

# Sadržaj

<b>1 Uvod</b>	<b>3</b>
1.1 Video sekvence . . . . .	3
1.2 Interpolacija u video sekvenci . . . . .	4
1.3 Primjene video interpolacije . . . . .	5
<b>2 Osnovne tehnike</b>	<b>6</b>
2.1 Duplikacija frejmova . . . . .	6
2.2 Linearna interpolacija . . . . .	7
<b>3 Optički tok</b>	<b>8</b>
3.1 Uvod . . . . .	8
3.1.1 Kernel konvolucija . . . . .	9
3.2 Uparivanje blokova . . . . .	10
3.2.1 PSNR . . . . .	12
3.2.2 Potpuna pretraga . . . . .	13
3.2.3 Trostepena pretraga . . . . .	13
3.2.4 Četverostepena pretraga . . . . .	14
3.2.5 Dijamantna pretraga . . . . .	15
3.2.6 Adaptive Rood Pattern Search - ARPS . . . . .	15
3.3 Fazna korelacija . . . . .	16
3.4 Diferencijalne metode . . . . .	18
3.4.1 Uvod . . . . .	18
3.4.2 Lucas-Kanade metoda . . . . .	19
3.4.3 Horn-Schunck metoda . . . . .	21
<b>4 Uklanjanje grešaka</b>	<b>23</b>
4.1 Neispravni vektori pomaka . . . . .	23
4.1.1 Detekcija neispravnih vektora pomaka . . . . .	24

4.1.2	Ispravljanje neispravnih vektora pomaka . . . . .	25
4.2	Izglađivanje granica između blokova . . . . .	25
<b>5</b>	<b>Implementacija interpolatora korištenjem OpenCV biblioteke</b>	<b>27</b>
5.1	OpenCV . . . . .	27
5.2	Implementacija . . . . .	29
5.3	Qt . . . . .	34
5.4	Rezultati interpolacije . . . . .	34
<b>6</b>	<b>Zaključak</b>	<b>39</b>

# Uvod

Cilj ovog rada jeste da objasni osnovne pojmove vezane za video sekvence, te samo neke od brojnih i raznovrsnih algoritama koji se u praksi koriste za video interpolaciju. Objasniti ćemo šta je to tačno video sekvenca, optički tok, interpolacija, framerate, i još mnogo toga. Algoritmi video interpolacije su veoma specijalizovani, te imaju relativno usko podršje primjene. Međutim, ipak su izuzetno korisni, te ćemo se sa konkretnim područjima primjene uskoro upoznati. Prije toga bi ipak bilo korisno napraviti kratak uvod u video sekvence, te u to kako ih računari vide.

## 1.1 Video sekvence

Ljudi svijet oko sebe vide kao sliku koja se kontinuirano mijenja. To jeste, svaki vremenski trenutak je različit od prošlog, koliko god da je blizu. Ljudi i objekti se kreću "glatko", bez potpuno naglih pokreta. Također, slike koje vidimo su prostorno kontinualne, prividno beskonačno detaljne. Računari, s druge strane, moraju slike i video fajlove čuvati u ograničenoj količini memorije, što znači da mora doći do kompromisa. Veća količina detalja zahtijeva više memorije za čuvanje. Svaka slika ima samo konačnu količinu detalja, što znači da se sastoji od konačnog broja piksela. Piksela (od eng. *picture element*) je najmanji djelić slike. Dva najčešća tipa piksela, pri čemu ćemo mi u ovom radu koristiti oba, su 1-komponentni i 3-komponentni (nekada se umjesto termina komponenta koristi termin kanal). Jednokomponentni piksel je djelić slike koji je jednobojan. 1-komponentni pikseli su gotovo isljučivo monohromatski, tj. njihova boja može biti crna, bijela, ili neka nijansa sive. 3-komponentni pikseli, s druge strane, se sastoje od 3 jednobojne komponente, koje zovemo podpikseli ili subpikseli. Ako je piksel oblika kvadrata, onda su podpikseli oblika pravougaonika (koji su po pravilu izduženi vertikalno). Na svim modernim ekranima, boje podpiksela su crvena, zelena i

plava. Ovaj raspored ćemo ubuduće zvati RGB (od eng. *Red-Green-Blue*). Drugi tipovi podpixela također postoje, koji se moraju konvertovati u RGB za prikaz na ekranima. Svaki ekran i svaka slika se sastoje od određenog broja piksela, koji su po pravilu raspoređeni u matricu. Ako na monitoru gledamo sliku koja ima više horizontalnih ili vertikalnih piksela od našeg monitora, onda ne možemo vidjeti sve detalje te slike. Koji detalji će biti izostavljeni zavisi od konkretnog algoritma korištenog za skaliranje slike, o kojima nećemo pričati u ovom radu.

Objasnili smo kako računari rješavaju problem prostorno kontinualnih slika, sada ćemo objasniti video sekvene. Isto kao što smo sliku rastavili u piksele, video ćemo rastaviti u *frejmove*. Frejmovi su pojedinačne sličice (ili okvirovi) od kojih se sastoje video sekvenca (sekvenca upravo zato što je to sekvenca frejmova). Svaki frejm provede određeno vrijeme na ekranu, nakon čega se mijenja narednim frejmom. U pravilu, to vrijeme je za svaki frejm isto (ako je to moguće), da bi video sekvenca koju gledamo izgledala što prirodnije. Broj frejmova koje vidimo svake sekunde (odnosno broj recipročan broju sekundi koji svaki frejm provede na ekranu) se zove *framerate*, a korištena jedinica je *frame per second*, skraćeno *fps*. Tako je maksimalni framerate većine ekrana 60 fps, framerate filmova je gotovo uvijek 24 fps, framerate igrica zavisi od mnogo faktora, ali je u većini slučajeva 30 ili 60 fps.

## 1.2 Interpolacija u video sekvenci

Sada kada znamo šta je framerate, možemo objasniti video interpolaciju. Naime, interpolacijom video sekvence se dobivaju novi frejmovi između postojećih. Taj novi frejm će biti veoma sličan svojim susjednim, jer u suštini predstavlja "među korak". Izlaz algoritma za video interpolaciju će biti video veoma sličan početnom, sa dodatnim frejmovima koji su generisani iz originalnih. U matematičkom smislu, interpolacija predstavlja izračunavanje novih vrijednosti na osnovu okolnih poznatih, što zvuči veoma slično video interpolaciji. Međutim, zbog svoje prirode, algoritmi interpolacije video sekvenci se potpuno razlikuju od onih korištenih u matematici.

### 1.3 Primjene video interpolacije

Kao što smo već rekli, izlaz algoritma za video interpolaciju jeste novi video koji je veoma sličan originalnom. Ako želimo, možemo koristiti taj video, koji će zbog svog višeg framerate-a izgledati prirodnije (pod uslovnom da je kvalitetno interpoliran) i glade. Tako bi sportski sadržaj mogao biti sniman kamerom visoke rezolucije, ali relativno niskog fremerate-a, nakon čega bi frejmovi mogli biti interpolirani da bismo dobili video koji je i visoke kvalitete i visokog framerate-a. Također bismo mogli koristiti video interpolaciju za video pozive, gdje je mrežna propusnost često ograničavajući faktor. Možemo snimiti video niskog framerate-a, koji ne zahtijeva visoku mrežnu propusnost, te interpolirati dolazeći video. Naravno, u oba ova slučaja će dobiteni rezultat biti nižeg kvaliteta nego video sniman visokim framerate-om, ali uz kvalitetne algoritme razlike neće biti značajna.

Postoje već projekti koji koriste video interpolaciju za kreiranje *slow-motion* video sekvenci. Umjesto zadržavanje ukupne dužine video sekvence povećavanja framerate-a, možemo povećati dužinu i zadržati framerate. Jedan primjer bi bio [Butterflow](#).

# Osnovne tehnike

Sada ćemo objasniti "trivijalne" algoritme interpolacije: duplikacija frejmova i linearna interpolacija. Linearna interpolacija bi se čak mogla koristiti samostalno za kreiranje interpoliranih frejmova, međutim, rezultat bi bio izuzetno mutan. U narednim odjeljcima ćemo objasniti načine rada i područja primjene ovih algoritama.

## 2.1 Duplikacija frejmova

Kao što samo ime kaže, duplicitirani frejm je identičan jednom od njegovih susjednih frejmova (prethodnom ili narednom). Duplicirani frejmovi se koriste u situacijama kada nema smisla koristiti standardne tehnike interpolacije. Naime, neki frejmovi se jednostavno potpuno razlikuju od svojih prethodnika. To se dešava pri promjeni scene ili bilo kakvoj nagloj promjeni slike. Frejm koji se značajno razlikuje od svog prethodnika nazivamo *keyframe*. Ako ćemo između svaka 2 susjedna frejma ubacivati određeni broj interpoliranih frejmova, onda moramo ubaciti nešto i prije keyframe-a. Jednostavno ćemo napraviti onoliko kopija posljednjeg frejma prethodne scene ili novog keyframe-a koliko nam treba interpoliranih frejmova. Ako pokušamo ipak interpolirati novi frejm između dva potpuno različita, dobit ćemo nešto što neće ličiti ni na jedan od dva frejma, te će biti veoma uočljivo gledaocima. Zbog toga je najbolje samo napraviti kopije. Alternativa tom pristupu bi bilo jednostavno preskočiti te frejmove i ostaviti ih kao susjedne. Nedostatak ovog pristupa je potencijalno stvaranje problema oko sinhronizacije ako uz video sekvencu imamo prateći audio zapis ili tekst prijevoda. Ta "rupa" u frejmovima bi značila da će audio zapis i prijevod kasniti za video zapisom. Tako da ćemo mi ipak koristiti duplikaciju prethodnog frejma.

