Pronalaženje i praćenje prometnih znakova
Tehnička dokumentacija
Verzija 1.1

Studentski tim: Toni Benussi

Petra Bosilj

Mateja Čuljak

Darko Jurić

Bojan Miklenić

Martin Morava

Ante Trbojević

Lucija Zadrija

Nastavnici: Zoran Kalafatić

Siniša Šegvić

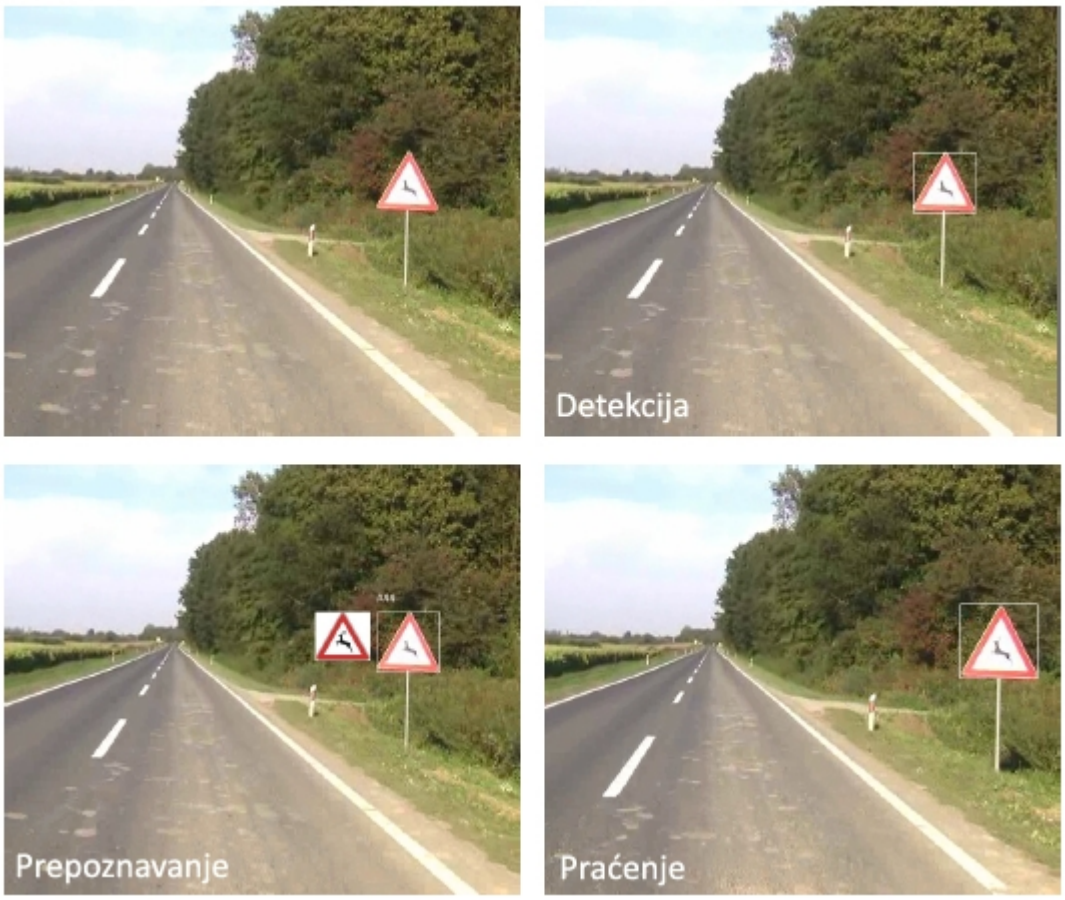
Sadržaj

Tehnička dokumentacija

1. Uvod

Prometni znakovi na cestama često nestaju, oštećuju se ili ih pak okolna vegetacija prekrije, što nikako nije poželjno. Kako državne i županijske službe imaju potrebu prometne znakove održavati u dobrom stanju, postupak održavanja prometnih znakova nastoji se automatizirati. U svrhu automatiyacije tog procesa koristi se specijaliziran softver i dodatna oprema koja je podrška korištenom softveru. Dodatnu opremu najčešće čine od vozilo s kalibriranom perspektivnom kamerom, diferencijalnim GPS prijamnikom, odometrom, žiroskopom i računalom. Takvo vozilo zajedno s kamerom i ostalim senzorima kreće se po prometnicama i pomoću kamere stvara video s pohranjenim podacima ostalih senzora. Nakon što je video stvoren, na računalu se prolazi kroz video i ručno se obavlja kartiranje i verifikacija. Kartiranje je postupak unošenja položaja prometnih znakova u kartu, a verifikacija je provjera jesu li svi znakovi na mjestu gdje bi trebali biti. Verifikacija se obavlja na osnovu dva videa, jednog u kojem su već kartirani prometni znakovi i drugog videa koji predstavlja tekuće stanje na terenu. Provjerom korespondentnih pozicija na obije video snimke utvrđuje se trenutno stanje prometnih znakova. Oba spomenuta procesa izvode se ručno, što je zahtjevno jer je potrebno po video zapisu tražiti prometne znakove, a kada se prometni znak primijeti, potrebno ga je označiti te mu odrediti vrstu i klasu.

Alternativa mukotrpnom ručnom izvođenju kartiranja i verificiranja je automatizacija tih procesa. Nakon što vozilo prikupi podatke, računalo će automatizirano, korištenjem specijaliziranog softvera, na stvorenom video zapisu obavljati detekciju, prepoznavanje i praćenje prometnih znakova. *Detekcija* prometnog znaka je određivanje pozicije prometog znaka na slici. *Prepoznavanje* se svodi na određivanje vrste i klase prometnog znaka, dok se *praćenje* prometnog znaka svodi na određivanje pomaka detektiranog prometnog znaka kroz nekoliko uzastopnih sličica (eng. *frame*) na videu. Proces detekcije, prepoznavanja i praćenja prikazan je na *Slici 1.1*.



Slika 1.1: Proces detekcija→prepoznavanje→praćenje

Jedan od alata pomoću kojeg je moguće obaviti označavanje i određivanja tipa znaka, dakle ručno ostvariti detekciju, prepoznavanje i praćenje, je program za označavanje MASTIF Marker, razvijen na zavodu ZEMRIS (dipl. ing. Karla Brkić). Marker ima tri osnovne funkcije:

- učitavanje videa ili niza statičnih slika,
- alate za ručno označavanje, brisanje oznake i odabir tipa prometnog znaka,
- spremanje podataka o oznakama u tekstualnu datoteku.

Cilj ovog projekta je omogućiti automatizaciju procesa detekcije i praćenja, a automatsko prepoznavanje znakova i izravna nadogradnjom Markera cilj je projekta *Integriranje dodanih mogućnosti u programski sustav Marker* [18].

Postoji nekoliko algoritama iz područja računalnog vida pomoću kojih je moguće automatizirati postupak detekcije i praćenja objekata. Najpoznatiji algoritam za detekciju je algoritam Viole i Jonesa, dok se za praćenje objekata koristi postupak Kanadea, Lucasa i Tomasija (KLT).

Oba spomenuta algoritma integrirana su unutar ljuske *cvsh2* (razvijeno za ZEMRIS-u, doc. dr. sc. Siniša Šegvić). Programska ljuska *cvsh2* sadrži skup algoritama i komponenti iz područja računalnog vida. Implementacija detekcije prometnih znakova koristi algoritam Viole i Jonesa [4], dok je postupak praćenja prometnih znakova implementiran već spomenutim algoritmom KLT.

Dva glavna nedostatka zatečene implementacije algoritma Viole i Jonesa unutar *cvsh2* ljuske ima dva glavna nedostatka: pojavu višestrukih odziva (jedan znak detektira se više puta) i presporo izvođenje. Implementacija algoritma KLT nema mogućnost praćenja više objekta.

Osim riješavanja ovih nedostataka, potrebno je rezultate obrade algoritama Viole i Jonesa i KLT preoblikovati u oblik koji će osigurati prenošenje rezultata vanjskim aplikacijama, koje pristupaju ljusci *cvsh2* preko dinamičke biblioteke *libmastif* [18].

U poglavlju 2 opisana su upute za instalaciju i pokretanje programskog okruženja u kojem je projekt razvijan. U poglavlju 3 ukratko se opisuje sučelje dinamičke biblioteke *libmastif* te se navode odgovarajuće promjene koje su izvršene nad implementacijama algoritama Viole i Jonesa i KLT u ljusci *cvsh2*. Poboljšanje algoritma detekcije, odnosno algoritma Viole i Jonesa razmatra se u poglavlju 4, dok se implementacija poboljšanog algoritma KLT opisuje u poglavlju 5. U posljednjem, 6. poglavlju razmatraju se još neke implementacije za detekciju kao što je OpenCV i detekcija prometnih znakova na osnovu boje [17], opisuje se postupak učenja detektora za trokutaste i okrugle prometne znakove i postupak evaluacije (ispitivanje uspješnosti i kvalitete detekcije).

2. Pregled korištene programske podrške

2.1 Pokretanje algoritama iz ljuske *cvsh2* u programskom okruženju Microsoft Visual Studio

2.1.1 Instalacija potrebnih komponenti i podešavanje postavki projekta

Programske aplikacije za prepoznavanje i praćenje prometnih znakova koje su, u okviru ovog projekta, optimizirane i prilagođene s ciljem da se mogu integrirati u konačni proizvod razvijane su u programskom okruženju Microsoft Visual Studio. Za testiranje ispravnosti rada i performansi implementacija algoritama Viole i Jonesa i KLT, programske implementacije integrirane su u ljusku *cvsh2* (Computer Vision Shell).

Ljuska *cvsh2* koristi određene komponente iz skupa biblioteka *boost* (skup biblioteka otvorenog koda široke primjene) koji se može preuzeti sa stranice *http://www.boost.org*. Potrebno je instalirati boost biblioteke (najbitnije su *system* i *filesystem* biblioteke) te unutar okruženja Visual Studio pod stavku *Project → Properties → C/C++ → General → Additional Include Directories* dodati putanju do direktorija u kojem se nalaze instalirane komponente.

` Također, radi uspješnog povezivanja (eng. *linking*) svih komponenti, kao dijela procesa prevođenja programske potpore, potrebno je nadopuniti stavku *Project → Properties → Linker → General → Additional Library Directiories* istom putanjom kao i u prethodnom koraku.

Kako bi se postigla kompatibilnost sa datotekama formata *.wmv* (*Windows Media Video*) potrebno je instalirati i *Windows Media Format SDK 11* (Software Development Kit) koji se može preuzeti sa stranice *http://msdn.microsoft.com/en-us/windows/bb190309.aspx*. Nakon toga te je potrebno ponoviti postupak dodavanja putanja do direktorija u kojima se nalazi *WMSDK 11* (postupak je isti kao i sa *boost* bibliotekama). Dodatno, pod stavku *Linker → Input* potrebno je dodati retke *„C:\WMSDK\WMFSDK11\lib\wmvcore.lib“ i „wfm32.lib“* radi kompatibilnosti sa starijom verzijom *.avi* formata.

Ako se koristi verzija okruženja Microsoft Visual Studio starija od verzije u kojoj je razvijen projekt (MS Visual Studio 2008), za otvaranje projekta potrebno je modificirati datoteku *vj.vcproj* u bilo kojem tekstualnom editoru (npr. Notepad) na način da se u polje Version upiše verzija Visual Studia koja se koristi.

2.1.2 Podešavanje parametara i rad sa programskim okruženjem

Ljuska *cvsh2* se poziva iz komandne linije (*Command Prompt* u operacijskom sustavu Windows) te služi kao unificirano sučelje za rad sa algoritmima iz područja računalnog vida, a ujedno dobavlja slike za korisnički kod, te iscrtava rezultate koje dobiva od korisničkog dijela koda.

Prilikom pokretanja aplikacije koja koristi ljusku *cvsh2*, potrebno je predati sljedeće parametre:

1. ulaznu datoteku (ili drugi izvor),
2. algoritam koji se želi primjeniti na dani ulaz (npr. Viola-Jones),
3. odredišnu datoteku sa rezultatima obrade ili je moguć tzv. interaktivni rad kojim se rezultati prikazuju prilikom obrade bez trajnog pohranjivanja,
4. parametre algoritma (ako ih algoritam zahtjeva).

