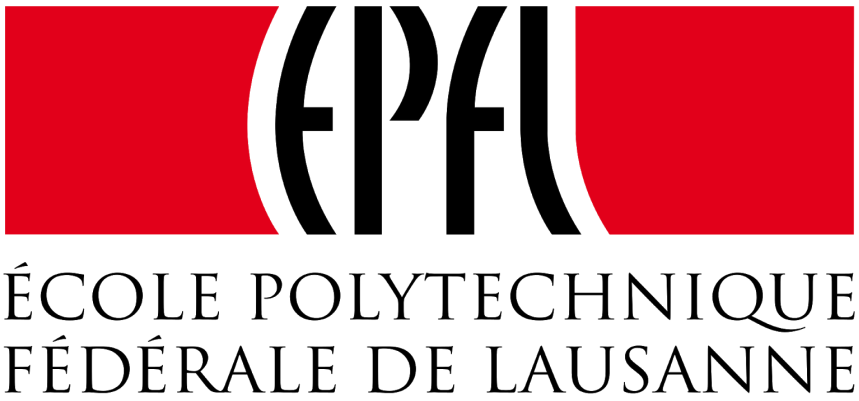


TABLE OF CONTENT

INTRODUCTION

First, let's try to explain the problem at hands. Video are composed of a succession of images that we call "frames" that are displayed one after the other at a defined speed called the "frame rate" (with unit the "fps" for frame per seconds). The motion in a video with an higher frame rate will look smoother and more real. If you slow down a video, the frame rate will also be lowered at the same time. With a low frame rate, the motion seems less smooth (and things looks less real) since people start to see the frames more individually. Additionally to the fact that a lower frame rate looks less smooth and real, there is another problem: Nowadays, most films and series that are displayed on TVs and/or in cinemas have a frame rate of 24 fps (24 * 1000/1001 to be exact). While 24 fps is the standard form films and series, the standard for computer monitors is 60 Hz (which is the frequency at which the monitor refreshes the image on the screen), which is perfect to display 60, 30, 20 and 15 fps content for example (any divider of 60), but since 24 is not a divider of 60 (24 = 60 * 2 / 5), it's not ideal and what happens in this case is that even frames will be displayed during 3 monitor refresh while odd frames will be displayed during 2 monitor refresh, which causes a slightly uneven motion.

Motion-compensated frame interpolation algorithm aims to generate intermediate frames from the original video frames, to increase the frame rate of a video and make the motion look smoother. There's already some algorithms on the market available that do that, SVP (SmoothVideo Project) being the most popular. Having an always perfect interpolation is definitely impossible, since there is definitely some information missing "between the frames", since otherwise it would mean that we would only need the first and last image of a video and some interpolation to get other frames, which is why there is always room for improvement in terms of quality. There's also the speed factor, since you might want to do it on real time, but SVP is already able to give real time results on machines which good enough specs.

Most of the current algorithms on the market are using a motion-flow algorithm to estimate the motion. The motion-flow method is based on two assumption about the video motion:

- The motion of an object between two frames is "small" compared to the size of the frame. (not more than 60 pixels for a 720p video for example)
- The motion of two pixel close to each other has to be very similar.

Those assumption make sense since two frames are supposed to be relatively close in time, so humanly visible motion should be pretty small and two pixel close to each other are probably from the same object but because of this second

assumption, most algorithm have trouble dealing with borders of moving objects, (since they don't know where the border are, and consider the whole as one big object) small particles, and all sort of thing and this causes unpleasant artifacts that they are trying to mask with different techniques. Since the algorithm is trying to find pixels with similar color, they also have trouble with object that are changing in color/light like a sunset sky.

Another problem comes with drawn content like anime (from the japanese abbreviation of "animation", which defines drawn content in the style of manga) where some of the animations are only at 12 or 8 fps, when the video is at 24 fps, which causes problems when you interpolate while only looking at the previous and next image. Having some optimisation to handle those cases (to either completely interpolate or completely not interpolate them) would be good.

Now comes the new method I try to implement in this project. Instead of estimating the motion using the motion-flow method, I use a feature matching algorithm combined with a motion coherence algorithm. Features in images is some part of the image that has special caracteriques, so that multiple images of the same object should detect the same feature on that object. The process of finding which feature in the image B corresponds to a certain feature of the image A is called feature matching, and involves a description of the features to compare them. There are several disadvantages of feature matching compared to motion flow:

- The motion you will obtain will be more sparse (since you only have information at the position of the feature, and it depends on how much features you have) but it's probably fine since you have more features in zones of greater contrasts, which is where a good interpolation matter the most.
- On most machines and with most feature detection and description algorithms, matching a lot's of features is slower, but we can probably reach enough speed and since motion-flow needs a lot of artifact masking, a good algorithm might even be able to go faster, even though speed is not the main objective of this project.
- Two similar objects have a great chance to be "wrong matched" (which can cause big errors) but this is were motion coherence can be used to make sure the feature you keep have a reasonably "logical" motion.