## 2.2 Linearna interpolacija

Kao što ćemo vidjeti kasnije, tehnike interpolacije koje ćemo mi koristiti neće kreirati cjelokupan frejm. Postojat će velika područja frejma za koje algoritam jednostavno nije dao nikakve podatke. Da bismo izbjegli veoma očigledna "prazna" područja crnih piksela, moramo koristiti neku drugu tehniku koja će zagarantovano davati podatke o svakom pikselu. A najjednostavnija takva tehnika (osim duplikacije) jeste linearna interpolacija, koja radi na sljedećem principu:

Neka su  $A$  i  $B$  vektori kolone koji predstavljaju dva piksela (svaki član vektora po jednu od tri komponente), a  $C$  je piksel interpoliranog frejma (kojeg želimo generisati). Piksel  $A$  je piksel na nekoj poziciji prethodnog, a piksel  $B$  je piksel na toj istoj poziciji narednog frejma. Mi sada želimo da vidimo koji će se piksel nalaziti na toj poziciji u interpoliranom frejmu. Realan broj  $\alpha$  ( $0 < \alpha < 1$ ) predstavlja relativnu poziciju interpoliranog frejma između rethodnog i narednog. To jeste, ako između ta dva frejma generišemo  $n$  novih, a frejm koji želimo generisati je  $k$ -ti po redu, tada je:

$$\alpha = \frac{k}{n+1} \quad (2.1)$$

Sada je vrijednost piksela  $C$  interpoliranog frejma:

$$C = (1 - \alpha) * A + \alpha * B \quad (2.2)$$

Ako između svaka dva frejma jednostavno želimo generisati jedan novi, tada je  $\alpha = \frac{1}{2}$ , te će svaka komponenta generisanog piksela biti aritmetička sredina komponenti piksela  $A$  i  $B$ .

Kao što ćemo kasnije vidjeti, pri generisanju novog frejma držat ćemo "matricu posjećenosti", koja će za svaki piksel držati jednu binarnu vrijednost. Sve vrijedosti će na početku biti postavljene na 0 (false). Pri upisivanju novog piksela, odgovarajuću vrijednost matrice ćemo postaviti na 1 (true). Na kraju, piksele na pozicijama sa vrijednošću 0 ćemo generisati na drugi način, konkretno upravo linearom interpolacijom.

# Optički tok

## 3.1 Uvod

Sada ćemo se početi baviti složenijim tehnikama interpolacije koje mogu davati mnogo bolje rezultate od linearne interpolacije. U idealnom slučaju, rezultirajući frejmovi će izgledati potpuno prirodno. Postoji više klase algoritama koji su u praksi korišteni, od kojih ćemo mi u ovom poglavlju objasniti tri: Tehnike zasnovane na faznoj korelaciji, upoređivanju blokova, i rješavanju diferencijalnih jednačina. Zajedničko svim ovim algoritmima jeste da njihov izlaz neće biti novi interpolirani frejm, već *polje optičkog toka*. Cilj ovog poglavlja jeste da odgovori na sljedeća pitanja:

1. Šta su vektori pomaka?
2. Šta je optički tok?
3. Kako možemo koristiti poznavanje optičkog toga za kreiranje interpoliranih frejmova?
4. Kojim metodama možemo izračunati optički tok?

Neka su zadana dva susjedna frejma,  $A$  i  $B$ , između kojih želimo interpolirati novi frejm. Intuitivno gledajući, neki objekti frejma  $A$  će se također nalaziti u frejmu  $B$ , samo na nekoj drugoj poziciji, blizu svoje originalne. Drugim riječima, postoje vektori pomaka koji odgovaraju tim objektima, koji imaju dvije komponente i dužkojih su objekti prividno pomjereni između frejmova  $A$  i  $B$ .

Određivanje vektora pomaka za objekte bi zahtjevalo traženje objekata u frejmovima, što je izuzetno zahtjevno samo po sebi. Umjesto toga, svi algoritmi koje ćemo objasniti će pridruživati pojedinačne vektore pomaka svakom pikselu jednog od frejmova ( $A$  ili  $B$ ), neki algoritmi mogu odrediti

koji je od ta dva slučaja korisnije razmatrati). Algoritam će jednostavno dobiti dva frejma, i za svaki piksel dati jedan vektor, njegov vektor pomaka. Skup svih vektora pomaka piksela jednog frejma na drugi nazivamo poljem optičkog toka, pri čemu optički tok možemo definisati kao prividno kretanje objekata u video sekvenci.

Najveći dio posla pri generisanju interpoliranog frejma upravo jeste traženje kvalitetnog optičkog toka. Nakon toga, proces je relativno jednostavan:

1. Svaki piksel pomjeriti duž svoga vektora pomaka pomnoženog sa  $\alpha$  (realan broj dobiven na isti način kao  $\alpha$  u dijelu u linearnoj interpolaciji) te ga spasiti na tu poziciju.
2. Polja za koja nije pronađen niti jedan piksel popuniti korištenjem linearne interpolacije.

### 3.1.1 Kernel konvolucija

Kernel konvolucija je tehnička često korištena za obradu slike, i ima veliki broj različitih primjena. Neke od njih uključuju detekciju ivica, izoštravanje, zamućivanje i druge razne filtere koji mogu biti primijenjeni na slike. Ulazi kernel konvolucije su slika i sam kernel, dok je izlaz obrađena slika. Kerneli funkcionišu na samo jednom dvodimenzionalnom signalu, tako da ćemo mi posmatrati samo intenzitete pojedinih piksela, ne njihove odvojene komponente (crvena, zelena i plava).

Kernel nije ništa drugo nego matrica realnih brojeva. Za naše potrebe, dimenzije matrice će morati biti neparni brojevi (vidjet ćemo zašto je lakše raditi sa takvim kernelima). Neka je zadana matrica intenziteta piksela  $A$ , kernel  $K$  i izlazna matrica  $B$ . Dimenzije izlazne matrice su iste kao dimenzije ulazne. Neka su dimenzije kernela (visina i širina)  $p$  i  $q$ . Svaki piksel izlazne matrice  $B_{i,j}$  računamo narednom formulom:

$$B_{i,j} = \sum_{y=-p'}^{p'} \sum_{x=-q'}^{q'} A_{i+y,j+x} * K_{y+p'+1,x+q'+1}$$

pri čemu su:

$$p' = (p - 1)/2$$

$$q' = (q - 1)/2$$

Zamislimo da smo postavili kernel iznad piksela sa koordinatama  $i, j$ , tako da element u sredini kernela ima upravo koordinate  $i, j$ . Ovdje vidimo zašto smo postavili ograničenje da dimenzije kernela moraju biti neparni brojevi. Zatim ćemo pomnožiti svaki element kernela sa elementom matrice  $A$  koji se nalazi tačno ispod njega. Zbir svih tih proizvoda upisujemo u element  $B_{i,j}$ .

Korištenjem različitih kernela možemo dobiti različite efekte. Ispod slijede primjeri nekih od češće korištenih:

$$\text{Detekcija ivica: } \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\text{Zamućivanje: } \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

$$\text{Izoštravanje: } \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Neki od algoritama koje ćemo objasniti koriste kernel konvoluciju, u opisu algoritama ćemo navesti i konkretne korištene kernele.

## 3.2 Uparivanje blokova

Prva klasa algoritama koje ćemo proučavati kreću od iste osnovne ideje: Podjeliti prvi frejm na blokove, te naći vektor pomaka svakog bloka iz prvog frejma u drugi. Ovi algoritmi značajno pojednostavljaju problem tako što pretpostavljaju da će svi pikseli unutar jednog bloka imati iste vektore pomaka. Ovaj pristup može dovesti do grešaka, ali čini izračunavanje vektora pomaka jednostavnim i brzim.

Ako nam je cilj samo kreirati jedan novi frejm između svaka dva postojeća, svaki blok ćemo pomjeriti duž pola izračunatog vektora pomaka. Ako nam je cilj interpolirati dva frejma između dva postojeća, blok ćemo pomjeriti duž jedne trećine vektora pomaka za prvi, i dvije trećine za drugi interpolirani frejm, itd. Cilj slijedećih algoritama jeste uparivanje blokova prvog frejma sa blokom iste veličine u drugom frejmu. Međutim, postoji nekoliko pitanja na koja moramo odgovoriti prije nego što možemo primijeniti ove algoritme:

- Koju veličinu bloka ćemo koristiti?
- Koliki će biti prozor pretrage, odnosno koliko će se svaki blok moći maksimalno pomjeriti između prvog i drugog frejma?
- Koji je kriterij sličnosti dva bloka?
- Kako odrediti uspješnost uparivanja?