Ulazna datoteka se postavlja tako da se prilikom poziva doda parametar *-sf=ime_datoteke* pri čemu *ime_datoteke* ime same datoteke zajedno sa potpunom putanjom do datotke. Ukoliko se žele obraditi sve datoteke unutar određenog direktorija, navodi se samo potpuna putanja direktorija i na kraj se dodaje znak „/“ (enl. *slash*). Dotična je mogućnost posebno pogodna pri radu sa slikama. Primjeri parametara putanje mogu se vidjeti u *Tablici 2.1*.

Tablica 2.1: Parametri putanje ulazne datoteke - primjeri

Primjer parametra	Objašnjenje parametra
`-sf="c:\video\prometni_znakovi.wvm"`	Obrađuje se samo datoteka *prometni_znakovi.wvm* u direktoriju *c:\video*
`-sf="c:\video/"`	Obrađuju se sve datoteke koje se nalaze u direktoriju *c:\video*

Odabir algoritma vrši se pomoću parametra *–a=ime_algoritma*. Ime svakog od algoritama odgovara imenu direktorija u kojem se nalazi njegova implementacija. Ukoliko se, na primjer, želi pozvati algoritam Viole i Jonesa potrebno je navesti sljedeći parametar prilikom poziva ljuske:

```
-a=vj
```

Rezultati obrade mogu se pohraniti u datoteku koristeći opciju *-d=ime_odredišne_datoteke*. Ova se opcija koristi pri uporabi algoritma u realnoj primjeni, tj. kada se podatci želi obraditi na što jednostavniji način bez potrebe da se upravlja procesom obrade datoteke korak po korak. Prilikom testiranja algoritma koristi se interaktivni način rada koji dopušta veću slobodu tako da se umjesto odredišne datoteke koristi opcija *-i=parametri*. Detaljan popis parametara sa objašjnenjima naveden je u *Tablici 2.2.*

Tablica 2.2: Parametri za interaktivni način rada

Parametar	Objašnjenje
`p`	obrada (od eng. *process*)
`s`	prikaz (od eng. *show*)
`a` *x*	adresa – redni broj sličice (eng. *frame*) videu
`n` *x*	obrađuje sljedećih *x* sličica
`help`	prikaz mogućih parametara
`quit`	izlaz iz aplikacije

Ove je parametre moguće predati odmah prilikom pokretanja programa ili prilikom samog rada, nakon obrade određenog broja sličica zadanog prilikom predhodnog upisa parametara. Primjeri korištenja ovih parametara s objašnjenjima mogu se vidjeti u *Tablici 2.3.* Primjeri u *Tablici 2.3.* odnose se na prvi poziv algoritma iz komandne linije, dok se prilikom interaktivnog načina rada parametri unose direktno i bez navodnika.

Tablica 2.3: Parametri za interaktivni način rada - primjeri

Primjer parametra	Objašnjenje
`-i="s a 4"`	prikaži četvrtu sličicu (eng. *frame*)
`-i="p n 10"`	obradi idućih deset sličica

Svaki od implementiranih algoritama ima vlastite parametre pa tako za algoritam Viole i Jonesa dodaje `-c=cascade`, gdje *cascade* označava kaskadu koja se koristi u ovoj implementaciji (moguće je izgraditi i koristiti i druge kaskade).

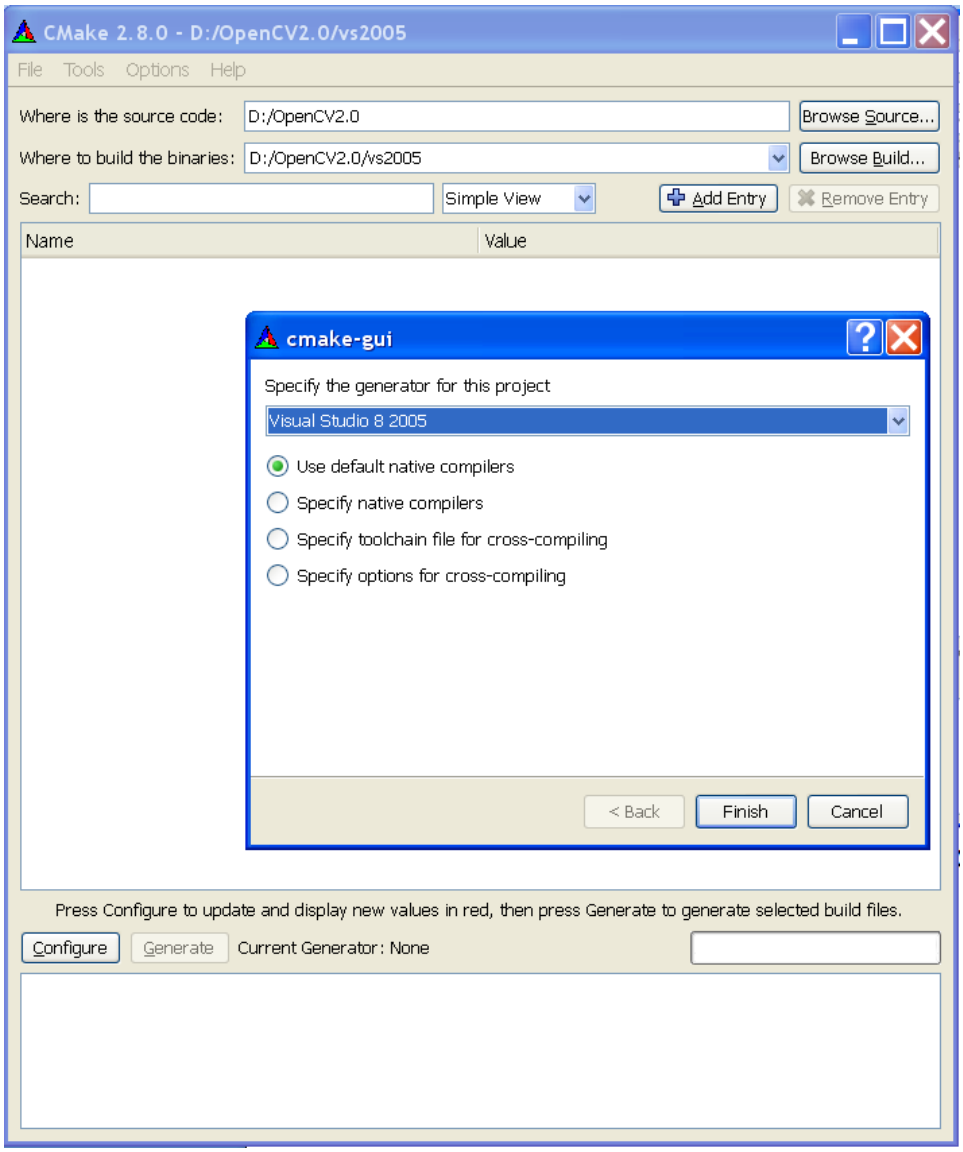
Za testiranje rada algoritma unutar programskog okruženja Microsoft Visual Studio, svi argumenti komandne linije unose se na za to predviđeno mjesto, pod stavku *Project → Properties → Debugging → Command Arguments.* Argumenti u interaktivnom načinu rada (ukoliko je pokrenut) upisuju se u prozoru komandne linije, koji otvara sam Visual Studio prilikom testiranja programa, na isti način kao da je algoritam pokrenut iz komandne linije.

2.2 Instalacija i podešavanje postavki biblioteke OpenCV

OpenCV [15] (Intel® Open Source Computer Vision Library) je biblioteka otvorenog koda (eng. o*pen source*) čiji je razvoj započeo *Intel*. Sadrži skup C funkcija i C++ razreda koje se često primjenjuju u obradi slika i algoritmima za računalni vid. Dostupna je na operacijskim sustavima Windows, Linux i MacOS X, izdaje se pod licencom *FreeBSD*. U sklopu ovog projekta korištena je verzija 2.0, koja dostupna je na Internetu i nalazi se na adresi: *http://sourceforge.net/projects/opencvlibrary/.*

Nakon preuzimanja izvornog koda s navedene adrese, potrebno binarne biblioteke. Izgradnja biblioteke može se opisati sljedećim koracija:

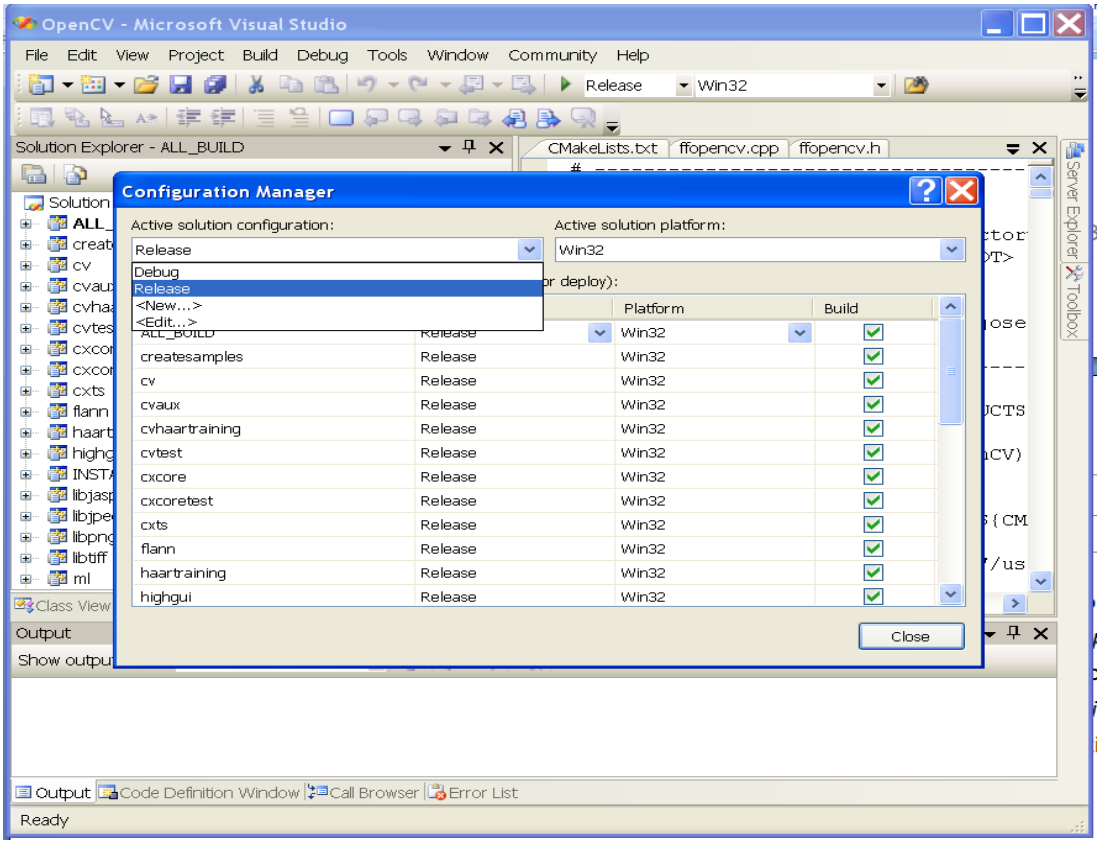
1. Potrebno je preuzeti i instalirati alat *CMake* koji je dostupan na adresi:*http://www.cmake.org/cmake/resources/software.html*.
2. Nakon instalacije, u novonastalom direktoriju *cmake-2.8.0-win32-x86* nalaze se direktoriji *bin*, *doc*, *man* i *share*. U direktoriju *bin* nalazi se izvršna datoteka *cmake-gui.exe,* potrebno ju je pokrenuti.
3. Pomoću ovog alata prevodi se *OpenCV* i to na slijedeći način.
   - Nakon pokretanja alata *Cmake* u novootvorenom prozoru u polje *Where is the source code:* potrebno je navesti direktorij u kojem će se nalaziti biblioteka *OpenCV.* U ovom slučaju kazalo je `C:\OpenCV2.0`.
   - U polje *Where to build the binaries* upisuje se kazalo na direktorij gdje će se nalaziti izgenerirane datoteke, npr.: `C:\OpenCV2.0\vs2005`, kako je prikazano na *Slici 2.1.*
   - Nakon toga, potrebno je pritisnuti tipku *Configure* i odabrati željeni razvojni alat
   - Podesiti ostale opcije prema želji
   - Ponovno pritisnuti tipku *Configure* te zatim tipku *Generate.*



Slika 2.1: Rad sa alatom Cmake

4. Ako se koristi Visual Studio ili bilo koji drugi razvojni alat, potrebno je otvoriti izgenerirani projekt, npr: `C:\OpenCV2.0\vs2005\OpenCV.sln` i prevesti u kunfiguraciji *Release* i *Debug*. To znači kiknuti redom na: *Build,* zatim *Configuration Manager*, pod *Active solution configuration* odabrati jednom *Debug,* drugi put *Release* te ostaviti u oba slučaja kvačicu na `ALL_BUILD`. Uz to, potrebno je uključiti *OpenMP* na projektima *cvhaartraining i haartraining* na

način na koji je opisano u odjeljku 4.2.1. Nakon uključivanja OpenMP-a, potrebno je prevesti samo projekte *cvhaartraining* i *haartraining,* desnim klikom miša na projekt i odabirom *Project Only* te npr. *Rebuild only cvhaartraining.*Nakon toga potrebno je pritisnuti F7 ili *Build – Build solution,* vrši se prevođenje cijele biblioteke OpenCV.