As you can see, I have counter-arguments for all of those, and the lower number of motion vectors to manage allows us to implement optimisations at a lower cost.

LITERATURE

1. Motion Flow

Motion flow is probably one of the simplest method you can think of to find motion. The technique consist in reducing the resolution a lot to first find some “sparse” and “coarse” motion by searching a close pixel with similar color in image B as the pixel in image A, and that for every pixels. Then you higher the resolution progressively while improving the motion of every higher resolution pixel until you reach the full resolution image. This technique can be pretty efficiently be done and avoid “big errors” but the motion obtain on border of objects is often wrong because of the assumption that close pixels probably have the same motion.

2. Higher frame rate and limitation

While it’s true that an higher frame rate gives a smoother motion, some people tend to dislike watching films at an higher frame rate because it looks “too real” and the “lower” frame rate of films can be considered as a part of the cinematographic effect, but it’s mostly a question of opinions and habits. At the opposite, from people I talked to and from my own experience, It seems that some people used to 60 fps or more feel pretty bad watching travelings at 24 fps for example, since they tend to see the frames a lot more individually. Now, while highering the frame rate will make it look smoother, there’s two limitation:

- First, having an higher frame rate than what your monitor can support is definitely useless for films and series.
- Secondly, our eyes have some kind of persistence which means that an image takes some time to “appear” and “disappear” to our eyes, like a capacitor takes some time to be charged to a certain level of energy. Due to this, higher and higher frame rate make less and less difference as you go higher.

3. Features, Detectors, Descriptors and matchers.

Feature matching is used to find a given object in different images and allows to define its location. To do feature matching, you have to use three parts:

- You first need a feature detector that will determine some “interest points” in an image, with the objective to find the same interest points in different pictures of the same object. Those points are often positioned in high contrast area, around borders and angles.
- You then need a feature descriptor that will describe those points (generate “descriptors”) to give you some data that you can use to compare points

between different images and find similar points. We could simply use the color of some pixels around the feature, but most descriptors offer more robustness to light exposition, angle, and scale changes.

- You then need a descriptor matcher that will compare the descriptors to find whether some interest point correspond to another interest point. This matcher will return matches will a distance associated to them. In this part, you can get a lot of wrong matches if you really want a lot of matches, and the higher the computed distance is, the higher the chances that the match is wrong are, but how much proportion of good feature you can have on a certain amount of features mostly depends on how good the detector and descriptor are.

4. Small side note on SSA subtitles

SSA stands for “Sub Station Alpha” and is a format of subtitles witch is part of the matroska (mkv) format. SSA is mostly used in “Fansubs”, which are subtitles made by independent group of fans of some kind of medias and are pretty common in anime. SSA subtitles give much more possibility than traditional subtitles and all sort of coloring, animation and transformations possibilities. In some video content, SSA transformations abilities are used to have subtitles that are placed on elements of the scene, over or besides the original text, in order for the viewer to be able to read the subtitle without even noticing that the text was added and not present in the original video. (It can be for a restaurant menu or a sign for example)

The problem when you use interpolation algorithm to interpolate the motion of the video is that the subtitles are not interpolated and subtitles that were placed on some elements of the scenes will definitely look bad since they move at a different frame rate than the scene elements. We will not cover this in this paper, but that’s definitely something that would improve the visual quality of the interpolation of those videos.

IMPLEMENTATION

Now let's talk about the implementation.

1. Tools used

For this project, I chose to use OpenCV for the feature matching part, and because of than, I chose to use python (python 3) since C++ is more complex to use and the Java wrapper for OpenCV is not as easy to get as the python wrapper, especially if you want the "contrib" part of OpenCV, which offers much more choice for feature detectors and matchers.

While python is a good language for image processing, it was a big challenge for me to use python since I was use to static and strong typing, and I'm not a big fan of python collection. I ended up discovering and using the python type hinting to help me in the process, and only started to use generators at the end.

I also used numpy for various operation and also because OpenCV uses numpy arrays to store images.