U praksi se koriste blokovi veličine 16x16 piksela, te prozor pretrage veličine 30x30 piksela. To znači da pretpostavljamo da se između dva susjedna frejma blokovi neće pomjeriti više od 7 piksela u bilo kojem od 4 kardinalna smjera. To nam daje 225 mogućih lokacija za svaki blok. Naravno, ne postoji definitivna, optimalna veličina bloka ili prozora pretrage za sve slučajeve. Manje blokove je brže uporediti, ali je njihov broj veći, te je veća vjerovatnoća da će dva bloka biti slučajno veoma slična. Veći prozor pretrage nam omogućava pronalaženje ispravnih vektora pomaka i u slučaju kada se desi pomak veći od 7 piksela, ali povećava vrijeme potrebno za izračunavanje te, slično kao u slučaju blokova, povećanjem prozora pretrage se povećava i vjerovatnoća uparivanja dva bloka koji su slični, ali zapravo ne predstavljaju isti blok. U svim slijedećim algoritmima ćemo koristiti blokove i prozore pretrage navedene veličine.

Sljedeće pitanje se tiče kriterija sličnosti dva bloka. Svaki blok je sastavljen od 256 piksela, koji se sastoje od 3 komponente: crvene, zelene, i plave. Svaka komponenta ima cijelobrojnu jačinu u rasponu od 0 do 255, uključivo. Za upoređivanje blokova nećemo koristiti sve 3 komponente piksela, nego ćemo izračunati njihov intenzitet, koji se dobija kao aritmetička sredina intenziteta crvene, zelene i plave komponente. Ovo se radi zato što jednostavno ne bismo imali koristi od odvojenog upoređivanja sve 3 komponente. Naravno, postoji vjerovatnoća da ćemo upariti dva piksela koji imaju slične intenzitete, a ne i iste boje (npr. potpuno crveni i potpuno zeleni piksel imaju iste intenzitete), ali računanjem vektora pomaka za sve 3 komponente zasebno bismo često dobili 3 različita vektora pomaka, što nam ne daju nikakvu korisnu informaciju. Zbog toga je bolje samo uzimati u obzir intenzitete piksela.

Korišteni kriteriji sličnosti blokova su veoma jednostavnji. Jedan je *Mean Absolute Difference (MAD)*, odnosno *Srednja Apsolutna Razlika*. Ova mjera nije ništa drugo nego suma apsolutnih vrijednosti razlika intenziteta odgo-

varajućih piksela u blokovima, podijeljena sa veličinom bloka. Drugim riječima, zadana je formulom

$$MAD = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |A_{ij} - B_{ij}|$$

Pri čemu  $N$  predstavlja visinu i širinu bloka (u našem slučaju 16), dok  $A_{ij}$  i  $B_{ij}$  predstavljaju vrijednosti piksela na koordinatama  $(i, j)$  (sa početkom u gornjem lijevom uglu bloka) prvog, odnosno drugog razmatranog bloka.

Druga, veoma slična mjera jeste *Mean Squared Error (MSE)*, odnosno *Srednji Kvadrat Greške*. Umjesto uzimanja apsolutne vrijednosti razlika piksela, ova mjera kvadrira razlike piksela, čime se više kažnjavaju veće razlike. *MSE* je zadana formulom

$$MSE = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (A_{ij} - B_{ij})^2$$

Mi ćemo ove funkcije ubuduće zvati zajedničkim imenom *funkcije cijene*. Cilj svih algoritama u ovom poglavlju jestе minimizacija funkcije cijene, bilo *MAD* ili *MSE*.

Još jedno veoma važno područje primjene algoritama uparivanja blokova (i zapravo područje gdje se najviše primjenjuju) jestе kompresija video sekvenci. O ovom poglavlju se obrađuju algoritmi koji su korišteni u standardima H.261, H.262 i H.263. U novijim standardima (pri čemu je najnoviji H.265, čija je prva verzija izašla 2013. godine) korišteni su napredniji algoritmi, koji u ovom radu neće biti obrađeni.

### 3.2.1 PSNR

Sada ćemo se osvrnuti na neke od osnovnih algoritama uparivanja blokova. Pretpostavit ćemo da su blokovi veličine 16x16 piksela, a prozori pretrage veličine 30x30 piksela, te da su svi pikseli monohromatski (crno-bijeli). Od svih slijedećih algoritama, samo prvi pronalazi optimalno uparivanje. Svi ostali ostali algoritmi su aproksimacije. Preciznije rečeno, postoji 225 mogućih lokacija gdje se počtni blok može nalaziti u prozoru pretrage. Definišimo matricu  $C_{15 \times 15}$ , pri čemu svakom elementu matrice  $C$  odgovara vrijednost funkcije cijene (*MAD* ili *MSE*) koju dobijemo ako postavimo blok na to

mjesto u prozoru pretrage (drugim riječima, elementu  $1,1$  odgovara vrijednost funkcije cijene koju dobijemo ako blok postavimo u gornji lijevi ugao prozora pretrage, pomjeranjem bloka dobijamo druge vrijednosti matrice). Aproksimativni algoritmi pretpostavljaju da ova matrica ima jednu najmanju (optimalnu) vrijednost, i da vrijednosti elemenata matrice monotono rastu kako se udaljavamo od ovog elementa. Kontinualni analogon ove osobine bi bila funkcija  $2$  realne promjenljive koja ima samo jedan lokalni minimum, koji je ujedno i globalni minimum. Za funkciju koja ima ovu osobinu kažemo da je *unimodalna*. U slučaju kad ova osobina postoji, možemo pronaći minimalnu vrijednost matrice jednostavnim spuštanjem po gradijentu, odnosno, počevši od proizvoljnog početnog elementa, u svakom koraku pređemo na bilo koji od manjih elemenata dok ne dođemo do nekog koji je manji od svih svojih susjednih elemenata.

Naravno, ne postoji ništa da nam garantuje ovu osobinu, što znači da će aproksimativni algoritmi samo u rijetkim slučajevima pronaći optimalno rješenje.

### 3.2.2 Potpuna pretraga

Ovaj algoritam se zasniva na potpunom pretraživanju svih mogućih lokacija za blok, te pronalaženju lokacije koja daje najmanju vrijednost funkcije cijene. Postoji  $225$  mogućih lokacija za blok unutar prozora pretrage, a za izračunavanje vrijednosti funkcije cijene moramo uporediti  $16 * 16 = 256$  piksela. To nam daje  $225 * 256 = 57600$  upoređivanja piksela po bloku. HD video (dimenzija  $1280 \times 720$  piksela) sadrži frejmove koji se sastoje od  $3600$  blokova, što nam ukupno daje preko  $200$  miliona upoređivanja piksela za svaki frejm video sekvene. Tako da nije teško vidjeti zašto se ovaj algoritam ne koristi u praksi.

### 3.2.3 Trostepena pretraga

Trostepena pretraga se oslanja na osobinu unimodalnosti matrice vrijednosti funkcije cijene. Ovaj algoritam je ovdje korišten isključivo kao uvod, jer ostvaruje veoma slabe rezultate u praksi. Algoritam je mnogo brži od potpune pretrage te koristi slične tehnike kao bolji algoritmi koji će biti obrađeni u narednim dijelovima.

Neka je zadan prozor pretrage veličine  $30 \times 30$  piksela i blok veličine  $16 \times 16$  piksela. Trostepena pretraga radi tako što u svakom koraku izračuna funkciju

cijene bloka na 9 lokacija u prozoru pretrage. Tih 9 lokacija su trenutni centar i sve kombinacije lokacija koje dobijemo ako blok pomjerimo za  $-S$ ,  $0$  ili  $S$  piksela po x i y osama (postoji 9 kombinacija, od kojih je jedna  $(0, 0)$ , što predstavlja naš centar).  $S$  predstavlja trenutnu *veličinu koraka*. Nakon izračunavanja funkcije cijene na svih 9 lokacija, izaberemo najbolju (tj. lokaciju koja daje najmanju vrijednost funkcije cijene), nju odredimo za novi centar, i smanjimo  $S$  na jednu polovinu trenutne vrijednosti. U prvom koraku,  $S$  će imati vrijednost 4, u drugom 2, a u trećem 1, dok će centar u prvom koraku predstavljati stvarni centar prozora pretrage. S obzirom da se na početku blok nalazi 7 piksela daleko od ivica prozora pretrage, a u 3 koraka (sa pomakom od 4, 2 i 1 pikselom) se moguće pomjeriti najviše tačno 7 piksela, blok može zauzeti bilo koju poziciju u prozoru pretrage.

U svakom od 3 koraka, imamo 9 različitih mogućih pozicija bloka, tako da izgleda da moramo izračunati funkciju cijene bloka 27 puta. Međutim, centar u drugom i trećem koraku je već izračunat u prethodnom koraku, tako da je zapravo potrebno izračunati samo 25 različitih funkcija pretrage, što je 9 puta manje od 225 kod potpune pretrage, što naravno predstavlja veoma značajno ubrzanje. Međutim, kao što je prije rečeno, ovaj algoritam generalno ne izračunava zadovoljavajuće vektore pomaka, tako da se u praksi ne koristi.