Slika 2.2: Podešavanje postavki prije prevođenja

U slučaju prevođenja iz komandne linije uz pomoć posebnih datoteka *Makefile*, potrebno je unjeti *path* do željenog direktorija i utipkati *make* ili *nmake...*

5. Potrebno je dodati novostvorene direktorije *Debug* i *Release* u *system path* na način da se otvori komandni prozor pritiskom na desnu tipku na direktorij *OpenCV2.0* i odabere *Open Command Window Here*. U komadni prozor se

upisuje *path*. U ovom slučaju u komandnu liniju upisujemo: *set* `PATH="C:\OpenCV2.0\vs2005\bin\Debug";"C:\OpenCV2.0\vs2005\bin\Release";%PATH%`.

3. Prilagodbe postojećih postupaka (nešto) biblioteka *libmastif*

3.1 Prenošenje točaka

Programskoj aplikaciji *Marker* potrebno je prenijeti rezultate detekcije zbog daljnje obrade. Dinamička biblioteka *libmastif*, sa sučeljem u C-u, omogućava spajanje vanjske aplikacije s algoritmima razvijenim unutar ljuske *cvsh2*.

Unutar dinamičke biblioteke *libmastif* nalazi se funkcija `EXPORT int CALLTYPE libMastif_dstPoints(LibMastifHandle handle, int index, LibMastifVectorDouble* pdata)` preko koje je moguće dohvatiti rezultate obrade. Prije poziva funkcije `libMastif_dstPoints` pozivatelj mora pozvati funkciju `EXPORT LibMastifHandle CALLTYPE libMastif_create(char const* arg)` pomoću koje se otvara veza s cvsh2 ljuskom. Preko argumenta `char const* arg` prenosi se identifikator algoritma koji će se pokrenuti:

- ako se poziva KLT algoritam, tada se u argument postavlja: „klt“
- ako se poziva algoritam Viole i Jonesa, tada se u argument postavlja: „vj“

Izlaz iz funkcije je cjelobrojni identifikator veći ili jednak nuli ili -1 ako zadani algoritam nije prepoznat. Putem funkcije `EXPORT int CALLTYPE libMastif_config(LibMastifHandle handle, char const* strConfig)` moguće je zadati konfiguracijski znakovni niz (`char const* strConfig`), pa tako ako se npr. poziva algoritam Viole i Jones, moguće je zadati putanju do kazala naučenog detektora. Preko argumenta `LibMastifHandle handle` prenosi se identifikator veze (izlaz funkcije `libMastif_create`).

Nakon stvaranje veze pozivatelj poziva funkciju `EXPORT int CALLTYPE libMastif_process` preko koje se prenosi slika koju je potrebno obraditi i započinje se izvođenjem iniciranog algoritma.

U svrhu dohvaćanja konačnih rezultata obrade koristi se već spomenuta funkcija `libMastif_dstPoints`, ali i funkcija `EXPORT int CALLTYPE libMastif_dstCount(LibMastifHandle handle)` koja dohvaća broj točaka vraćenih kao rezultat obrade.

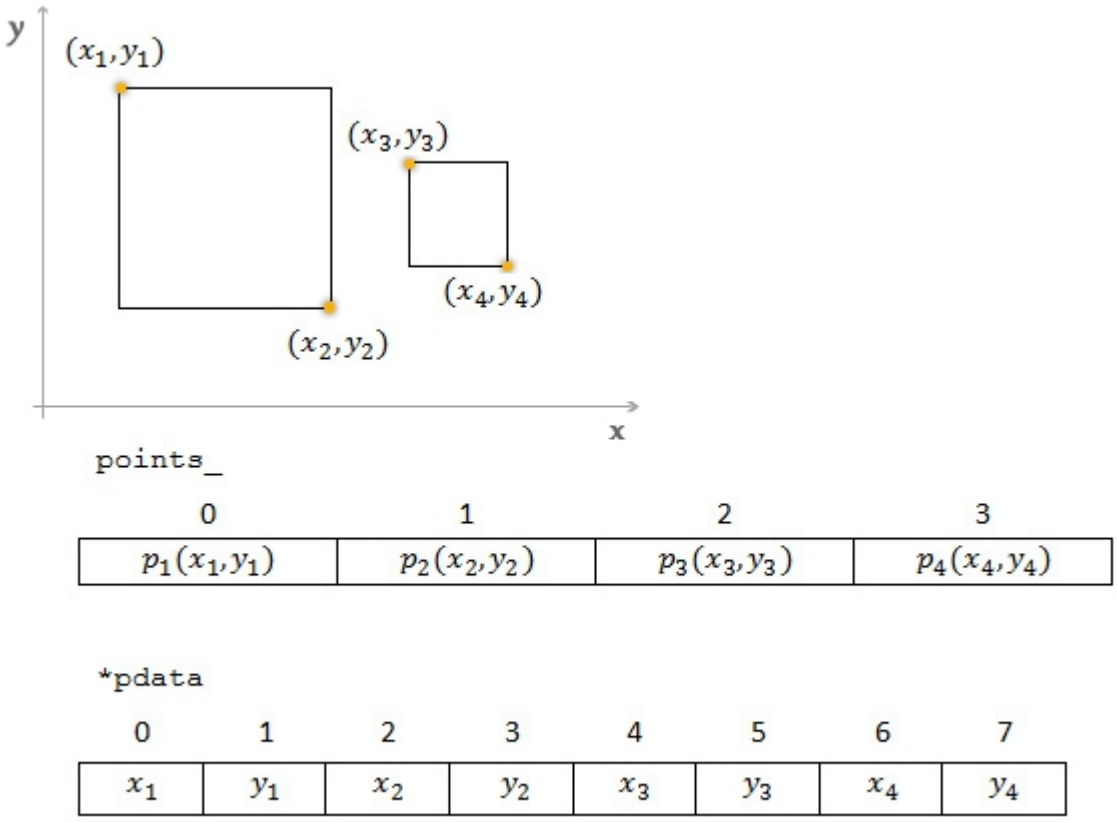
U funkciji `EXPORT int CALLTYPE libMastif_dstPoints(LibMastifHandle handle, int index, LibMastifVectorDouble* pdata)` argument `LibMastifHandle handle` je identifikator veze, dok se preko argumenta `int index` može odrediti iz koje izlazne slike dohvatiti rezultate. Algoritam Viole i Jones ima samo jednu izlaznu sliku pa je u tom slučaju taj argument uvijek nula, međutim algoritam KLT vraća 3 izlazne slike, pa je pomoću argumenta `int index` moguće odrediti iz koje izlazne slike je potrebno dohvatiti rezultate. Posljednji argument je pokazivač `pdata` tipa `LibMastifVectorDouble`, preko kojeg pozivatelj (vanjska aplikacija) treba prenijeti pokazivač na alociranu strukturu `LibMastifVectorDouble` unutar koje se nalazi pokazivač `double* buf` na alocirano polje, u kojem se očekuju rezultati dobiveni izvršavanjem odgovarajućeg algoritma unutar *cvsh2* ljuske. Sažeto rečeno, funkcija `libMastif_dstPoints` preuzima anotaciju iz tekućeg algoritma i iz te anotacije uzima x i y koordinate i kopira ih slijedno u polje na koje pokazuje pokazivač `double* buf`.

Kako implementacija detekcije iz završnog rada Tomislava Babića [4] koja je upakirana u ljusku *cvsh2* nije pohranjivala nikakve rezultate u vektor `points_`, bilo je potrebno implementirati dio koji će osigurati prenošenje točaka - rezultata obrade dinamičkoj biblioteci *libmastif.* Unutar datoteke alg_vj.cpp, odnosno u tijelu procedure `alg_vj::process`, nalazi se dio izvornog kôda koji rješava navedeni problem:

```
//gornja tocka
ann_.addPixel(grid::Pixel(rc[0],rc[1]),6, "", win_ann_abstract::PS_Dot);

//donja tocka
ann_.addPixel(grid::Pixel(rc[2],rc[3]),6, "", win_ann_abstract::PS_Dot);
```

Pritom se u vektor `points_` zapisuje prvo gornja-lijeva točka detekcije, a zatim donja-desna točka detekcije, znači u polje `*buf` prvo će se zapisati na *i*-tom indeksu x-koordinata gornje lijeve točke, na *i+1* y-koordinata gornje lijeve točke, zatim na mjestu s indeksom *i+2* x-koordinata donje-desne točke itd (*Slika 3.1*).



Slika 3.1: Prenošenje točaka

4. Poboljšanje algoritma detekcije

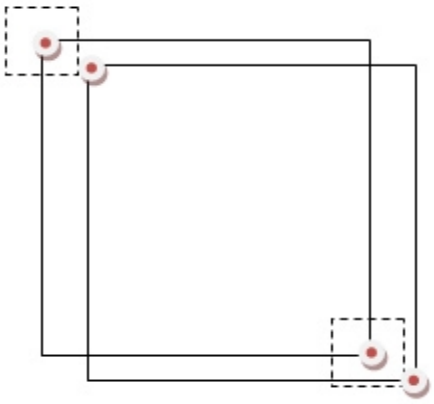
4.1 Grupiranje bliskih odziva u implementaciji algoritma Viole i Jonesa

Implementacija algoritma Viole i Jonesa gotovo uvijek vraća višestruke odzive detektiranog objekta (*Slika 4.1*), a ponekad se višestruki bliski odzivi pojavljuju i na pogrešnim detekcijama. Razlog pojavljivanja višestrukih bliskih odziva je posljednji detektor koji je osjetljiv i na male promjene u translaciji [2]. Kako je detektirane odzive potrebno prenijeti programskom alatu *Marker*, koji će dalje obrađivati detekcije (npr. prepoznavanje prometnog znaka), nema smisla prenositi višestruke odzive istog objekta, tj. prometnog znaka, jer će tada *Marker* vršiti višestruku obradu nad istim prometnim znakom, zbog čega dolazi do nepotrebne redundancije. Dakle prije prenošenja detekcija *Markeru*, trebalo bi odzive koji su dovoljno bliski grupirati u jedinstveni odziv. Postprocesiranjem i uvođenjem određenih heuristika može se postići zadovoljavajuće grupiranje bliskih odziva.

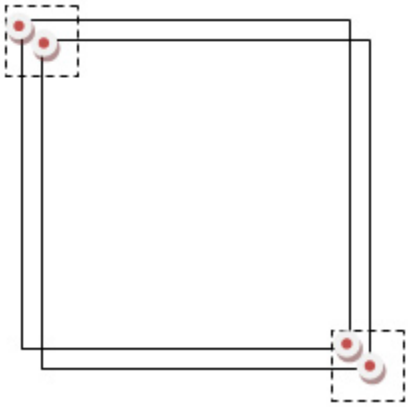
Slika 4.1: Primjer bliskih odziva

Osnovna ideja grupiranja je bliske odzive svrstati u zajednički skup i zatim izračunati jedinstveni odziv koji najbolje reprezentira odzive unutar skupa [2]. Taj reprezentativni odziv računa se aritmetičkom sredinom svih odziva koji se nalaze u skupu. Odziv se smatra bliskim odzivom u odnosu na susjedni odziv samo ako postoji preklapanje između ta dva odziva. Dva odziva se preklapaju ako se koeficijent translacije između ta dva odziva nalazi unutar zadanih granica. Na taj će način svi odzivi koji se međusobno preklapaju biti svrstani u isti skup.

Dva odziva se preklapaju ako se točke površinom manjeg odziva (gornja-lijeva i donja-desna točka) nalaze unutar zamišljenih pravokutnika koji omeđuju gornju-lijevu i donju-desnu točku površinom većeg odziva, kao što se može vidjeti na *Slici 4.2* i *Slici 4.3*.