2. Motion detection and generation of MotionMatches

The first thing we have to do is to compute the motion. For that task, we first use a feature matching technique. From the feature matching, we want a motion field as dense and robust as possible to get a more precise interpolation. Because of that, I've tried a lot (almost all that are available in OpenCV) of different combinations of detectors and descriptors to see which one gives the best matching. You can see my testing algorithms in detector-test.py and matching-test.py and some of the results in the matching-tests folder. I looked into the configuration of all the detectors to increase the amount of detected features and compared the percentage of matches that looked right in function of the distance of the two points on the two consecutive video image to evaluate their quality. I ended up using AGAST for the detection and BRIEF for the description since they give very good matching results and generate a pretty dense motion field while being quite fast to compute. On one of my machines, I achieved a complete detection and description of approximately ~5000 features in about ~0.01 seconds.

For the matcher, I use a simple brute-force matcher since the FlannMatcher (which is using a clustering algorithm) in OpenCV didn't give better results. I could probably improve the matching speed by separating the features in different image areas in order to only try to match features that are close to one other in order to increase the matching speed, but I thought It would not be the priority.

Now comes the last part of the motion vector generation and also the most important part of the whole algorithm: The motion coherence filter.

Instead of just using 2 frames at a time to compute motion like most algorithm does, I chose to use more frames here, in order to follow the features over more frames and check if they “move correctly” to both be able to take have a denser and more robust motion field. That’s where my MotionMatch class and their ability to be “merged” comes into play.

Once I obtained the matches over all the possible pair of consecutive frames, I transform all the matches into my own made “MotionMatch” objects which allows me to both merge them over several frames and compute a better distance estimation of “how good” a feature is, considering both the original matching distances and considering the motion of the feature over several frames.

While having a “high” speed is possible (an object that travels one 6th of the frame width per frame for example), having a high acceleration is very unlikely (only a few pixels per frame), and for this reason, comparing the acceleration of the features seems to make much more sense that comparing the speed. Also, the derivative of acceleration (which is named “jerk”), which is how much does the acceleration change over the frames also shouldn’t be very high. We could probably, go higher in the derivatives, but I thought this much would be enough. A small error in the chain of features points matched can easily be seen in the acceleration or jerk component.

Using those values (estimated from subtraction) and the average match distance of the matches, I made a lot of testing and came up with a new formula for the final match distance that is defined in motion_match.py.

From this new match distance, I check for each final match obtain that it is better than any other matches going through the same points, and that it is good enough using a threshold.

For example, from 100 images, I have 99 lists of matches, and after 3 consecutive merging (when using degree=4), I end up with 96 lists of MotionMatches, each MotionMatch corresponding to a feature matched over 5 consecutive frames.

Due to the amount of features and the computation needed for the merging, some optimisation where needed to make sure the algorithm does not run too slow and does not eat to much memory.

In a previous version, I was mostly using python lists at every single step, which ended up taking much too much memory when it comes to process videos even with only a few hundred frames, so I tried to use more generators instead, which I was not very familiar with.

Even with generators, the merging process was generating too much possible merges and only filtering at the end, which still took too much time, so I ended-up pre-filtering at every merge step.

Now with all that, the performances of the merges are still pretty bad and I think python collections are partially in fault here. In the future, using Spark collections or something of this kind could probably increase the merging speed a lot, and I wish I had the had to do that before the project deadline.

In the testVideo.py, I use pickle to save the final collection of MotionMatches to a file in order to be able to retrieve it later to directly start the frame interpolation from it. It takes a lot of space and I which I could find a way to store the motion more efficiently in the future.

3. Frame interpolation from the MotionMatches

Now that you have the MotionMatches, all there is to do to get some motion vectors (that I called “MotionFeatures”) is to call the “genMotionFeature” of those MotionMatches with the new frame time as an argument. This function generate a new “MotionFeature” with its coordinate being the estimated position of the feature at that time, and the relevant information to find the corresponding pixels in the previous and next image.

Now, there is supposedly several methods to interpolate from those “sparse” vectors, and the method I implemented is pretty simple but also pretty slow. In order to generate the new image, I iterate over all the pixels of the new image, and check what feature is the closest to that pixel, and I consider that the pixel is moving the same way and at the same speed as that feature. From that motion, I get the corresponding pixel in the previous and next image and average them to get the new pixel value. This averaging allow less visible artifact in bad interpolated areas by generating some kind of motion blur while still providing very sharp images in other areas.

There is two major problems of this technique. The first one is about speed, and is that I need to do a pretty expensive task of finding the closest feature for every single pixel. To try to reduce that cost - since I process pixels line by line - I save the best feature I found for a pixel to be able to use it for the next pixel, and I group the features in columns of the image, which means I only have to search in close columns to try to find a feature closer than the one I previously found. Also, since the probability to find a closer feature on the left of the pixel is pretty low, I only search to the right. This whole thing considerably speeds-up the process, since you don’t have to compute the distance to every single feature for each pixel, but it’s still really slow.