### 3.2.4 Četverostepena pretraga

Četverostepena pretraga radi na istom principu kao trostepena, samo sa drugačijim izborom veličine koraka. U prvom koraku  $S$  iznosi 2, odnosno lokacije koje pretražujemo će biti  $(0, 0)$ ,  $(0, 2)$ ,  $(2, 0)$ ,  $(2, 2)$ ,  $(-2, 0)$ ,  $(0, -2)$ ,  $(-2, -2)$ ,  $(2, -2)$  i  $(-2, 2)$ . U slučaju da lokacija  $(0, 0)$  daje najmanju vrijednost funkcije cijene, prelazimo na četvrti korak. Ovo vrijedi i u drugom i trećem koraku, za trenutni centar pretrage. Drugi i treći korak također imaju veličinu koraka  $S = 2$ , dok za četvrti korak vrijedi  $S = 1$ . Ukoliko prođemo kroz sva četiri koraka, vidimo da je maksimalni pomak jednak  $2+2+2+1 = 7$ .

Ovaj pristup zahtijeva od 17 do 27 upoređivanja blokova. Razlog ovome jeste što je veličina koraka u prva 3 koraka ista, što znači da će se značajan broj lokacija poklapati, te ih ne moramo računati ponovno. Iako je manje konsistentna od trostepene pretrage (koja uvijek zahtijeva 25 upoređivanja blokova), četverostepena pretraga će u većini slučajeva biti brža. Druga prednost je što četverostepena pretraga bolje detektuje male vektore pomaka (1 do 2 piksela u jednom smjeru), zato što je početni korak manji ( $S = 2$  u

odnosu na  $S = 4$ ), kao i zato što u slučaju da je najbolja lokacija u centru, automatski prelazimo u četvrti korak, koji može samo još podesiti vektor pomaka za 1 piksel u bilo kojem smjeru.

### 3.2.5 Dijamantna pretraga

Dijamantna pretraga je slična četverostepenoj pretrazi, uz dvije modifikacije. Prva je da oblik koji prave lokacije koje pretražujemo nije više kvadrat čije su stranice paralelne sa koordinatnim osama, nego kvadrat rotiran za  $45^\circ$ (čiji oblik podsjeća na dijamant ili romb), a druga da ne postoji gornja granica na maksimalan broj koraka. Naravno, algoritam će uvijek terminirati jer postoji 225 mogućih lokacija, a funkcija cijene za svaku lokaciju se računa samo jednom, druga razlika samo znači da algoritam neće automatski preći na manju veličinu koraka nakon nekog broja koraka, nego se to radi samo ako, od 9 lokacija koje razmatramo u trenutnom koraku, najbolji rezultat daje trenutni centar. Do posljednjeg koraka, veličina koraka  $S$  iznosi 2, što u prvom koraku daje sljedeće lokacije koje trebamo pretražiti:  $(0, 0)$ ,  $(2, 0)$ ,  $(0, 2)$ ,  $(-2, 0)$ ,  $(0, -2)$ ,  $(1, 1)$ ,  $(1, -1)$ ,  $(-1, 1)$  i  $(-1, -1)$ . U posljednjem koraku, veličina koraka  $S$  iznosi 1, što znači da ćemo posmatrati samo trenutni centar i 4 lokacije direktno iznad, ispod, lijevo i desno od trenutnog centra (pomjerene po jedan piksel u kardinalnim smjerovima).

Garanciju da će algoritam eventualno preći na posljednji korak nam daje činjenica da ovaj algoritam (i oba prethodna) uvijek prelazi na lokaciju koja daje bolju vrijednost funkcije cijene (ili ostaje na trenutnoj lokaciji). S obzirom da postoji konačan (225) broj lokacija, a nikad ne prelazimo na lokaciju koja nam daje veću vrijednost funkcije cijene, eventualno ćemo doći do lokalnog minimuma (koji je u idealnom slučaju i globalni minimum).

### 3.2.6 Adaptive Rood Pattern Search - ARPS

Algoritam koji daje najbolje rezultate koji ćemo opisati u ovom poglavljju je ARPS. Za razliku od do sada opisanih algoritama, ARPS uzima u obzir ne samo vrijednosti piksela bloka i prozora pretrage, nego i prethodno izračunati vektor pretrage. Konkretno, da bismo koristili ARPS, potreban nam je jedan vektor pomaka za koji očekujemo da će vjerovatno biti sličan stvarnom vektoru pomaka trenutnog bloka. Mi ćemo koristiti vektor pomaka bloka koji se nalazi lijevo od trenutnog u tu svrhu. Ako se trenutni blok nalazi uz lijevu ivicu frejma (i ne postoji blok lijevo od njega), onda ćemo koristiti

vektor pomaka bloka iznad trenutnog. A u krajnjem slučaju, ako računamo vektor pomaka za prvi blok (u gornjem lijevom uglu frejma), možemo koristiti potpunu pretragu koja zagarantovano daje optimalno uparivanje (uticaj na performanse nije značajan jer potpunu pretragu vršimo samo jednom za jedan frejm).

Neka prethodni vektor pomaka iznosi  $(x, y)$ . Uvest ćemo pomoćnu promjenljivu  $h = \max(|x|, |y|)$ . U prvom koraku ćemo izračunati funkciju cijene na 6 lokacija:  $(0, 0), (h, 0), (0, h), (-h, 0), (0, -h), (x, y)$ . Odnosno, pretražujemo lokaciju  $(0, 0)$  (jer su nula vektori pomaka česti), vektor pomaka od kojeg smo krenuli  $(x, y)$ , te 4 lokacije koje su isto daleko od  $(0, 0)$  po tzv. *Manhattan udaljenosti*. Treba uzeti u obzir da je moguće da je  $x = 0$  ili  $y = 0$ , u kojem slučaju je potrebno pretražiti samo 5 lokacija (2 se poklapaju). Ako je  $x = y = 0$ , onda je riječ o samo jednoj lokaciji, i možemo odmah preći na drugi korak algoritma.

Drugi korak ARPS algoritma je isti kao drugi korak dijamantne pretrage. Od pretraženih lokacija izaberemo najbolju, zatim izračunavamo 4 lokacije direktno iznad, ispod, lijevo i desno od trenutne najbolje, te prelazimo na najbolju od te 4. Algoritam se završava kada je trenutna lokacija bolje od 4 koje je okružuju. Kao i uvijek, možemo ubrzati algoritam tako što ne računamo opet funkciju cijene za lokacije koje smo već obradili.

### 3.3 Fazna korelacija

Fazna korelacija je tehnika koju možemo koristiti za brzo pronalaženje mjesta preklapanja dviju sličnih slika, što nama upravo i treba. Možemo uzeti jedan blok piksela jednog frejma, te, koristeći faznu korelaciju, brzo pronaći gdje se taj blok piksela nalazi u drugom frejmu (ili barem neki drugi, veoma sličan blok). Direktnom primjenom formule bismo dobili algoritam koji je zapravo veoma sličan potpunoj pretrazi, koja je opisana u poglavljju o algoritmima uparivanja blokova. Međutim, možemo koristiti *Brzu Fourierovu Transformaciju* (FFT) za značajno ubrzanje. S obzirom na to da jedan korak algoritma uključuje izračunavanje Hadamardovog proizvoda dvije matrice (koji je jedino moguće izračunati ako su matrice istih dimenzija), ovaj algoritam je ograničen na traženje jednog ili više mjesta poklapanja dva bloka piksela iste veličine. Također, ako želimo koristiti neki od FFT algoritama, dimenzije blokova moraju biti stepeni broja dva. Preporučene dimenzije.

Fazna korelacija je tehnika zasnovana na korištenju algoritma za brzu

Fourierovu transformaciju za traženje mesta poklapanja dva bloka veoma efikasno. Osim korištenja većih blokova nego u prethodnim poglavljima, također ćemo koristiti blokove koji se preklapaju, što će nam dati više različitih kandidata za vektor pomaka jednog piksela. Od tih kandidata ćemo izabrati vektor koji trenutni pixel uparuje sa najsličnjim. Pri podjeli frejmova na blokove, cilj nam je da svaki pixel originalnog frejma ima barem jedan blok u kojem se nalazi u kojem je njegov upareni pixel. Ova metoda uparuje čitave blokove piksela sa drugim blokovima tako što traži mesta preklapanja u frekvencijskom domenu, što je moguće uraditi u linearном vremenu umjesto kvadratnom. S obzirom da je kompleksnost FFT algoritama loglinearna, traženje fazne korelacije također ima loglinearnu kompleksnost. Da bismo primijenili FFT algoritam, dimenzije blokova moraju biti stepeni broja 2. Preporučena veličina je 128x128 piksela. Diskretna 2-D Fourierova transformacija je opisana sljedećom formulom:

$$F(u, v) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} f(m, n) e^{-2\pi i(um+vn)}$$

Koraci algoritma:

1. Podijeli obje slike na preklapajuće blokove kako je opisano
2. Za svaki pixel i svaki blok u kojem se taj pixel nalazi, izračunaj 2D FFT piksela u tom bloku i u prethodnom i u narednom frejmu
3. Izračunaj Hadamardov proizvod matrice dobivene iz prethodnog, i kompleksno konjugovane matrice dobivene iz narednog frejma
4. Normalizuj rezultat
5. Izračunaj inverznu Fourierovu transformaciju
6. Vektor pomaka dobijemo kao koordinate maksimuma krajnje matrice

Proces može biti opisan sljedećom formulom:

$$F^{-1} \left\{ \frac{G(k)H^*(k)}{|G(k)H^*(k)|} \right\}$$

Pri tome  $F^{-1}$  predstavlja inverznu Fourierovu transformaciju,  $G(k)$  i  $H(k)$  su blok prethodnog i narednog frejma u frekvencijskom domenu,  $*$

je unarni operator kompleksne konjugacije, a proizvod matrica je zapravo Hadamardov proizvod, odnosno jednostavno proizvod pojedinačnih elemenata na istim pozicijama u dvije matrice.