Slika 4.2: Nema preklapanja



Slika 4.3: Postoji preklapanje

Postupak određivanja preklapanja pravokutnika možemo opisati na sljedeći način: neka je prvi odziv d_1, a drugi odziv d_2.

Neka je d_{veci} oznaka površinom većeg odziva tj.

$$d_{veci} = veci(d_1, d_2),$$

i neka je d_{manji} oznaka za drugi, manji odziv.

$$d_{manji} = manji(d_1, d_2).$$

Neka je Z zamišljeni pravokutni, i neka su Z_{sirina} i Z_{visina}, redom širina i visina zamišljenog pravokutnika.

Dimenzije zamišljenog pravokutnika računaju se prema formulama:

$$Z_{sirina} = tolerationOverlapping \cdot d_{veci\,sirina}$$

$$Z_{visina} = tolerationOverlapping \cdot d_{veci\,visina},$$

pri čemu je $tolerationOverlapping$ konstanta pomoću koje je moguće utjecati na osjetljivost grupiranja, odnosno dimenzije dimenzije zamišljenog pravokutnika iznosit će (tolerationOverlapping*100)% iznosa dimenzija većeg odziva

Za potrebe opisa algoritma (određivanja preklapanja) poslužit će dva zamišljena (identična) pravokutnika Z_G i Z_D koji imaju dimenzije izračunate na gore opisan način.

Zamišljeni pravokutni Z_G postavlja se tako da točka sjecišta dijagonala zamišljenog pravokutnika Z_G, bude gornja lijeva točka većeg odziva d_{veci}.

Zamišljeni pravokutni Z_D postavlja se tako da točka sjecišta dijagonala zamišljenog pravokutnika Z_D, bude donja desna točka većeg odziva d_{veci}.

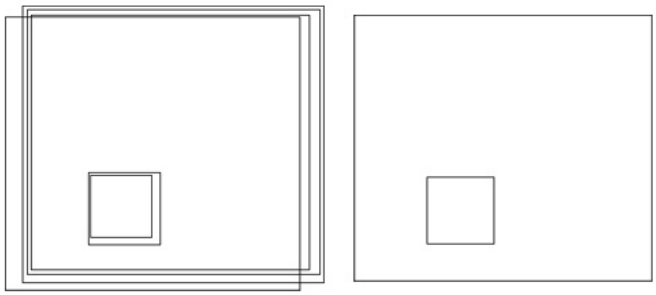
Ako se gornja desna točka manjeg odziva d_{manji} nalazi unutar zamišljenog pravokutnika Z_G i ako se donja desna točka manjeg odziva d_{manji} nalazi unutar zamišljenog pravokutnika Z_D tada se odzivi d_1 i d_2 preklapaju (*Slika 4.3*).

Ako se dva odziva preklapaju tada se oni stave u isti skup. Nakon obrade svih odziva, odnosno nakon utvrđivanja preklapanja preostalih odziva, oni odzivi koji se nalaze u istom skupu grupiraju se u jedan odziv.

U implementaciji upravo funkcija `bool Filter::overlapping(Element r1, Element r2)` ispituje da li se dva odziva preklapaju. `Element r1` predstavlja jedan odziv, a `Element r2` drugi odziv.

Preko varijable `float tolerationOverlapping` moguće je utjecati na osjetljivost grupiranja.

Nakon grupiranja bliskih odziva moguća je situacija prikazana na *Slici 4.4*. koja prikazuje rezultat grupiranja. Iz slike se vidi kako su grupiranjem početnih 6 odziva,



Slika 4.4: Grupiranje bliskih odziva

proizašla 2 grupirana odziva. Površinom veći odziv nastao je grupiranjem 4 odziva, a manji grupiranjem 2 odziva. Manji odziv se cijelom površinom nalazi unutar većeg odziva. Dva manja odziva grupirana su u zasebni odziv jer ne zadovoljavaju uvjete preklapanja s većim odzivima. Ipak, bilo bi smisleno odbaciti manji grupirani odziv u potpunosti jer se on sastoji od samo 2 odziva, što je dvostruko manje od većeg odziva, pa je za očekivati da manji odziv ne predstavlja kvalitetnu detekciju prometnog znaka.

Zbog ovakvih situacija, nakon grupiranja bliskih odziva, još se jedanput prolazi kroz skup grupiranih odziva i uzimaju se dva po dva grupirana odziva.

Neka je prvi uzeti grupirani odziv D_1, a drugi D_2. Neka je N_1 broj odziva od kojih je sačinjen D_1, a N_2 broj odziva od kojih je sačinjen D_2 i neka su P_1 i P_2 odgovarajuće površine grupiranih odziva D_1 i D_2 respektivno.

Ako je $N_2 << N_1$, odnosno ako vrijedi

$$\frac{N_2}{N_1} \leq \text{tolerationBtwNeighbors}$$

i ako postoji presjek (s površinom I) između D_1 i D_2

i ako je $I \approx P_2$, odnosno ako vrijedi

$$\frac{I}{P_2} > \text{tolerationIntersection}$$

tada se odziv D_2 odbacuje.

Izvorni kôd kojim je implementirano grupiranje, a čiji se rezultat izvođenja može vidjeti na *Slici 4.5,* nalazi se u datoteci *grupiranje.cpp*.

Slika 4.5: Primjer grupiranja bliskih odziva

Unutar biblioteke OpenCV, implementiran je algoritam za detektiranje objekata algoritmom Viole i Jonesa koji također vrši grupiranje bliskih odziva postprocesiranjem [1]. U toj implementaciji algoritama, postoji koeficijent *min_neighbours*, koji služi za odbacivanje grupiranog odziva koji se sastoji od *N* odziva, pri čemu se odbacivanje primjenjuje ako je zadovoljeno *N* < *min_neighbours*. Time se mogu eliminirati pogrešne detekcije, međutim dakako moguće je i odbacivanje točnih (uspješnih) detekcija, pa ta metoda nije implementirana u projektu.

4.2 Ubrzanje postojeće implementacije algoritma Viola-Jones

Implementaciji detekcije prometnih znakova iz završnog rada Tomislava Babića [4] potrebno je prosječno 8,02 sekundi za obradu jedne slike. Ovi rezultati (kao i svi ostali rezultati testiranja) dobiveni su testiranjem na uzorku od 100 slika dimenzija 720x576, na procesoru Intel® Core™2 Duo 2,4 GHz. Takva duga obrada je nepraktična i predstavlja problem pri korištenju implementacije. Veliko se ubrzanje dobilo korištenjem opcije prevodioca /Ox (potpuna optimizacija), nakon čega je bilo potrebno prosječno 2,25 sekundi po slici. Za daljnje povećanje brzine postojeće implementacije, razmatrane su i korištene tehnologije OpenMP, pisanje dijelova programskog koda u strojnom kodu koristeći instrukcijski skup x86 te njegova ekstenzija SSE (*Streaming SIMD Extension*) [9].

Prije pristupa samom ubrzanju implementacije izvršeno je profiliranje izvornog koda pomoću alata Intel® VTune™ Performance Analyzer. Profiliranjem je ustanovljeno da najviše vremena, čak 96%, uzima funkcija `Cascade::runCascade(int*, int*, int, int, int, int, double)` pa se shodno tome pri modificiranju programskog koda potrebno usredotočiti na funkciju `Cascade::runCascade` (datoteka `ext_vj_cascade.cpp`).

4.2.1 OpenMP

Tehnika paralelnog programiranja je jedna od metoda kojom bi se moglo postići ubrzanje. Kako većina današnji računala sadrži barem 2 jezgre, paralelizacijom izvornog koda, tj. uvođenjem dretvi gdje je to moguće, pogotovo u segmentima koji troše najviše vremena u obradi, može se očekivati značajnije ubrzanje.

Uvidom u izvorni kod funkcije `Cascade::runCascade` koja se razmatra za ubrzanje primjećuju se tri ugnježdene `for` petlje, gdje se gotovo sav koristan posao obavlja u zadnjoj (razinski najnižoj) `for` petlji. Funkcija `Cascade::runCascade` poziva se jedino unutar procedure `ext_vj` (datoteka `ext_vj.cpp`), unutar koje se također nalaze tri ugnježdene petlje, a funkcija `Cascade::runCascade` se poziva unutar bloka zadnje petlje što u konačnici daje 6 ugnježdenih programskih petlji. Iz toga proizlazi da je za efikasno

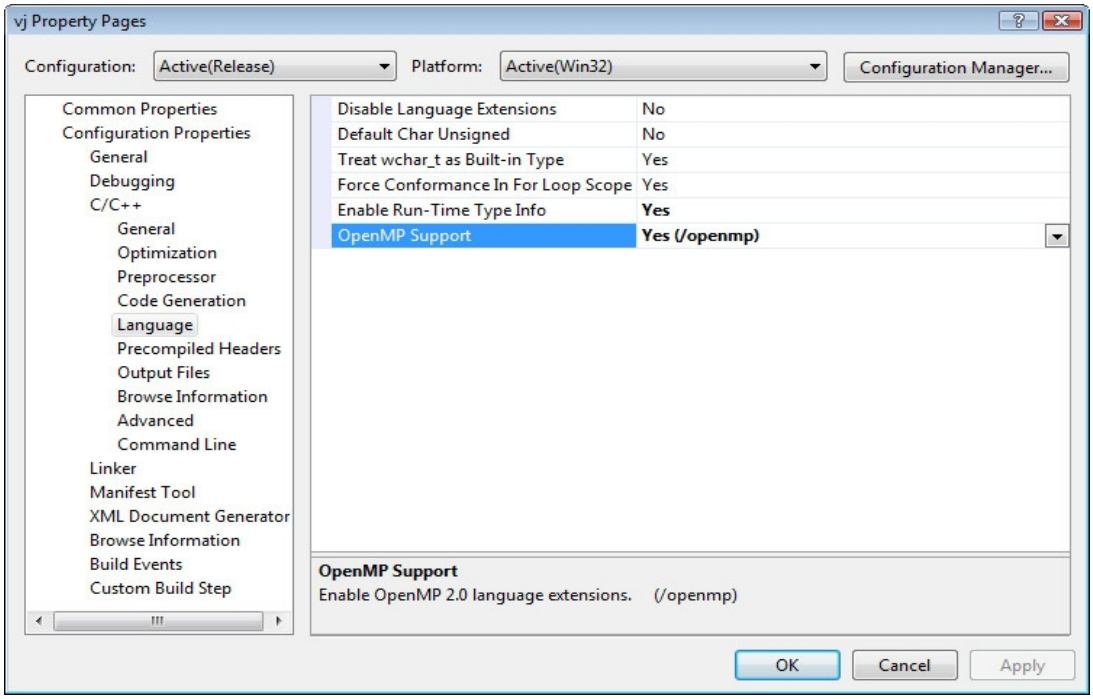
postizanje ubrzanja potrebno paralelizirati neku od tih 6 petlji.

Paralelizacija je implementirana korištenjem programskog sučelja OpenMP (eng. *Open Multi-Processing*) koje znatno olakšava paralelno programiranje. OpenMP podržava paralelno programiranje u jezicima C/C++ te Fortran i na platformama Unix i Microsoft Windows. Blokove izvornog koda moguće je paralelizirati korištenjem direktiva prevoditelja pomoću kojih je moguće utjecati na ponašanje izvođenja programa. Korištenjem OpenMP direktiva moguće je jednostavno paralelizirati petlje i nezavisne blokove izvornog koda [3].

Razvojna okolina Visual Studio također podržava OpenMP, konkretno Microsoft Visual Studio 2008 podržava OpenMP 2.0. Međutim, da bi se mogle koristiti OpenMP direktive u unutar C++ izvornog koda, potrebno je uključiti podršku za OpenMP u postavkama prevoditelja.

Unutar Visual Studia 2008 to se postiže:

1. postavljanjem stavke *Project* → *Properties* → *C/C++* → *Language* → *OpenMP Support* na „Yes“, kao što je prikazano na Slici 4.6



Slika 4.6: Uključivanje OpenMP-a u Visual Studiu

2. u izvornom kodu implementacije komponente (u *.cpp* datoteci) koja koristi OpenMP direktivne navesti `#include <omp.h>`.(*Slika 4.7*)

```
#include "ext_vj.hpp"
#include "grupiranje.hpp"

#include <math.h>
#include <string.h>
#include <omp.h>

#define GROUPING
//local functions
namespace{

  void rgbToGrey (int width,
  {
```

Slika 4.7: #include <omp.h>

U programskom jeziku C/C++, OpenMP direktive su izvedene pragmama. Za prevođenje direktive zadužen je C/C++ pretprocesor. Sve OpenMP direktive za C/C++ imaju sintaksu **#pragma omp <akcija>** i pojavljuju se neposredno prije bloka izvornog koda za koji je potrebno primijeniti zadanu akciju (direktivu) [8].