The second major problem is about the segmentation of the final image, since there is some very definite areas where a motion is applied, and another completely different motion can be applied to the pixels next to it. Also, since the quality of some matches is better than other, it would also be good to take that into account.

4. Alternative interpolation methods (not implemented)

To resolve the problem of the segmentation and also probably improve speed, I thought about 2 solution that I did not have the time to implement. Since a great part of my project resides in this kind of reflections, I will explain those two solutions in detail here:

The first solution is to use a big height x width x 3 float array (let's call it M) to compute the motion of every pixels of a new image. The idea is, for each feature to multiply a vector (dx, dy, 1) (with dx and dy corresponding to the motion) to a weight w depending to the feature quality and then to a weight matrix (probably a N x N 2D gaussian matrix), to get a 3D N x N x 3 array (let's call it F) as a result with the motion on dx and dy weighted by the proximity to the feature location, and the corresponding weight on the 3rd layer. Each MotionFeature would then have it's own F array, and we would only need to add all those array to the M array (at the corresponding location).

In the end, we just have to divide the dx and dy layer of the matrix by the weight layer in order to get the motion average weighted at each pixel by the feature distances, and a very fast iteration over all pixel would allow us to compute every pixel's value from the previous and next image pixels with just some simple addition and rounding.

Adding matrices with numpy would probably be much faster than doing some inefficient iterating over every pixels and this would avoid the segmentation of the image.

The second solution is a bit more complex but could possibly offer much better results. The idea is to have a two big height x width x 3 array M and D, M containing int ids, and D containing float distances. The layers of the M matrix would correspond to ids of the 3 closest MotionFeature to a given pixel while the layers of the D matrix would correspond to the effective distance to those feature. The idea is to compute a distance matrix F of size N x N for a given feature (using a 2D gaussian or something similar), and to replace the value in the M and D matrix for each pixel where the distance is better (using some vectorized float comparison to generate a matrix of booleans)

At the end, you would end up with the 3 closest feature for each pixel, and you could compute the corresponding motion, and the corresponding values for the previous and next pixels, and you could choose or mix those values in function of how much the two (previous and next) colors corresponds.

To improve the results even more, improving the motion of each pixel in the same way the motion-flow algorithm does could be a good idea.

A third idea that I will only briefly evoke was to regroup features in areas and form triangles to be able to use a GPU to generate the frame faster. (using the original previous and next frames as texture transformed the right way to match)

5. Main driver

There is no real main file available right now, but there is two files that allows you to run and test the algorithm. There's test.py to generate one frame from pngs and there's testVideo.py to double the framerate of a video file.

Since the algorithm is very very slow, I do not recommend to run the algorithm on a big video. Even a 540p video of 4 seconds at 24 fps takes about an hour to be processed.

To double the framerate of a video (located anywhere), you can use:

python testVideo.py NAME_OF_THE_VIDEO_FILE

Please note that the algorithm does not have anything to detect and deal with scene changes yet.

To generate an interpolated frame from a sequence of png called "blades (i).png" (i from n to n+5) located in the input/ folder, you can modify the value n in the file test.py to define which frame id you want to start at and run:

python test.py blades

All the code is available at https://github.com/Zyfarok/swiftframes/tree/delivery

RESULTS

The best way to see the results are two short video where I doubled the frame rate. Those two videos are in the input folder:
https://github.com/Zyfarok/swiftframes/tree/delivery/input

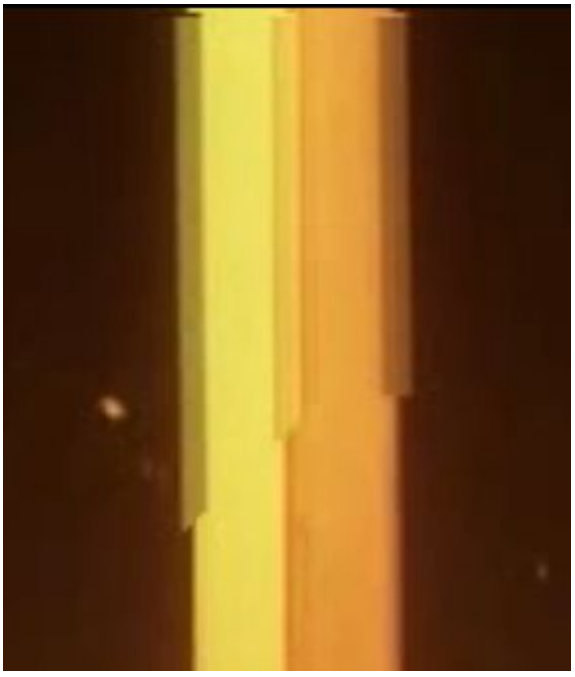
The one ending with -out.avi are the frame-doubled ones and the other ones are the original videos. The quality of the frame-doubled one is pretty low since the encoder used (XVID) is pretty bad and I couldn't afford to dive into this world to encode in a better quality but it's probably good enough to see the artifacts and errors that appears.