## 3.4 Diferencijalne metode

### 3.4.1 Uvod

Da bismo objasnili ove metode, moramo prvo pojasniti *jednačinu optičkog toka*. Ova jednačina glasi:

$$I_x u + I_y v + I_t = 0$$

ili

$$\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} = 0$$

Jedan frejm, pri čemu samo posmatramo intenzitet piksela, a ne i njihove boje, možemo zamisliti kao realnu funkciju dvije realne promjenljive  $I(x, y)$ . Naravno, vrijednosti parametara  $x$  i  $y$  moraju biti prirodni brojevi (ili nula), te će sam rezultat funkcije biti cijeli broj u intervalu  $[0, 255]$ , međutim, uz ovakvo predstavljanje frejma možemo koristiti metode analize za rješavanje problema. Možemo čak čitav video predstaviti kao jednu jedinu funkciju tri realne promjenljive  $I(x, y, t)$ , pri čemu  $x$  i  $y$  predstavljaju koordinate piksela u frejmu, dok  $t$  predstavlja redni broj samog frejma. Iz ovakve funkcije možemo izvesti jednačinu optičkog toka.

Za zadani piksel  $I(x, y, t)$ , naš cilj je pronaći  $\Delta x, \Delta y, \Delta t$  takve da:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

Zatim ćemo  $I(x, y, t)$  aproksimirati Taylorovim polinomom prvog reda:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) \approx I(x, y, t) + \frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t$$

Koristeći prethodne dvije jednačine, vidimo da je:

$$I(x, y, t) \approx I(x, y, t) + \frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t$$

Odnosno:

$$\frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t \approx 0$$

Ako obje strane jednačine podijelimo sa  $\Delta t$  i uvedemo nove oznake:

$$u = \frac{\Delta x}{\Delta t}$$

$$v = \frac{\Delta y}{\Delta t}$$

Rezultat je upravo jednačina optičkog toka (pri čemu smo znak  $\approx$  zamjenili znakom  $=$ ). Ako nas samo zanimaju pikseli u narednom frejmu (odnosno ako je  $\Delta t = 1$ , jednačina postaje:

$$\frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} = 0$$

Vidimo da čak ni ovu pojednostavljenu jednačinu ne možemo jednoznačno riješiti jer imamo dvije nepoznate ( $\Delta x$  i  $\Delta y$ ). Mi ćemo u nastavku opisati dva algoritma koji na dva različita načina rješavaju ovaj problem. Ako su nam za zadani piksel poznate vrijednosti  $\Delta x$  i  $\Delta y$ , onda vektor  $(\Delta x \Delta y)$  predstavlja upravo traženi optički tok, koji možemo koristiti za kreiranje interpoliranih frejmova.

### 3.4.2 Lucas-Kanade metoda

Osnovna ideja iza Lucas-Kanade metode jeste korištenje više od jednog piksela. Korištenjem okoline piksela čiji optički tok pokušavamo izračunati, dobit ćemo više jednačina (npr. ako koristimo 3x3 područje oko zadanoj piksela dobit ćemo 9 nezavisnih jednačina). Naravno, sistem od 9 jednačina sa 2 nepoznate gotovo nikada nema jedinstveno rješenje, tako da Lucas-Kanade metoda pronalazi najbolje rješenje metodom najmanjih kvadrata. Nije nužno koristiti tačno 9 jednačina, metoda radi sa bilo kojim brojem piksela u okolini zadanoj.

Umjesto pisanja jednačine optičkog toka 9 puta za svih 9 piksela, možemo svih 9 predstaviti jednostavnom matričnom jednačinom:

$$S \begin{pmatrix} u \\ v \end{pmatrix} = \vec{t}$$

pri čemu je  $S$  9x2 matrica čiji su redovi vektori oblika

$$(I_x(x + p, y + q), I_y(x + p, y + q)), p, q \in (-1, 0, 1)$$

dok je  $\vec{t}$  vektor-kolona čiji su članovi  $-I_t(x + p, y + q), p, q \in (-1, 0, 1)$ . Cilj nam je pronaći  $u$  i  $v$  takve da je zbir kvadrata razlika lijevih i desnih strana jednačina minimalan.

Rješenje dobijamo tako što pomnožimo slijeva obje strane sa  $S^T$ :

$$S^T S \begin{pmatrix} u \\ v \end{pmatrix} = S^T \vec{t}$$

a zatim pomnožimo obje strane slijeva sa  $(S^T S)^{-1}$ :

$$\begin{pmatrix} u \\ v \end{pmatrix} = (S^T S)^{-1} S^T \vec{t}$$

Izračunati elemente matrice  $I_t$  nije teško, jer možemo samo uzeti razliku vrijednosti piksela narednog i trenutnog frejma, odnosno definisemo:

$$I_t(x, y, t) = I(x, y, t + 1) - I(x, y, t)$$

Prostorne izvode  $I_x, I_y$  bismo mogli izračunati najjednostavnije formulama:

$$I_x(x, y, t) = I(x + 1, y, t) - I(x, y, t)$$

$$I_y(x, y, t) = I(x, y + 1, t) - I(x, y, t)$$

Ovi izvodi odgovaraju korištenju kernela  $\begin{bmatrix} 0 & -1 & 1 \end{bmatrix}$ , odnosno njegove transponovane forme. Također možemo koristiti složenije kernele, u nadi da ćemo tako dobiti više informacija o prostornim parcijalnim izvodima. Jedan primjer bi mogao biti kernel:

$$\left[ \frac{1}{12} \quad \frac{-8}{12} \quad 0 \quad \frac{8}{12} \quad \frac{1}{12} \right]$$

za  $I_x$ , te njegova transponovana forma za  $I_y$ .

S obzirom da sada imamo matrice  $I_x, I_y, I_t$ , možemo jednostavno izračunati matricu  $S$  i vektor  $\vec{t}$ . Prije toga možemo izglađiti matrice  $I_x, I_y$  i  $I_t$  korištenjem kernela:

$$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Ostala nam je još samo jedna prepreka, a to je invertovanje matrice  $S^T S$ . Bolje rečeno, s obzirom da se radi o 2x2 matrici, samo invertovanje je moguće uraditi jednostavnom formulom:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Prije nego što možemo to uraditi, moramo provjeriti da li je matrica singularna, što ćemo uraditi tako što prvo izračunamo svojstvene vrijednosti matrice, te provjerimo da li su obje dovoljno velike. Svojstvene vrijednosti  $\lambda_1$  i  $\lambda_2$  matrice dimenzija 2x2 možemo dobiti kao rješenja jednačine:

$$\det \begin{bmatrix} a - \lambda_1 & b \\ c & d - \lambda_2 \end{bmatrix} = 0$$

Ako su i  $\lambda_1$  i  $\lambda_2$  ispod nekog praga  $\tau$ , matrica je singularna, te je optički tok nula-vektor. Ako je jedna od svojstvenih vrijednosti veća od  $\tau$ , matrica je loše uslovljena. Ako su obje svostvene vrijednosti iznad praga  $\tau$ , matrica nije singularna, te možemo izračunati optički tok.

Slično kao standardni algoritmi uparivanja blokova, Lucas-Kanade metoda ima problema u slučajevima kada postoji veliki broj potencijalnih kandidata. Zbog toga se ova metoda najčešće koristi za pronalaženje *rijetkog* optičkog toka, odnosno toka koji je poznat za samo neke piksele. Ako želimo koristiti ovu metodu za interpoliranje frejmova, moramo pronaći vektore pomaka za sve piksele korištenjem podataka koje dobijemo iz malog broja tačaka. Ako nas samo zanima rijetki optički tok, trebamo naći piksele čije će nam kretanje biti lako pratiti. U praksi se za tu svrhu koriste pikseli na uglovima, jer pomjeranje tih piksela i njihove okoline će uvijek biti lako uočljivo. Ako umjesto uglova koristimo npr. ivice, pomjeranje duž ivice nije uočljivo. Popularan i pouzdan algoritam za detekciju uglova jeste Shi-Tomasi algoritam.