Za paralelizaciju `for` petlji direktiva glasi `#pragma omp parallel for`. Ako želimo paralelizirati blok naredbi tada se takav blok omeđuje vitičastim zagradama i na početak bloka stavlja se `#pragma omp parallel` (blok izvode sve raspoložive dretve), a unutar bloka pomoću direktive `#pragma omp section` označuje se izvorni kôd koji će izvršiti samo jedna dretva. Primjeri paralelizacije for petlje i bloka naredbimogu se vidjeti u *Tablici 4.1*.

Tablica 4.1: Primjeri paralelizacije

Paralelizacija for petlje	Dvije dretve paralelno će izvršiti izvorni kôd

`#pragma omp parallel for`
`for(int i=0; i<10; i++)`
`{`
`...`
`}`
	`#pragma omp parallel`
	`{`
	`#pragma omp sections`
	`{`
	`#pragma omp section`
	`// prvi blok naredbi`
	`#pragma omp section`
	`// drugi blok naredbi`
	`}`
	`}`

Valja napomenuti kako blok ili petlja koja se paralelizira ne smije sadržavati naredbe kojima se „skače“ izvan tog bloka, odnosno blok ne smije sadržavati naredbe: `break`, `exit`, `return` ili `goto` s labelom koja se nalazi izvan bloka. Za ispravnu paralelizaciju, broj iteracija for petlje mora biti izračunljiv pri nailasku na petlju za vrijeme izvođenja. Petlje koje u uvjetnom izrazu sadrže varijable ne-cijelobrojnog tipa podataka kao što su `float` ili `double` također se ne mogu paralelizirati.

Primjeri `for` petlji koje nije moguće paralelizirati mogu se vidjeti u *Tablici 4.2.* Jednoj od tih petlji nije moguće izračunati broj izvođenja petlje pri nailasku na izraz `for(int i = 1; i > 0 ; i++)` prilikom prevođenja izvršnog kôda, dok se u uvijetu druge pojavljuje necjelobrojni podatak `a[i]` tipa `float`.

Tablica 4.2: For petlje koje se ne mogu paralelizirati

```
int a[10] = {2};
#pragma omp parallel for
for(int i = 1; i > 0 ; i++)
{
      a[i] = a[i] * pow(2,i);
      if ( i >= 9 )i = -1;
}
```

```
float a[10] = {2.0};
#pragma omp parallel for
for(float i = 1.0; a[i]<16.0 ; i+=0.2)
{
      a[i] = a[i] * pow(2,i);
      if ( i >= 9 ) i = 1.0;
}
```

Nepažljivom paralelizacijom pomoću OpenMP direktiva moguće je prouzročiti neispravno izvođenje programa, kao u prvom primjeru u *Tablici 4.3.*

Tablica 4.3: Primjer neispravne i ispravne paralelizacije

```
int a[5]={1};
#pragma omp parallel for
for(int i = 1; i < 5 ; i++)
{
      a[i] = a[i-1] * 2;
}
```

```
int a[5]={1};
#pragma omp parallel for
for(int i = 1; i < 5 ; i++)
{
      a[i] = (int) pow(2.0, (double) i);
}
```

Izvođenje ovog primjera ne daje očekivane rezultate, jer svaka iteracija zavisi o predhodnoj iteraciji [7]. Pošto je zadana paralelizacija `for` petlje, svaku će iteraciju petlje izvoditi zasebna, neovisna dretva. Ako su na raspolaganju 4 dretve, svaka od tih dretvi dobit će jednu od 4 iteracija zadane `for` petlje. Dretvi koja izvodi *i*-tu iteraciju potreba je vrijednost *a[i-1]* koja se izračunava u *(i-1)* iteraciji, no kako se dretve izvode paralelno, nije moguće znati da li je dretva odgovorna za *(i-1)* iteraciju izračunala svoju vrijednost za *a[i-1]* i pohranila vrijednost u dijeljenu memoriju u trenutku kada *i*-ta dretva

zatraži vrijednost *a[i-1]*. Ovaj je primjer ipak moguće paralelizirat na način da se dobije ispravan rezultat (drugi primjer u *Tablici 4.3*), što se postiže promjenom algoritma. Isplativost postizanja paralelizacije promjenom algoritma u ovom slučaju je upitna, jer se preoblikovani algoritam koristi funkcija `pow` čije je izvođenje vremenski zahtjevno, no primjer koristi kao ilustracija preoblikovanja algoritma koji se ne može ispravno paralelizirati u paralelno izvodivi algoritam koji daje ispravno rješenje.

4.2.2 Paralelizacija izvornog koda iz završnog rada Tomislava Babića

Ubrzanje je postignuto paralelizacijom druge ugnježdene petlje u funkciji `ext_vj`. Originalni izvorni kôd prije unošenja OpenMP direktiva:

```
double x = 0.0, y = 0.0;
for (y = 0; y<(height - cascade.height-0.5); y=y+step2)
{
        i = myround(y);
        for (x = 0; x<(float)(width-cascade.width-0.5); x=x+step2)
        {
              int j = myround(x);
               result = cascade.runCascade(imgInt, imgSqInt, j, i, width,
height, factor);
              if (result == 1)
              {

                    detections.push_back(ext_vj_rect(4));

                    detections.back()[0]=j;
                    detections.back()[1]=height-i;
                    detections.back()[2]=j+cascade.width;
                    detections.back()[3]=height-i-cascade.height;
                  #ifdef GROUPING
                    filt.addDetection(detections.back());
                  #endif
              }
        }
  }
```

Kako u rješenju, u funkcijama `ext_vj` i `runCascade`, nema međusobno zavisnih petlji, pristup paralelizaciji je relativno jednostavan jer ne uključuje veće izmjene u izvornom kôdu.

Originalni izvorni kôd ipak je bilo potrebno izmijeniti zato što se u izrazima `for` petlje koriste varijable s posmičnim zarezom. Varijable `y`, `x`, `step2` su tipa `double`, a

OpenMP ne podržava varijable s posmičnim zarezom. U problematične `for` petlje uvedene su nove varijable cjelobrojnog tipa podataka i izvršena su odgovarajuća preoblikovanja izvornog koda, čime se osigurava ispravno izvođenje programa. Promjenjeni programski kod glasi kako slijedi:

```
int x, y;
int cascHeight = cascade.height;
int cascWidth = cascade.width;
int myStep = (int) myround(step2);
double x2=0.0;
double y2=0.0;

int yL = (int)((double)(height-cascHeight)/step2);
int xL = (int)((double)(width-cascWidth)/step2);

#pragma omp parallel for num_threads(4) firstprivate
(x2,width,height,factor,cascHeight, cascWidth, imgInt, imgSqInt)
private(i,x,result) schedule(dynamic)
for (y = 0; y<yL; y++)
{
        i = myround(y2);
        x2=0.0;
        for (x = 0; x<xL; x++)
        {
              int j = myround(x2);
              result = cascade.runCascade(imgInt, imgSqInt, j, i, width,
                                          height, factor);
              if (result == 1)
              {

                  #pragma omp critical
                  {
                    detections.push_back(ext_vj_rect(4));
                  }
                    detections.back()[0]=j;
                    detections.back()[1]=height-i;
                    detections.back()[2]=j+cascWidth;
                    detections.back()[3]=height-i-cascHeight;

                  #ifdef GROUPING
                    filt.addDetection(detections.back());
                  #endif
              }
              x2 = x2+step2;
        }
        y2 = y2+step2;
  }
```

Unutar OpenMP direktive: `#pragma omp parallel for num_threads(4) firstprivate (x2,width,height,factor,cascHeight, cascWidth, imgInt, imgSqInt) private(i,x,result) schedule(dynamic)`, klauzula `num_threads(4)` govori prevoditelju

s koliko dretvi izvršiti paralelizaciju. Svaka dretva će izvršiti jednu iteraciju for petlje, a po završetku iteracije dinamički (`schedule(dynamic)`) će se dodijeliti nova iteracija. Ovo svojstvo doprinosi dodatnom ubrzanju, jer sve iteracije neće imati jednaku količinu posla. Da je klauzula `schedule` napisana `static`, tada bi se prije početka izvođenja petlje unaprijed odredilo koju iteraciju će izvršiti koja dretva [8].

Varijable koje se nalaze unutar klauzule `private` bit će kopirane s onoliko kopija kolika ima dretvi, a svaka će kopija biti dodijeljena zasebnoj dretvi. Takve varijable nazivaju se privatnim varijablama i dohvatljive su samo unutar djelokruga dodijeljenog određenoj dretvi [8]. Uvođenjem privatnih varijabli osigurava se ispravno izvođenje rješenja, jer ako bi varijabla bila dijeljena, tada bi dretva mogla povući vrijednost koju je zapisala prijašnja dretva što u nekim slučajevima to nije poželjno. Valja napomenuti kako privatne varijable nisu inicijalizirane pa bi prije samog čitanja trebalo dodijeliti vrijednost varijabli.

Atributna klauzula `firstprivate` ima slična svojstva kao `private` klauzula - razlika je u tome što se kopirane varijable inicijaliziraju ovisno o vrijednostima koje su imale originalne varijable prije ulaska u blok koji se paralelizira.

Naredba `push_back(ext_vj_rect(4))` omeđena je direktivnom *CRITICAL* direktivom, čime se sprječava da više dretvi izvodi tu naredbu u isto vrijeme. Direktiva

$$n_i - ukupan\ broj\ iteracija\ i - te\ petlje$$

$$num\_threads - broj\ dretvi$$

Slika 4.9: Legenda za Tablicu 5.4

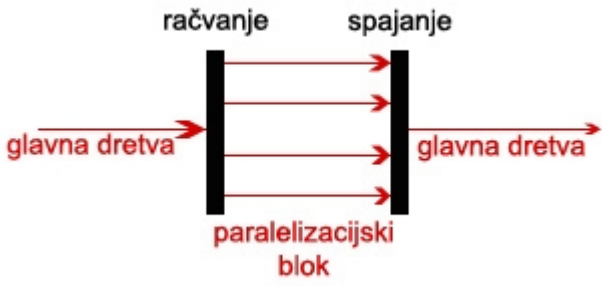
```
#pragma omp critical
{
      detections.push_back(ext_vj_rect(4));
}
```

CRITICAL u ovom slučaju potreba je zbog dinamičke alokacije memorije. Kako je vektor `detections` dijeljeni vektor, istovremena dinamička alokacija memorije za novi element u vektoru od dvije ili više dretvi narušila bi strukturu vektora.

Korištenjem tehnologije OpenMP implementacija je ubrzana za otprilike 87% (ubrzanje sa 2,25 na 1,2 sekundi po slici, uz optimizaciju prevodioca). Kako je procesor

korišten za testiranje dvojezgreni, ovo je ubrzanje iznimno dobro (idealno ubrzanje koje se može postići višedretvenošću na dvojezgrenom procesoru je 100%), čemu je vjerojatno pridonjela tehnologija Intel® Hyper-Threading koja donosi dodatno ubrzanje.

Valja istaknuti kako paralelizacijom zadnje (šeste) ugnježdene petlje umjesto druge ne bismo postigli nikakvo ubrzanje već suprotno, usporavanje, i to trostruko. Kao što se može vidjeti na *Slici 4.8*, prilikom ulaska u paralelizacijski blok troši se procesorsko vrijeme na račvanje (eng. *fork*) u više dretvi, a prilikom izlaska na spajanje (eng. *join*) u jednu glavnu dretvu.



Slika 4.8: Proces račvanje-spajanje

Ako se uvedu oznake kao na *Slici 4.9*, tada se broj izvođenja račvanja i spajanja prilikom paralelizacije druge odnosno šeste petlje može opisati formulama navedenim u *Tablici 4.4*. Iz tih se formula može vidjeti da je broj račvanja i spajanja mnogo veći pri paralelizaciji šetste ugnježdene petlje nego druge, što je upravo i razlog zbog kojega se paralelizacija šeste ugnježdene petlje ne isplati.