In the video you can clearly see problems in three situations:

- Not enough MotionFeature where found to interpolate. In this case the images are just blend together (it's probably good enough in some cases since the absence of features suggest a blurry image)
- Features were found but some imprecisions on the position or the lack of density of the motion field causes artifacts which look like some kind of duplication effect.
- Similar and aligned features of the object were matched the wrong way and generate artifacts (the regular lights and posts can generate intermediate lights and posts in between the original ones)

The quality of the interpolation is highly improved compared to the first results where motion coherence was not implemented.

Example of duplication effect:



Example of aligned features problem:

CONCLUSION

To summarize what I did in this project, I spent a lot of time thinking and designing differents algorithms and techniques for motion estimation from feature matching using motion coherence and also for interpolation from the obtained motion vectors. The time I spent at coding was also quite high and I was slowed down by different factors like python collections and typing and the complexity of the algorithms which used a lot's of collections of collection.

Now, I would say that I'm not fully satisfied by the results for mainly 2 reasons:

- First, there's still too much errors in the MotionMatches compared to how much feature I keep after the filter and the distance formula could probably be improved to filter them. Another thing that could improve those results is to use a homemade detector and descriptor that would be specialised in video motion to increase the initial good matching rate. Even though the feature detectors and descriptors I use are pretty good, they are not optimised for the specific use of motion in videos and taking that into account could actually improve the results. Additionally to the multiple frame motion coherence technique I used, I could also try to do a local motion coherence in order to detect MotionMatches that have a totally different motion direction and intensity than any other MotionMatch in order to filter them out which should help a lot in fixing the aligned similar features problem. The last solution I explained in the "Alternative interpolation methods" above could also help fixing the duplication effects.
- The second reason is about speed. Even though speed was not my main objective and I was mostly aiming on interpolation quality, having a speed that looks realistically low enough to be able to process a 25 minutes episode in less than a week would have been great. Some optimisations and parallelisation on both the feature matching and interpolation part could be done, to improve the speed.

Even though I'm not fully satisfied, I have at least proven that motion compensated frame interpolation from feature matching is doable and I'm thinking about potentially continuing this project alone since I think there's still a lot to investigate, especially in the interpolation part, and I would like to see how good and fast an algorithm using feature matching can be. Another thing that could potentially be investigated is the apparition and disapparition of features in order to detect appearing/disappearing object and handle them correctly.

APPENDIX

The main function that generates the motion vectors (MotionMatches) from the original images is the computeMotion function in *motion.py*.

The main function that generates an interpolated frames from the MotionFeatures is the *gen_inter_frame* function located in *interpolation.py*.

Those two function and files are the main part of the project, and most of other files are there to run/test those functions (*test.py* and *testVideo.py*), or define useful tools (like the MotionMatch and MotionFeature classes) for the work.

matching-tests.py and detector-tests.py were used to try different combinations of detectors, descriptors and matchers in order to compare the quality and speed of the matches.

To use the test.py file to interpolate a png file, you should first place a sequence of files using the “IMG_NAME (i).png” pattern, where i is the id of the frame in the sequence. Those file should be placed in the input folder and then you should make sure the value of “n = 1” corresponds to the frame you want to start at, and then you can use this script using:

python test.py IMG_NAME

And this script will generate the image between n and n+1 using a definite number of input images (5 by default).

Using the testVideo.py file is simpler since you don’t need to place the video in any particular folder, but make sure you use a pretty short and low resolution video if you don’t want to wait for ours. Simply using:

python testVideo.py NAME_OF_THE_VIDEO_FILE

Will generate the motion and then interpolate the given video. The algorithm saves the intermediate computed motion to a file so that you can change the first “if(True)” to “if(False)” to directly restart at the image interpolation if you need to for any particular reason.

REFERENCES

- SmoothVideo Project: https://www.svp-team.com/wiki/Main_Page
- SSA subtitles: https://www.matroska.org/technical/specs/subtitles/ssa.html
- OpenCV: https://opencv.org/
- AGAST feature detector: http://www.i6.in.tum.de/Main/ResearchAgast
- BRIEF feature descriptor: https://www.cs.ubc.ca/~lowe/525/papers/calonder_eccv10.pdf