### 3.4.3 Horn-Schunck metoda

Horn-Schunck je metoda koja direktno računa sve vektore pomaka tako što minimizira funkciju:

$$E = \iint (I_x u + I_y v + I_t)^2 dx dy + \alpha \iint \left\{ \frac{\partial u^2}{\partial x} + \frac{\partial u^2}{\partial y} + \frac{\partial v^2}{\partial x} + \frac{\partial v^2}{\partial y} \right\} dx dy$$

Dio funkcije pod prvim dvojnim integralom je dobiven direktno iz jednačine optičkog toka. Minimizacijom prvog dijela ćemo dobiti precizan optički tok, čiji su vektori pomaka slični stvarnim. Dio funkcije pod drugim integralom predstavlja mjerilo glatkoće optičkog toka. S obzirom da su  $u$  i  $v$  komponente vektora pomaka, sabirci pod drugim integralom nam daju indikaciju prostornih promjena u optičkom toku. Veće promjene znače i manje glatko polje optičkog toka, susjedni vektori pomaka se više razlikuju. Minimizacijom ovog drugog dijela funkcije upravo povećavamo glatkoću polja. Povećanjem faktora  $\alpha$  povećavamo značaj glatkoće optičkog toka, dok smanjivanjem povećavamo značaj preciznosti.

$I_x$ ,  $I_y$  i  $I_t$  bismo mogli izračunati korištenjem kernela koje smo naveli pri opisu Lucas-Kanade metode. Međutim, Horn-Schunck metoda koristi Sobel konvolucijski kernel za izračunavanje prostornih izvoda:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

i njegova transponovana forma.

Nakon izračunavanja  $I_x$ ,  $I_y$  i  $I_t$ , postavit ćemo sve vektore pomaka na nula vektore, te ih postepeno poboljšavati rekurzivnim formulama koje predstavljaju srž Horn-Schunck algoritma:

$$u_{x,y}^{k+1} = \bar{u}_{x,y}^k - \frac{I_x(I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2}$$

$$v_{x,y}^{k+1} = \bar{v}_{x,y}^k - \frac{I_y(I_x \bar{u}_{x,y}^k + I_y \bar{v}_{x,y}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2}$$

Pri tome,  $k$  u superscriptu označava trenutnu vrijednost, a  $k + 1$  sljedeću vrijednost komponenti vektora optičkog toka, dok  $\bar{u}_{x,y}^k$  i  $\bar{v}_{x,y}^k$  predstavljaju prosječne vrijednosti vektora pomaka za 4 piksela koji okružuju piksel na poziciji  $(x, y)$ . U svakoj narednoj iteraciji ćemo smanjiti vrijednost funkcije.

# Uklanjanje grešaka

Direktnom primjenom algoritama uparivanja blokova, moguće je dobiti interpolirane frejmove. Međutim, ti frejmovi će biti veoma slabe kvalitete. U ovom poglavlju će biti razrađene tehnike pomoću kojih ćemo riješiti sljedeća 3 problema:

1. Postojat će velike "rupe" u interpoliranom frejmu, jer za značajan dio piksela neće biti pronađen niti jedan blok koji ih pokriva. U prvom dijelu će biti objašnjena tehnika koju ćemo koristiti za njihovo popravljanje.
2. Bez obzira na to koji algoritam uparivanja koristimo, za neke blokove će biti generisani neispravni vektori pomaka. U drugom dijelu će biti objašnjene neke tehnike za detekciju i korekciju neispravnih vektora pomaka.
3. U zavisnosti od toga koji algoritam uparivanja koristimo, postoji niža ili viša vjerovatnoća da će granice između blokova u interpoliranom frejmu biti veoma oštare i očigledne. Ova činjenica opet proizlazi iz nesavršenosti algoritama uparivanja. Da bismo učinili interpolirane frejmove realističnijim i ugodnijim za gledati, u trećem dijelu će biti razrađene neke tehnike pomoću kojih je moguće umanjiti ovaj efekat.

## 4.1 Neispravni vektori pomaka

Korištenjem algoritama uparivanja blokova, dobili smo odgovarajuće vektore pomaka za svaki blok veličine  $16 \times 16$  piksela. Međutim, postoji visoka vjerovatnoća da neki od tih vektora ne odgovaraju stvarnom pokretu u video sekvenci. Prvo ćemo definisati šta zapravo znači neispravan vektor pomaka, kako ih detektovati, a zatim kako ih popraviti.

### 4.1.1 Detekcija neispravnih vektora pomaka

U ovom odjeljku ćemo objasniti generalnu metodu detekcije neispravnih vektora pomaka, pri čemu ćemo izračunavanje određenih konstanti ostaviti za poglavlje vezano za implementaciju (te konstante ćemo dobiti eksperimentalno).

Vektor pomaka za neki blok, bez obzira na korištenu metodu, je dobiven korištenjem *MAD* metode. Međutim, *MAD* nije efektivan u situacijama u kojima postoji veliki broj kandidata sa istom *MAD* mjerom. Da bismo preciznije izračunali ispravnost vektora pomaka, koristit ćemo još jednu mjeru, a to je *BAD* (*Boundary Absolute Difference*). Za blok  $B_1$  nekog frejma, i njemu upareni blok  $B_2$  iz narednog frejma, *BAD* računamo kao zbir apsolutnih vrijednosti razlika piksela na ivici bloka  $B_1$  i njima najbližih piksela koji okružuju blok  $B_2$ . Za blok veličine  $16 \times 16$  piksela, imat ćemo 64 ivična piksela. Radi konzistentnosti, možemo koristiti *SAD* umjesto *MAD*, odnosno *Sum Absolute Difference*, koji jednostavno predstavlja ukupni zbir apsolutnih vrijednosti razlika piksela, bez dijeljenja sa veličinom bloka. Nakon pronalaženja *SAD* i *BAD* mjera za odgovarajući blok, također trebamo odrediti konstante  $T_1$ ,  $T_2$ ,  $T_3$  i  $T_4$ , pri čemu je  $T_3 > T_1$  i  $T_4 > T_2$ . Ove konstante ćemo koristiti u algoritmu za određivanje kvalitete vektora pomaka, koji glasi:

- Ako je  $SAD < T_1$  ili  $BAD < T_2$ , vektor pomaka je ispravan
- Ako je  $SAD < T_3$  i  $BAD < T_4$ , vektor pomaka je ispravan
- U suprotnom, vektor pomaka je neispravan

Postoji još jedna metoda detekcije neispravnih vektora pomaka, koja se izvršava direktno na vektorima, umjesto na blokovima. Ovom metodom možemo otkriti vektore koji se značajno razlikuju od svojih susjednih vektora, što je najčešće indikator neispravnosti. Svaki vektor pomaka se sastoji od  $x$ -komponente i  $y$ -komponente, te ćemo definisati razliku dva vektora pomaka kao zbir apsolutnih vrijednosti razlika njihovih  $x$  i  $y$  komponenti, odnosno, razlika dva vektora pomaka  $mv_1$  i  $mv_2$  je  $|mv_{1x} - mv_{2x}| + |mv_{1y} - mv_{2y}|$ . Ako je za neki vektor pomaka zbir razlika sa njegovih 8 susjednih vektora veći od neke konstante, vektor označimo kao neispravan.

### 4.1.2 Ispravljanje neispravnih vektora pomaka

Neovisno od toga kako smo odredili neispravnost vektora pomaka, koristit ćemo istu metodu ispravljanja. S obzirom da očekujemo da će susjedni blokovi imati iste vektore pomaka, najlakši način da ispravimo neispravan vektor pomaka jeste da ga zamijenimo jednim od 8 susjednih vektora. Izbor ćemo izvršiti tako što ćemo, od 8 susjednih vektora, uzeti onaj sa minimalnim zbirom razlika od ostalih 7. Odnosno, uzet ćemo onaj vektor pomaka  $mv_k$  koji daje najmanji rezultat funkcije  $\sum_{i=1}^8 (|mv_{kx} - mv_{ix}| + |mv_{ky} = mv_{iy}|)$ , pri čemu su  $mv_1, mv_2, \dots, mv_8$  susjedni vektori pomaka našeg neispravnog vektora.

## 4.2 Izglađivanje granica između blokova

Korištenjem algoritama koji uparuju čitave blokove sa drugim blokovima, interpolirani frejmovi će imati veoma vidljive vertikalne i horizontalne linije između granica blokova. Korištenjem BAD metode možemo detektovati vektore pomaka koji rezultovati ovim artefaktima, ali to ne znači da ćemo ih moći uvijek kvalitetno ispraviti. Tako da je poželjno, radi poboljšanja kvalitete, promijeniti piksele u neposrednoj okolini tih artefakta radi njihovog maskiranja. Piksele možemo promijeniti korištenjem kernel konvolucije, koristeći kernel koji zamuće sliku u okolini artefakata. Koristit ćemo 1-D kernele za zamućivanje horizontalnih i vertikalnih linija koji su zasnovani na Gaussovoj funkciji. Vrijednosti matrice kernela možemo dobiti iz Gaussove funkcije na jednakim intervalima, pri čemu je također bitno da je kernel simetričan, što možemo dobiti ako uzimamo vrijednosti funkcije centrirane oko tačke 0. Matrice kernela za vertikalne i horizontalne linije će biti međusobno transponovane. U slučaju da se piksel nalazi u presjeku vertikalne i horizontalne linije, možemo koristiti kernel dobiven iz 2-D Gaussove funkcije. U oba slučaja, parametar  $\sigma$  predstavlja standardnu devijaciju. Povećanjem ovog parametra se povećava zamućenost slike, što smanjuje artefakte, ali se gubi više informacija o originalnoj slici.