Tablica 4.4: Broj račvanja i spajanja prilikom paralelizacije

$n_1 \cdot n_2 \cdot n_3 \cdot n_4 \cdot n_5 \cdot \frac{n_6}{num\_threads}$	Broj izvođenja procesa račvanja i spajanja prilikom paralelizacije šeste ugnježdene petlje.
$n_1 \cdot \frac{n_2}{num\_threads}$	Broj izvođenja procesa račvanja i spajanja prilikom paralelizacije druge ugnježdene petlje.

4.2.3 Ubrzanje korištenjem strojnog koda

Arhitektura x86 danas je sveprisutni standard za stolna i prijenosna računala te je platforma podržana od velike većine operacijskih sustava [10]. Upravo zbog toga, strojni kod koji generira prevodioc i koji je korišten prilikom ubrzanja pisan je u instrukcijskom skupu x86. Korišteno je također i proširenje instrukcijskog seta x86 poznato pod nazivom SSE [9]. Vektorskim tehnikama može se postići paralelizacija na nivou podataka, jer omogućava obradu više podataka pomoću samo jedne instrukcije koje se nazivaju još i vektorske instrukcije [11].

Kako bi se pronašla mjesta na kojima se isplatilo pokušati ubrzati program pisanjem dijelova programa u strojnom kodu, bilo je potrebno proučiti strojni kod koji generira sam prevoditelj na temelju napisanog programskog koda u programskom jeziku C++. U razvojnom okruženju MS Visual Studio, ovo se može postići na idući način*:* u izborniku projekta odabere se *Properties*, te se u izbornicima *Configuration Properties* → *C/C++* → *Output Files* u polje *ASM List Location* navede lokacija na kojoj korisnik želi da mu se spreme od prevodioca generirane datoteke koje sadrže strojni kod.

Iz proučavanja strojnog koda proizlaze dva moguća načina ubrzanja: modifikacija C++ koda na mjestima na kojima se u strojnom kodu generira mnogo memorijskih pristupa, naredbi strojnog množenja i ostalih sporih naredbi na način da se količina sporih naredbi smanji te zamijena dijelova C++ programskog koda strojnim kodom.

4.2.4 Zamjena C++ koda strojnim kodom

Pisanje dijelova programskog koda koristeći strojni kod može se postići na više načina: pisanjem dijelova koda kao funkcija pisanih u strojnom kodu smještenih u zasebnim datotekama ili pisanjem strojnog koda "inline", tj. direktno u dio programskog koda u C++-u gdje će se strojni kod izvršavati.

Korištenjem drugog načina, "inline" koda, izbjegavaju se pozivi funkcija, što može biti vremenski zahtjevna operacija jer zahtjeva korištenje naredbe grananja te je zbog toga odlučeno strojni kod koristiti na ovaj način. Ovaj način umetanja strojnog koda nije

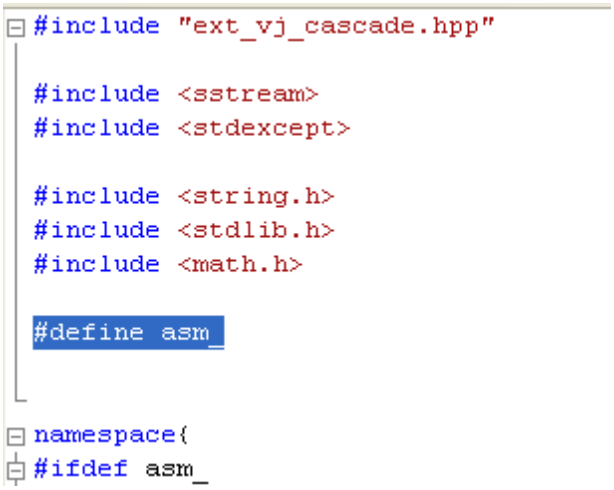
definiran standardom pa se tako interpretacija instrukcija može razlikovati od kompajlera do kompajlera. U nastavku je opisano korištenje ovog načina rada sa strojnim kodom u programskom okruženju Microsoft Visual Studio.

Podržana je naredba `__asm`, te se unutar blok koda koji pripada ovoj naredbi piše kod prema pravilima asemblera, kao što je prikazano na jednostavnom primjeru na *Slici 4.10*. Koristi se instrukcijski skup x86, a podržane su sve instrukcije koje dopušta platforma na kojoj se program izvodi. Kod se razlikuje od pravog strojnog koda prema tome što je pristup memoriji ostvaren preko varijabli, tj. umjesto memorijskog operanda instrukcije piše se ime C++ varijable. Imena varijabli zamjenjuju se prilikom prevođenja koda u prave memorijske operande no osim toga struktura strojnog koda ostaje ista.

```
int main(){
      unsigned int i=0;
      __asm{
            mov eax, i        ; MOVe value of i to EAX register
            add eax, 12       ; ADD 12 to EAX
            mov i, eax        ; MOVe the result back to i
      }
      cout << i;
      return  0;
}
```

Slika 4.10: Primjer korištenja naredbe __asm

Ovaj je način ubrzanja programskog koda korišten na tri mjesta u programu. Korištenje strojnog koda može se uključiti odnosno isključiti globalnom definicijom `asm_` (kod na *Slici 4.11* koristit će dijelove pisane strojnim kodom), tj. ukoliko ne postoji ta definicija, dijelovi pisani u strojnom kodu ne će se uzimati u obzir prilikom prevođenja.



*Slika 4.11: Globalna definicija asm_*

Na dva mjesta u kodu (jedno od kojih se može vidjeti na *Slici 4.12*) pristupa se polju uz izračun indeksa polja na kompliciran način, korištenjem množenja, četiri puta za redom. Kako se indeksi polja računaju na sličan način, korištenjem strojnog koda pokušalo se izbjeći uzastopno množenje istih podataka te dohvat istog podatka (množitelja) više puta iz memorije. Ubrzanje postignuto na ovaj način bilo je zanemarivo malo s obzirom na brzinu izvođenja programa te se može smatrati neuspjehom.

```
s0 = integral[xpom+max_width*ypom];
s1 =integral[xpom+max_width*yd];
s2 = integral[xd+max_width*ypom];
s3 = integral[xd+max_width*yd];
sum = s0 - s1 -s2 + s3;
sqsum = sqInt[xpom+max_width*ypom] - sqInt[xpom+max_width*yd]
-sqInt[xd+max_width*ypom] + sqInt[xd+max_width*yd];
```

Slika 4.12: Dio originalnog koda koji je zamjenjen strojnim kodom

Na ovaj se način također pokušala ubrzati funkcija zaokruživanja podatka s pomičnim zarezom na cijelobrojni podatak (funkcija `myround`). Staru i novu verziju funkcije `myround` može se vidjeti u *Tablici 4.5*.

Tablica 4.5: Nova i stara verzija funkcije myround

`inline __declspec(naked) int myround (double a){`
`__asm{`
`cvtsd2si eax, qword ptr[esp+4]`
`ret`
`}`
`}`
	`inline int myround (double a){`
	`return (int)(a+.5);`
	`}`

Pri ovome se koristila naredba `cvtsd2si` iz proširenog instrukcijskog skupa SSE. Naredba `cvtsd2si` (Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer with Truncation) služi za pretvaranje podatka tipa double, koji se zadaje kao drugi operand u obliku 64-bitne memorijske lokacije, u cjelobrojnu vrijednost koja se dobiva kao rezultat zaokruživanja na cijeli broj. Registar u koji se sprema rezultat naredbe navodi se kao prvi operand. Dodavanjem ove instrukcije postiglo se vidljivo ubrzanje koje ukupnom konačnom ubrzanju programa pridonosi 16% u verziji bez optimizacije prevodioca (ubrzanje sa 1,45 na 1,24 sekundi po slici) odnosno 7% sa optimizacijom prevodioca (ubrzanje sa 0,71 na 0.66 sekundi po slici).

4.2.5 Modifikacija C++ koda

C++ kod modificirao se na onim mjestima gdje se generiralo mnogo memorijskih poziva sa ciljem da se smanji njihov broj. Jedan od primjera ovakve modifikacije koda jest promjena redosljeda instrukcije koje se mogu izvršavati proizvoljnim redosljedom bez utjecaja na rezultate izvršavanja programa na način da se instrukcije koje koriste iste podatke iz memorije (tj. iste varijable) poredaju jedna iza druge te se tako izbjegne višestruko dohvaćanje tog podatka iz memorije.

Najviše memorijskih poziva pa tako i najuspješnije modifikacije ovog tipa pojavile su se na mjestu pristupa složenoj strukturi unutar programske petlje. Članovima ove strukture pristupalo se više puta unutar višestruke programske petlje, a svaki je pristup ovoj strukturi u strojnom kodu generirao pristup memorijskoj lokaciji uz iznimno složen izračun memorijske adrese. Prva ideja rješavanja ovog problema bilo je smještanje podataka ove strukture u lokalnu varijablu, čime bi se izbjeglo složeno izračunavanje memorijske adrese. Ovo se pokazalo dobrim za manju strukturu, ali pri kopiranju veće strukture sama operacija kopiranja velike količine podataka bila je sporija od ubrzanja koje se postiglo na taj način. Konačno riješenje problema postignuto je korištenjem reference, odnosno pamćenjem memorijske lokacije strukture koja se zbog toga nije morala svaki puta izračunavati. Na ovaj je način postignuto najveće ubrzanje koda od 77% (uz potpunu optimizaciju prevodioca i sva ostala ubrzanja, vrijeme izvođenja se smanji sa 1,17 na 0,66 sekundi po slici). Dijelovi novog programskog koga kojima je

postignuto ubrzanje kao i stara verziju tog programskog koda prikazani su na *Slici 4.13* (novi kod) i *Slici 4.14* (stari kod).

```
for (int i=0; i<count; i++)
{
  BoostClassifier& bc_i=boost_classifier[i];
  int haar_n = bc_i.count;

  ...

  for (int j = 0; j<haar_n; j++)
  {

    HaarClassifier& hc_ij=bc_i.haar_classifier[j];
    int rect_n = hc_ij.count;

    ...

    for (int k = 0; k<rect_n; k++)
    {
      Rect& hrect_ijk=hc_ij.rect[k];
      int xr = hrect_ijk.x;
      int yr = hrect_ijk.y;
      int wr = hrect_ijk.width;
      int hr = hrect_ijk.height;

      ...
    }
    ...
  }
  ...
}
```

Slika 4.13: Novi programski kod koji koristi reference

```
for (int i=0; i<count; i++)
{
  int haar_n = boost_classifier[i].count;

  ...

  for (int j = 0; j<haar_n; j++)
  {

    int rect_n = boost_classifier[i].haar_classifier[j].count;

    ...

    for (int k = 0; k<rect_n; k++)
    {
      int xr = boost_classifier[i].haar_classifier[j].rect[k].x;
      int yr = boost_classifier[i].haar_classifier[j].rect[k].y;
      int wr = boost_classifier[i].haar_classifier[j].rect[k].width;
      int hr = boost_classifier[i].haar_classifier[j].rect[k].height;

      ...
    }
    ...
  }
  ...
}
```

Slika 4.14: Stari programski kod koji ne koristi reference

5. Poboljšanje algoritma praćenja

5.1 Istovremeno praćenje znakova u algoritmu KLT

Dosadašnja implementacija koda uključivala je praćenje isključivo jednog objekta, dok su istovremeno neki dijelovi koda podržavali praćenje više objekata. Nadogradnja postojećeg dijela svela se na nadogradnju i izmjenu funkcije alg_klt2::process te pisanje pomoćne funkcije. Rad koji je napravljen najlakše je objasniti analizom navedene funkcije i naglašavanjem učinjenih promjena. U odjeljku 5.1.1 opisana je funkcija alg_klt2 te potrebna pomoćna funkcija, a odjeljak 5.1.2 naglašava promijene koje su morale biti napravljene da bi algoritam bio u mogućnosti pratiti više znakova, odnosno objekata.

*5.1.1 Opis funkcije **alg_klt2::process***

Ova funkcija direktno komunicira sa algoritmom KLT, šaljući mu podatke poput trenutne slike u videu (eng. *frame*), izabranih točaka te dijelova slika koji se prate. Također šalje praćene točke vektoru `std::vector<Point2D> points_` koje su kao parametar bile potrebne u povezivanju ovoga algoritma sa ostatkom projekta. Na trenutnoj slici se odabiru značajke (*engl. features*). Značajka koju možemo nazvati točka interesa je točka ili skup točaka pomoću kojih algoritam slijedi pojedine objekte na slici.

Funkcija je implementacija istoimene virtualne funkcije. Parametri funkcije mogu se vidjeti na S*lici 5.1* koja prikazuje njezinu deklaraciju. Kao ulazne parametre funkcija prima sliku iz videa u kojem se prate znakovi i događaje koji su bili uzrokovani korisnikovim djelovanjem (npr. klik miša). Treći parametar nije bitan, pa je izostavljen (napisan je samo tip). Izlaznih parametara nema.

```
void alg_klt2::process(const img_wrap& src, const win_event_vectorAbstract&
events, int);
```

Slika 5.1: Deklaracija funkcije alg_klt2::process

Parametar `src` je jedna slika videa pa prema tome njezin format može varirati. Kao primjer koji je korišten za opis ove funkcije uzet je video čiji su okviri formata 720 x 576 piksela, 24-bitnih boja. Njezina veličina se zadržava, ali se boje pretvaraju u paletu

sivih nijansi (crno-bijela slika, veličina piksela 8 bita) idućim dijelom koda:

```
src_.reformat(img_fmt(src.fmt(), 1));
```

Algoritmu se predaje trenutna slika na kojoj je korisnik mogao označiti dijelove za praćenje:

```
klt_.process(src_);
```

Pojedini objekti mogu se označiti jednostrukim klikom miša na taj objekt, ili označavanjem neke površine na slici.

Kako se mora pratiti više točaka upotrebljen je vektor `vector<grid::Pixel> vecClicks` koji sadrži podatke o točkama koje je korisnik odabrao na tekučoj slici. Vektor se sastoji od podataka tipa `grid::Pixel` u kojem su zapisani podatci o dvijema kordinatama (x i y) pojedinog piksela.

Dijelovi slike označeni za praćenje se pohranjuju u vektor `vector<img_wrap> vecReferences` koji je globalno definiran za razred `alg_klt2`. Struktura *img_wrap* sadrži podatke o slici, kao i samu sliku. Za pohranjivanje lokacija svih valjanih točaka služi vektor `vector<math::Point2D> vecCurPos`.

Radi ubrzanja algoritma izmjena vektora `vecCurPos` se vrši prilikom korisničkog unosa nove točke na način da se nadoda nova točka, a izbace sve one točke koje se ne prate. Prilikom njegove izmjene na analogan način se vrši mijenjanje elemenata vektora `vecReferences` i `vecCurMag`.

Kao što je već spomenuto, u svakom slici videa korisnik može označiti točku ili neku površinu. Točka se označava jednim klikom na tekući video, dok se kvadratna površina proizvoljne veličine označava klikom na dijagonalno simetrične vrhove željene površine. Prilikom označavanja sama funkcija pamti točke u gornjem lijevom i donjem desnom kutu površine, dakle za označavanje površine koristi se paran broj točaka. Da bi se razlikovalo je li korisnik označio pravokutnik ili samu točku potrebno je provjeriti broj točaka koji je označen, a zapisan u vektoru `vecClicks`. Ukoliko je označena samo točka simulira se označen kvadrat veličine 15x15 piksela na način da se prepravi navedeni vektor. Na kraju korisničke interakcije sa sučeljem (nad jednom slikom videa), svi se uzastopni parovi točaka tumače kao površine i prepravljaju tako da predstavljaju

Slika 5.6: Slika nastala spajanjem elemenata vektora vecReferences

gornji lijevi i donji desni kut površine. Ukoliko je broj točaka u vektoru neparan, zadnji se unos u vektoru tumači kao točka, a ne površina. Umjesto tog unosa dodaje se predefiniran kvadrat dimenzija 15x15 piksela sa centrom u unesenoj točci, na način da se x i y koordinati unesene točke oduzima po 7 piksela da bi se dobila točka koja označava gornji lijevi kut pravokutnika i zatim se koordinatama tako dobivene točke dodaje po 15 piksela da bi se dobila točka koja odgovara donjem desnom kutu virtualne označene površine. Kod koji prepravlja vektor `vecClicks` na ovaj način može se vidjeti na *Slici 5.*2.

```
if (vecClicks.size()%2==1){
    vecClicks.back()-=grid::Pixel(7,7);
    vecClicks.push_back(
      vecClicks.back()+grid::Pixel(15,15));
}
```

Slika 5.2: Pretvaranje označene točke u virtualnu površinu

Gornje lijeve kordinate tako pripremljenih točka se dodaju u vektor `vector<grid::Pixel> vecLocs`. Pri tome treba paziti da se doda odgovarajuća točka jer korisnik priklikom odabira površine površinu može označiti na više načina, npr. prvo lijevi gornji kut, a zatim desni donji. Popunjava se i vektor `vector<int> vecScales` u kojem su zapisani podaci o veličini stranice kvadrata koji predstavlja odabranu površinu za praćenje koja u slučaju odabira jedne točke iznosi konstantno zadanih 15 piksela. Općenito se za stranicu kvadrata uzima veća od stranica pravokutnika označenih od korisnika, tj. maksimum širine i visine označene površine.

Sljedeće što je potrebno napraviti jest odabrati stupanj piramide u ovisnosti o veličini odabrane površine. Za površinu veličine 15x15 piksela dobivenu odabirom jedne točke faktor piramide jednak je 1. Pri odabiru površine druge veličine bira se ona razina piramide pomoću koje se površina može skalirati približno na vrijednost površine

dobivene odabirom samo jedne točke. Kod kojim se vrši odabir faktora piramide vidljiv je na *Slici 5.3*.

```
img_wrap imgPyrAtFeatureScale;
int factorPyr=1;
for (int i=0; i<klt_.nPyrLevels(); ++i, factorPyr*=2){
  if (scaleFeature<2*factorPyr){
    klt_.imgPyr(i, imgPyrAtFeatureScale);
    break;
  }
}
```

Slika 5.3: Odabir faktora piramide

Površina se zatim isijeca iz slike dodajući sa svake strane površine pomak koji je zadan konstantom `offsetKLT` i koji iznosi 1. Zbog tog pomaka, veličina površine koja predstavlja jednu točku nakon izrezivanja biti će 17x17 piksela. Kako algoritam KLT radi sa konstantnom rezolucijom površine, sve se slike pohranjene u vektoru `imgFeature` skaliraju na veličinu 17x17 piksela. Primjer skalirane slike koja se pohranjuje u varijablu `imgSeed`, može se vidjeti na *Slici 5.4.* Ova je slika označeni dio znaka sa *Slike 5.12*.

Slika 5.4: Slika imgSeed

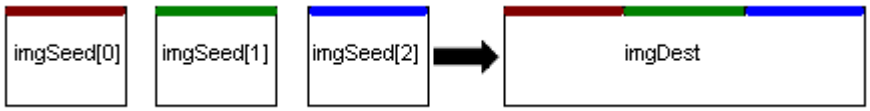
Dio programskog koda kojim se obavlja skaliranje površine i spremanje tako dobivene značajke u vektor `vecReferences` može se vidjeti na *Slici 5.5*.

```
img_wrap imgSeed(img_fmt(szKLT+2*offsetKLT, szKLT+2*offsetKLT,1));
ipp::rescale(imgFeature_, imgSeed);
...
vecReferences.push_back(imgSeed);
```

Slika 5.5: Skaliranje površine i dodavanje u vecReferences

Nakon obrade svake odabrane površine, sve se te površine (tj. svi elementi vektora `vecReferences`) spajaju u jednu sliku `imgDest`, kao što je prikazano na *Slici 5.6*. Ovaj je postupak simbolički prikazan na *Slici 5.7.*

Postupak spajanja slika u odredišnu sliku ostvaruje se kopiranjem redova piksela (okteta) slika iz vektora `vecReferences` u jedan red odredišne slike `imgDest`. Postupak spajanja obavlja sljedeći dio koda:

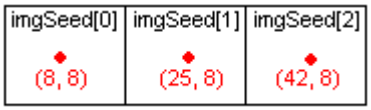


Slika 5.7: Spajanje elemenata vektora vecReferences

```
static void img_util_tile(std::vector<img_wrap> seedSources, img_wrap&
imgDest);
```

Kordinate centara svake od spojenih slika (prikazano na *Slici 5.8)* se pohranjuju u vektor `vector<math::Point2D> pts`, koji je, uz sliku `imgDest` potreban za inicijalizaciju algoritma KLT. Prije same inicijalizacije algoritma, ponovo se namješta piramida:

```
klt_.setPyramid(1,2,0.01);
klt_.init(pts, imgDest);
```



Slika 5.8: Koordinate centara spojenih slika

U vektor `vecCurPos` spremaju se koordinate sredina odabranih površina na početnoj slici, čije je izračunavanje i popunjavanje prikazano na *Slici 5.9*. Pri izračunu se koristi vektor `vecScales` u kojem su zapisane veličine praćenih površina i vektor `vecLocs` koji sadrži koordinate odabranih površina, kako bi se uračunala odstupanja nastala zbog odabira praćene površine drugačije (tj. veće od) odabrane površine.

```
int szReal=vecScales[i-countFeaturesPrev];
math::Point2D ptFeature( grid2math(
    vecLocs[i-countFeaturesPrev] +
    grid::Pixel(szReal/2,szReal/2)));
vecCurPos.push_back(ptFeature);
```

Slika 5.9: Popunjavanje vecCurPos

Omjeri veličine slika i konstante `szKlt` upisuju se u vektor `vecCurMag`. Nakon

ponovnog namještanja razine piramide, algoritmu se za svaku točku predaje pozicija njezinog centra na trenutnoj slici (eng. *frame*) i magnituda pratećih značajki, kako je prikazano u kodu na *Slici 5.10*.

```
for(int i=0; i<(int)vecCurPos.size(); i++){
   klt_.resume(i, vecCurPos[i], vecCurMag[i]);
}
```

Slika 5.10: Predavanje podataka za trenutnu sliku algoritmu KLT

Na kraju označavamo točke na videu. Točka se dobivanju naredbom `klt_.fThis(i)`. Za svaku točku provjerava se je li još uvijek praćena naredbom klt_.fAge(i) te ako jest iscrtava se okvir oko praćenog dijela slike. Same točke *libmastif*-u vraćaju se tako da se zapišu u vektor `std::vector<Point2D> points_`, što je prikazano dijelom koda na *Slici 5.11*.

```
ann_.addPixel(grid::Pixel(x0,y0),6, "",
        win_ann_abstract::PS_BoldHuge);

ann_.addPixel(grid::Pixel(x1,y1),6, "",
        win_ann_abstract::PS_BoldHuge);
```

Slika 5.11: Vraćanje točaka libmastif-u

Primjer slike iz videa u kojem se prati više objekata algoritmom KLT može se vidjeti na *Slici 5.12.*Veliki pravokutnik označava površinu definiranu od korisnika (nastalu pomoću dva klika mišem), dok mane površine predstavljaju točke koje se prate.

Slika 5.12: Praćenje više objekata algoritmom KLT

Detalji vezani uz predaju praćenih točaka drugim dijelovima programa nisu obrađeni u okviru ove dokumentacije, jer taj dio implementacije nije u potpunosti dovršen. U opisu funkcije u okviru ovog poglavlja najmerno su naglašavani upravo oni dijelovi koji su bili najviše izmjenjeni i nadopunjeni.

5.1.2 Izmjene izvršene prilikom prilagodbe algoritma

Kako je trebalo omogućiti praćenje više objekata, bilo je potrebno zapamtiti točke koje su odabrane od strane korisnika. Uveden je novi vektor `vecClicks`. Nadalje potrebno je bilo zapamtiti sve trenutno praćene točke – vektor `vecCurPos`. Za slike koje su se pratile uveden je vektor `vecReferences` te slika koja predstavlja spojene praćene slike - `imgDest`, a napisana je i funckcija `img_util_tile` za spajanje slika. Praćene

slike nisu morale biti jednakih veličina, a za njihovo je pamćenje zadužen vektor `vecScales` dok su omjeri između veličine slike i konstante `szKlt` pamte u vektoru `vecCurMag`. Kako sada imamo spojenu sliku (`imgDest`) morali smo odrediti koordinate centara spojenih slika što je učinjeno pomoću vektora `pts`.

Sama izmjena koda uključivala je popunjavanje i uporabu navedenih strukura podataka. Tako se kod obrade korisničkog unosa (klikovi miša na slici) trebalo odrediti o kakvoj se izabranoj površini radi, modificirati ulazne točke te ih potom predati na daljnje korištenje. Uvođenjem više slika za praćenje, osim njihovog pamćenja, ukazala se potreba za skaliranjem slika na odgovarajuću veličinu i spajanjem više slika u jednu uz određivanje njihovih centara. Posebna pažnja posvećena je osvježavanju vektora koji nose podatke o praćenim dijelovima slike video zapisa. Navedene promjene su uspješno ukomponirane u postojeću implementaciju.