1-D Gaussova funkcija:

$$G(x) = \frac{e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}}$$

2-D Gaussova funkcija:

$$G(x, y) = \frac{e^{-\frac{x^2+y^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}}$$

Primjer 1-D kernela (za  $\sigma = 1$ ):

$$\begin{bmatrix} 0.06136 & 0.24477 & 0.38774 & 0.24477 & 0.06136 \end{bmatrix}$$

Primjer 2-D kernela (za  $\sigma = 1$ ):

$$\begin{bmatrix} 0.003765 & 0.015019 & 0.023792 & 0.015019 & 0.003765 \\ 0.015019 & 0.059912 & 0.094907 & 0.059912 & 0.015019 \\ 0.023792 & 0.094907 & 0.150342 & 0.094907 & 0.023792 \\ 0.015019 & 0.059912 & 0.094907 & 0.059912 & 0.015019 \\ 0.003765 & 0.015019 & 0.023792 & 0.015019 & 0.003765 \end{bmatrix}$$

Da ne bismo dio slike učinili svjetlijim ili tamnijim, zbir svih vrijednosti matrice kernela mora biti jednak 1.

# Implementacija interpolatora korištenjem OpenCV biblioteke

Algoritmi opisani u ovom radu se mogu relativno jednostavno implementirati u bilo kojem modernom programskom jeziku. Zapravo, najveći problem predstavlja čitanje i pisanje video datoteka. Naime, za to nam je potreban način da pozovemo odgovarajući *codec* (eng. *coder-decoder*) za format video datoteke koji koristimo. OpenCV biblioteka nam omogućava upravo čitanje i pisanje video datoteka, te direktni pristup pikselima svakog frejma, što je nama od ključnog značaja za svaki opisani algoritam.

## 5.1 OpenCV

OpenCV (eng. *Open Computer Vision*) je biblioteka originalno razvijena od strane Intel-a, koja se fokusira na oblasti računarske vizije i mašinskog učenja. Originalna implementacija biblioteke je u C++ programskom jeziku, dok također postoji podrška za C, Java, Python i MATLAB. Funkcionalnosti biblioteke koje su nama potrebne uključuju čitanje i pisanje video datoteka sa punom kontrolom nad atributima (poput formata, rezolucije, framerate-a, itd.), direktna kontrola nad pikselima svakog pojedinačnog frejma, pronalaženje optičkog toka kroz ugrađene funkcije, te neke druge funkcionalnosti poput mogućnosti prikazivanja bilo kojeg frejma na ekranu u bilo koje vrijeme. Sve ovo predstavlja jedan veoma mali dio mogućnosti biblioteke, koja ima preko 2500 implementiranih i optimizovanih algoritama., zbog čega je korištena u mnogim od najvećih tehnoloških kompanija na svijetu.

Tri najvažnije OpenCV klase koje ćemo koristiti za implementaciju su `cv::Mat`, `cv::VideoCapture`, te `cv::VideoWriter`. Sve klase i funkcije OpenCV biblioteke se nalaze u `cv` imenskom prostoru. `cv::Mat` je višenamjenska klasa koju OpenCV koristi za držanje raznih tipova matrica, pri čemu

svaki element može imati jedan ili više različitih kanala. Očigledna korist ovoga jeste da možemo istu klasu korsititi za držanje frejmova u boji (gdje svaki piksel ima 3 kanala) i samo intenziteta svakog piksela. Na primjer, ako imamo varijablu `frejm` koja je tipa `cv::Mat`, da bismo pristupili vrijednosti piksela u boji na koordinatama  $(x, y)$ , koristimo sljedeći kod:

---

```
cv::Vec3b piksel = frejm.at<cv::Vec3b>(y, x);
```

---

Pri tome, `cv::Vec3b` je klasa koja drži tri bajta (jedan za svaki kanal), kojima možemo pristupiti korištenjem subscript operatora `piksel[0]`, `piksel[1]` i `piksel[2]`. I `.at` metoda i subscript operator vraćaju reference, tako da je isti pristup moguće koristiti i za postavljanje vrijednosti piksela, bilo čitavih ili pojedinačnih komponenti. Još treba napomenuti da subscript operatori ne referenciraju kanale piksela uobičajnim RGB (crvena-zelena-plava) redoslijedom, nego obrnutim (BGR). `cv::Mat` također ima `.rows` i `.cols` attribute, koji sadrže broj redova odnosno broj kolona frejma (drugim riječima, rezoluciju).

`cv::VideoCapture` i `cv::VideoWriter` klase koristimo za čitanje odnosno pisanje video datoteka. Pri tome, `cv::VideoCapture` može uzimati video podatke i iz drugih izvora, poput web kamera. U oba slučaja, `.open` metoda se koristi za otvaranje datoteke za čitanje odnosno pisanje (pri čemu u slučaju pisanja također navodimo potrebne informacije poput formata, rezolucije, i framerate-a video zapisa), tok se `.release` koristi za pravilno zatvaranje. `cv::VideoCapture` implementira `.read` metodu, ili ekvivalentni `>>` operator koji pročita, dekodira, i vrati naredni nepročitani frejm (vrijednost tipa `cv::Mat`). Da bismo znali da smo došli do kraja video zapisa, preporučeni način jeste poziv metode `.empty()` nad vraćenim frejmom, koja će vratiti logičku vrijednost `true` ako smo već pročitali posljednji frejm video zapisa, zbog čega će svaki drugi pokušaj čitanja rezultirati praznim frejmom.

Prateći istu logiku, `cv::VideoWriter` implementira `.write` metodu, umjesto koje možemo koristiti `<<` operator. Parametar u oba slučaja predstavlja vrijednost tipa `cv::Mat`, koja će biti dodana na kraj video zapisa. Pozivom prije spomenute metode, video zapis će biti zapisan u datoteku. Ako parametri proslijeđeni konstruktoru `cv::VideoWriter` objekta ne odgovaraju formatu frejmova koje pokušavamo dodati u video zapis, napravljena datoteka će biti veličine svega nekoliko kilabajta i neće sadržavati nikakve korisne informacije. Treba imati na umu da `cv::VideoWriter`, kroz ovaku "direktnu" upotrebu, neće ubaciti audio zapis u video datoteku. To je za

očekivati, jer nismo audio zapis nigdje ni naveli.

## 5.2 Implementacija

Osnovni tok interpolatora je sljedeći:

1. Inicijaliziraj objekte za čitanje i pisanje video datoteka
2. Učitaj prethodni frejm
3. Učitaj naredni frejm
4. Ako je naredni frejm prazan, završi sa radom
5. Inicijaliziraj objekat za pronalaženje optičkog toka, te pozovi metodu za izvršavanje
6. Pronađeni optički tok proslijedi metodi za generaciju interpoliranog frejma
7. Prodi kroz svaki piksel i na osnovu optičkog toka odredi poziciju u interpoliranom frejmu
8. Pozovi metodu koja popravlja "rupe" u frejmu
9. Popravi greške opisane u poglavljju 4
10. Novi frejm upiši u video zapis
11. Zamijeni prethodni i naredni frejm (tako da će prijašnji prethodni frejm biti prebrisani)
12. Vrati se na korak 3

Najviši nivo koda interpolatora se nalazi u `Interpolator::run` metodi:

---

```
void Interpolator::run()
{
    // Provjera ispravnosti opcija
    if (input_file_name.empty())
        throw std::logic_error("Input file name not present");
    if (output_file_name.empty())
```

```

        throw std::logic_error("Output file name not present");
if (interpolator_options.frames_to_generate < 1)
    throw std::domain_error("frames_to_generate has to be
                           at least 1");

// Ucitavanje i postavljanje informacija o video zapisu
video_capture.open(input_file_name);
video_info.fourcc =
    (int)video_capture.get(CV_CAP_PROP_FOURCC);
video_info.fps = (int)video_capture.get(CV_CAP_PROP_FPS);
video_info.height =
    (int)video_capture.get(CV_CAP_PROP_FRAME_HEIGHT);
video_info.width =
    (int)video_capture.get(CV_CAP_PROP_FRAME_WIDTH);
video_info.frame_count =
    (int)video_capture.get(CV_CAP_PROP_FRAME_COUNT);
known_pixel_map =
std::vector<std::vector<char>>(video_info.height,
    std::vector<char>(video_info.width, 0));

total_frames = video_info.frame_count;
frames_processed = 0;

// Otvaranje cv::VideoWriter objekta
video_writer.open(output_file_name,
                  CV_FOURCC('H', '2', '6', '4'),
                  video_info.fps *
                  (interpolator_options.frames_to_generate +
                   1),
                  cv::Size(video_info.width,
                           video_info.height));
if (!video_writer.isOpened())
    throw std::logic_error("Can't open video writer!");

// Glavna petlja
while (true)
{
    if (previous_frame.empty())

```

```

{
    video_capture >> previous_frame;
    video_capture >> next_frame;

    if (previous_frame.empty() || next_frame.empty())
        throw std::logic_error("Problem with video,
                               stopping");
}
else
{
    std::swap(previous_frame, next_frame);
    video_capture >> next_frame;

    if (next_frame.empty())
    {
        video_writer << previous_frame;
        return;
    }
}

// Poziv funkcije za generisanje novih frejmova
generate_intermediate_frames();
video_writer << previous_frame;

for (const auto& interpolated_frame :
     interpolated_frames)
    video_writer << interpolated_frame;
}
}

```

---

`CV_FOURCC('H', '2', '6', '4')` je macro za specifikaciju tzv. *FOURCC* tipa video formata. H.264 je veoma popularan format, tako da interpolator sve video zapise spašava tako. Naredni isječak koda se poziva ako želimo pronaći opetički tok Dijamantnom pretragom:

---

```

Optical_flow_field& Diamond_search::calculate()
{
    if (cost_map.empty())

```

```

        reset_cost_map();
if (opt_flow_field.data.empty())
    init_opt_flow_field();

for (int block_start_y = 0; block_start_y <
    prev_frame->rows; block_start_y += block_size)
{
    for (int block_start_x = 0; block_start_x <
        prev_frame->cols; block_start_x += block_size)
    {
        Vec2 block_opt_flow =
            calculate_block_opt_flow({block_start_x,
            block_start_y});

        for (int pixel_offset_y = 0; pixel_offset_y <
            block_size; pixel_offset_y++)
        {
            for (int pixel_offset_x = 0; pixel_offset_x <
                block_size; pixel_offset_x++)
            {
                int row = block_start_y + pixel_offset_y;
                int col = block_start_x + pixel_offset_x;

                if (row >= prev_frame->rows || col >=
                    prev_frame->cols)
                    continue;

                opt_flow_field.data[row][col] =
                    block_opt_flow;
            }
        }
    }
}

return opt_flow_field;
}

```

---

Vec2 je klasa koja samo sadrži dva cijelobrojna atributa. Vidimo upravo

**for** petlje koje prolaze kroz svaki blok u frejmu, te nakon izračunavanja optičkog toka, svaki piksel u bloku, dodjeljujući svakom izračunati vektor pomaka.

Linija koda u kojoj se izračunava krajnja pozicija piksela u interpoliranom frejmu:

---

```
Vec2 projected_position = Vec2(j, i) +
    (opt_flow_field.data[i][j] *
        (((double)frame_idx + 1) /
            (interpolator_options.frames_to_generate
            + 1)));
```

---

`Vec2(j, i)` je pozicija od koje krećemo, tj. pozicija piksela u prethodnom frejmu. `opt_flow_field.data[i][j]` je objekat tipa `Vec2` koji predstavlja vektor pomaka za trenutni piksel. Vektor pomaka pomnožimo sa faktorom koji zavisi od pozicije frejma između dva originalna. Npr. Ako između svaka dva frejma generišemo 3 dodatna, a trenutno generišemo prvog od njih, taj faktor će iznositi 0.25. Taj rezultat dobijemo ako u formulu uvrstimo vrijednosti `frame_idx = 0`, `interpolator_options.frames_to_generate = 3`.

Klasa `Interpolator` je glavna klasa korištena za implementaciju, te jedina čiju instancu moramo koristiti da bismo generisali novi video zapis. Klasa `Optical_flow_calculator` je apstraktna klasa koja je osnova za bilo koju klasu koju koristimo za izračunavanje optičkog toka. `Optical_flow_calculator` također sadrži dodatne pomoćne funkcije, poput `is_legal` koja provjerava da li su koordinate piksela validne, te `grayscale_pixel`, koja vraća intenzitet piksela prostom aritmetičkom sredinom njegovih komponenti. Glavna metoda je `calculate`, koja je čisto virtualna, te je trebaju implementirati izvedene klase. `calculate` vraća referencu na izračunati optički tok, što nam omogućava korištenje optičkog toka bez kopiranja, što bi usporilo izvršavanje. To također znači da ne smijemo dopustiti da objekat bude izbrisан dok imamo referencu na optički tok. Sam optički tok je sadržan u `Optical_flow_field` objektu, čiji atributi uključuju dimenzije polja optičkog toka (radi lakšeg pristupa), te same podatke sačuvane kao tip `std::vector<std::vector<Vec2>>`, odnosno jedan `Vec2` po pikselu. Razlog zašto metoda `calculate` vraća referencu a ne `const` referencu jeste što će optički tok biti kasnije izmijenjen u metodama za korekciju, korištenjem već obrađenih algoritama.

## 5.3 Qt

Qt framework, razvijen 1990-ih od strane dvojice Norveških programera, je jedan od najpopularnijih GUI (eng. *Graphical User Interface*) framework-a za C++ programske jezike (s tim da je Qt dostupan i za druge jezike). Omogućava jednostavan razvoj korisničkih sučelja, koristeći najpopularnije i najčešće korištene kontrole, poput dugmadi (`QPushButton`), unosa teksta (`QLineEdit`), indikatora napretka (`QProgressBar`), i mnogih drugih. Qt je izabran zbog svoje jednostavnosti, popularnosti, te činjenice da je napisan u C++-u. Ako koristimo Qt framework, najlakše nam je koristiti Qt-evo razvojno okruženje, Qt Creator, koje nam omogućava pravljenje korisničkih sučelja koristeći obični drag-drop pristup, te je integrisano sa Qt bibliotekama, što značajno olakšava razvoj. S obzirom da se Qt brine samo prezentacijskom logikom, koja nije od ključnog značaja za projekat, nećemo opisivati taj aspekt implementacije.

## 5.4 Rezultati interpolacije

Sada ćemo pogledati nekoliko frejmova jedne video sekvene preuzete sa web stranice youtube.com. Dimenzije korištenog video zapisa su 1280x720 piksela, a framerate iznosi 24 fps. Interpolator je koristio algoritam dijamantne pretrage za traženje optičkog toka, te je između svaka dva frejma generisan još jedan:

Figure 5.1: Prethodni frejm



Figure 5.2: Interpolirani frejm



Figure 5.3: Naredni frejm



Prvo što je uočljivo u interpoliranom frejmu jesu mnogobrojne horizontalne i vertikalne linije. To su artefakti koji su rezultat korištenja jednostavnog algoritma za interpolaciju. Iako su vektori pomaka prilično tačni, jednostavnim "lijepljenjem" blokova (koji su u ovom slučaju veličine 16x16 piksela) ćemo uvijek dobiti ovakve artefakte. Da bismo to popravili, morali bismo koristiti naprednije metode za interpolaciju ili značajno poboljšati algoritam izglađivanja ivica, koji treba minimizirati upravo ove artefakte. Na interpoliranom frejmu, sa desne strane papira, također možemo primijetiti da je interpolator ispravno zaključio da se ti blokovi kreću prema lijevo (jer se papir kreće prema lijevo), te su zbog toga uklonjeni neki pikseli bijelog papira i zamijenjeni pikselima pozadine. Algoritam je također ispravno zaključio da se većina blokova ne pomjera između dva frejma, zbog čega su artefakti lokalizovani na dijelove frejma koji se mijenjaju.

Još jedan primjer istog efekta možemo vidjeti na sljedeća tri frejma:

Figure 5.4: Prethodni frejm



Figure 5.5: Interpolirani frejm



Figure 5.6: Naredni frejm



Na narednom frejmu vidimo šta se desi ako pokušamo generisati novi frejm prije keyframe-a, odnosno kada prethodni i naredni frejm nisu slični:

Figure 5.7: Neuspjeli pokušaj interpolacije



# Zaključak

Interpolacija frejmova ima mnogobrojne realne primjene, od poboljšanja kvalitete video zapisa, do video kompresije. Algoritmi i metode korišteni za interpolaciju također imaju mnoge druge primjene. Optički tok, prividno kretanje objekata kroz video zapis je od ključne važnosti za mnoge oblasti računarske vizije, poput raznih vrsta robota, kamere koje detektuju, prate i mjere pokret, te prepoznavanja i praćenja raznih oblika. U svrhu interpolacije smo objasnili nekoliko algoritama, od kojih neki nisu uopšte zasnovani na izračunavanju optičkog toka, a drugi koriste složene matematičke metode za precizno i efikasno računanje. Neke metode, poput obične linearne interpolacije, su toliko brze da mogu biti korištene u realnom vremenu. Kao i u svim ostalim oblastima računarskih nauka, još uvijek se radi na traženju boljih, bržih i efikasnijih metoda.

Od metoda koje izračunaju optički tok, najjednostavnije su zasnovane na čistom uparivanju blokova piksela i traženju blokova koji se najbolje podudaraju. S obzirom da bi izračunavanje kvalitete uparivanja svaka dva para blokova piksela bilo neprihvatljivo sporo, pretraga je ograničena na relativno mali broj potencijalnih kandidata, i ubrzana heurističkim metodama koje se razlikuju od algoritma.

Naprednije metode uključuju uparivanje ubrzano korištenjem brze Fourierove transformacije, rješavanjem sistema linearnih jednačine, ili iterativnim poboljšanjem kvalitete optičkog toka. Nakon izračunavanja optičkog toka, ideja je uvjek ista: piksele pomjerimo jedan dio puta duž njihovog vektora pomaka, ostale piksele izračunamo drugim metodama (poput linearne interpolacije), te popravimo koliko možemo nastale artefakte. Kvalitet generisanih frejmova nikada neće biti isti kao kvalitet originalnih, ali korištenjem boljih i bržih metoda, približavamo se tom idealnom cilju.