5.2 Ubrzanje izvođenja algoritma KLT

Jedna od značajnih funkcija u postupku automatskog prepoznavanja i označavanja prometnih znakova je praćenje detektiranih znakova nakon njihove detekcije radi izbjegavanja višestruke detekcije istog znaka na nekoj dionici. Naš projekt u tu svrhu koristi se algoritmom KLT odnosno metodom diferencijalnog praćenja po Kanadeu, Lucasu i Tomasiju. U ovom postupku praćenja jedan od vremenskih najzahtjevnijih postupaka je konvolucija slike koja se obrađuje Gaussovim filtrom. Kako se ubrzanjem vremenski najzahtjevnijeg dijela algoritma postiže najveći utjecaj na brzinu izvođenja cijelog algoritma, ubrzanja su usredotočena na postupak konvolucije u cilju maksimalnog ubrzanja zahtjevnog postupka praćenja prometnih znakova.

5.2.1 Kako ubrzati konvoluciju?

Podatak koji se obrađuje i nad kojim je potrebno izvršiti konvoluciju u našem projektu su dvodimenzionalne slike. Dvodimenzionalna konvolucija vrši se množenjem elemenata slike s elementima dvodimenzionalne konvolucijske jezgre po formuli:

$$r[i,j] = \sum_m \sum_n s[i-m, j-n]k[m,n]$$

U gornjoj formuli *r[i,j]* je rezultat konvolucije, *s[i-j, j-n]* su elementi slike koji se množe sa *k[m,n]* koji predstavlja element jezgre s kojim se množi. Ovakvom postupku potrebno je $m \cdot n \cdot p \cdot q$ operacija množenja za obaviti konvoluciju nad slikom dimenzija *p*x*q*.

Ubrzanje ovog algoritma možemo dobiti već primjenom elementarnih znanja iz matematike koja nas uči da postoje matrice koje možemo zapisati kao umnožak vektor retka i vektor stupca. Ukoliko našu konvolucijsku jezgru razdvojimo na jedan redak i jedan stupac dobit ćemo separabilni filtar. Primjer razbijanja dvodimenzionalne konvolucijske jezgre na ovakav način prikazan je na *Slici 5.13.* Nakon što je razdvojimo

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

Slika 5.13: Razbijanje dvodimenzionalne konvolucijske jezgre

na ovakav način potrebno je dva puta obaviti konvoluciju nad slikom; prvo konvuliramo sliku s jednim elementom separabilnog filtra te potom s drugim. Ovakvim postupkom konvolucije potrebno nam je $p \cdot q \cdot (m+n)$ operacija množenja za obaviti konvoluciju, te se u odnosu na množenje dvodimenzionalnom konvolucijskom jezgrom dobiva ubrzanje od $\frac{m*n}{m+n}$ puta.

Zbog gore prikazanih razloga u našem projektu za konvoluciju koristimo funkciju `convolveSeparate()` koja poziva funkcije `convolveHoriiz()` i `convolveVert()` za horizontalnu odnosno vertikalnu konvoluciju.

5.2.2 Ubrzanje konvolucije korištenjem SIMD skupa instrukcija

Glavni problem kod izvođenja postupka konvolucije predstavljao je vrlo veliki broj operacija množenja koje je potrebno izvršiti. Rezultat toga je veliko vrijeme izvođenja postupka koje direktno utječe na brzinu rada čitavog postupka označavanja znakova. Pažljivim razmatranjem postupka konvolucije može se primjetiti kako se postupak sastoji od mnogo množenja elemenata slike istim konvolucijskim elementom. Zaključuje se da se postupak može ubrzati paralelnim množenjem više slikovnih elemenata istim elementom.

Ovakvu vrstu paralelizma, koja se naziva još i paralelizam na nivou podataka, omogućava upravo SIMD skup procesorskih instrukcija. Ovim se skupom instrukcija postiže paralelni rad sa više podataka tipa float pomoću procesorskih registara iz skupa *xmm* [11].

Pošto se u svaki *xmm* registar može učitati do četiri 32-bitna podatka, glavna ideja je na sve četiri lokacije u jedan *xmm* registar učitati trenutni element konvolucijske jezgre. Nakon toga učitaju se 4 slikovna elementa u drugi *xmm* registar koji se množi sa konvolucijskom jezgrom u prvom *xmm* registru. Na ovaj se način rezultat konvolucije 4 slikovna elementa dobiva u samo jednom ciklusu. Ovaj postupak izvodi se instrukcijama strojnog koda prikazanim na *Slici 5.13* [16].

```
movss xmm1, xmmword ptr [ebx+eax*4]
shufps xmm1, xmm1,0
movups xmm2, xmmword ptr [esi+eax*4]
mulps xmm2,xmm1
addps xmm3,xmm2
```

Slika 5.13: Strojni kod konvolucije

Prva naredba na *Slici 5.13*, `movss` učitava prvi 32 bitni podatak konvolucijske jezgre u registar `xmm1`. Druga naredba `shufps` kopira taj podatak 4 puta tako da se popuni svih 128 bitova u registru čime dobivamo 4 puta zapisan isti podatak u registru. Treća naredba `movups` učitava 4 slikovna elementa u registrar `xmm1`. Naredbom `mulps` množimo elemente registara xmm1 i xmm2, te potom rezultat spremamo u registar `xmm3`. Upravo naredba `mulps` omogućava dobivanje rezultat konvolucije četiri elementa u jednom ciklusu.

5.2.3 Implementacija konvolucije pomoću SIMD skupa instrukcija

Kako bi se iskoristile mogućnosti koje pruža SIMD skup instrukcija, implementaciju algoritma KLT dodane su nove funkcije `convolveSeparateSSE()`, `convolveHorizSSE()` i `convolveVertSSE()`. Za korištenje SIMD implementacije konvolucije, poziv funkcije `convolveSeparate()` zamijeni se pozivom funkcije `convolveSeparateSSE()`.

Zbog brzine izvođenja strojnih instrukcija, ali i specifičnosti MSVC kompajlera te korištenih struktura podataka, funkcije `convolveHorizSSE()` i `convolveVertSSE()` napisane su većinom u strojnom jeziku.

Specifičnost ove SIMD implementacije nalazi se u tome da su funkcije `convolveHorizSSE()` i `convolveVertSSE()` praktički iste. Zbog način spremanja dvodimenzionalnih polja u memoriju koji onemogućava brzo uzimanje 4 uzastopna vertikalna elementa slike SIMD instrukcijama, nakon izvršavanja horizontalne konvolucije, rezultantna se slika transponira. Nad transponiranom slikom još jednom se vrši horizontalna konvolucija, ovaj put sa vertikalnim kernelom. Ponovnim transponiranjem dobivenog rezultata dobiva se ispravna rezultantna slika.

5.2.4 Rezultati ubrzanja SIMD skupom instrukcija

Rezultati usporedbe brzine izvršavanja algoritma kada koristi staru implementaciju konvolucije, napisanu u jeziku C++ uz optimizaciju prevodioca, sa novom implementacijom konvolucije napisanom strojnim jezikom uz korištenje SIMD skupa instrukcija nalaze se u *Tablici 5.1*.

Tablica 5.1: Rezultati usporedbe C++ i SIMD implementacije konvolucije

Broj slika	**Vrijeme SIMD**	**Vrijeme C++**	**fps SIMD**	**fps C++**	**% ubrzanje**
1	0.6s	1.1s	1.6	0.9	83.33
100	23.9s	32.4s	4.2	3.1	35.56
200	48.4	71.0s	4.1	2.8	46.69

Iz priloženih rezultata vidljivo je da se korištenjem SIMD instrukcija, odnosno podatkovnog paralelizma, postiže ubrzanje od čak 48% u odnosu na staru implementaciju. Također je vidljivo da podatkovni paralelizam dolazi do većeg izražaja na većoj količini podataka, odnosno ubrzanje na 100 slika je manje (36%) nego ubrzanje na 200 slika.

5.2.5 Daljnje ideje za ubrzanje algoritma

Iako implementacija konvolucije SIMD skupom instrukcija značajno ubrzava množenje slikovnih elemenata sa elementima jezgre i dalje značajni se dio vremena gubi na transponiranje slike nakon izvođenja prve i druge horizontalne konvolucije (vertikalna konvolucija je također implementirana horizontalnom konvolucijom, kao što je navedeno u odjeljku 5.3.2). Ovo otvara mogućnosti ubrzanja u vidu implementacije funkcije za horizontalnu konvoluciju s automatskim transponiranjem rezultata konvolucije. Ovime bi se značajno uštedilo na vremenu pristupa podacima i njihovom premještanju te ostvarilo dodatno ubrzanje.

6. Učenje poboljšanog detektora

6.1 Evaluacija postojeće implementacije algoritma Viole i Jonesa

Implementacija je evaluirana tako da je korišten zaseban skup slika za testiranje te zaseban skup slika za evaluaciju. Prije evaluiranja je potrebno u projekt (u datoteci `ext_vj.cpp`) dodati sljedeće naredbe:

1. prije funkcije `void ext_vj()` dodati:

```
#define EVAL 1
#if EVAL
 #include <sys\timeb.h>
 #include <fstream>
#endif
```

2. unutar funkcije `void ext_vj()`, prije `int width = src.fmt().width()` dodati:

```
#if EVAL
 struct timeb evalStart, evalEnd;
 time_t evalLen;
 ftime (&evalStart);
#endif
```

3. nakon sljedećeg dijela koda:

```
free (imgGrey);
free (imgInt);
free (imgSqInt);
```

dodati:

```
#if EVAL
 ftime (&evalEnd);
  evalLen = 1000 * (evalEnd.time - evalStart.time) +
               evalEnd.millitm - evalStart.millitm;
  std::fstream evalOut("eval.out", std::ios::out | std::ios::app);
  if (detections.size() == 0) {
              evalOut << evalStart.time << " 0 0 0 0 " << evalLen <<
endl;
  } else {
    for (size_t i=0; i<detections.size(); ++i){
              ext_vj_rect& rc(detections[i]);
               evalOut << evalStart.time << " " << rc[0] << " " <<
(height-rc[1]) << " " << rc[2] << " " << (height-rc[3]) << " " <<
evalLen << endl;
    }
  }
  evalOut.close();
#endif
```

Evaluacija se pokreće na sljedeći način: kada se u kodu uključi evaluacija

(*EVAL 1*) i kod se kompajlira, prilikom rada u radnom direktoriju stvori se datoteka *eval.out*. Evaluacija se vrši pozivom *eval.py* skripte koja je priložena na dnu odjeljka i kojoj se predaju datoteka uzoraka tipa .*seq* (drugi argument) koja je služila za testiranje programa i koja je generirala datoteku *eval.out* koja se predaje kao prvi argument. Prije pokretanja rada programa, ako *eval.out* datoteka postoji u radnom direktoriju, potrebno ju je obrisati (ili obrisati njen sadržaj). Rezultati evaluacije sa objašnjenjima i formule za izračun pojedinih podataka dani su u *Tablici 6.1.*

Tablica 6.1: Rezultati evaluacije

Naziv podatka	**Vrijednost podatka**	**Objašnjenje**	**Formula za izračun**
broj slika	91	broj slika na kojima je projekt evaluiran	
true positive (tp)	90	broj slika za koje implementacija daje ispravne rezultate	
false positive (fp)	69	broj slika na kojima su nađeni objekti koje nismo tražili	
false negative (fn)	11	broj slika na kojima nisu nađeni traženi objekti	
Preciznost (P*recision)*	0.566037735849	udio točnih odziva od svih dobivenih odziva	$\text{Precision} = \frac{tp}{tp + fp}$
Odziv (R*ecall)*	0.891089108911	udio nađenih objekata od svih koje smo tražili	$\text{Recall} = \frac{tp}{tp + fn}$
F1	0.692307692308	statistička mjera, izračun po formuli	$F1 = \frac{2 \cdot Precision \cdot Recall}{Precission + Recall}$

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import re
import sys

def usporedi(k, e, tolerancija):
	return (abs(e[0] - k[0]) <= tolerancija) and (abs(e[1] - k[1]) <= tolerancija) and (abs(e[2] - k[2]) <= tolerancija) and (abs(e[3] - k[3]) <= tolerancija)

def main(args):
	# Tolerancija za svaku koordinatu
	tolerancija = 20

	if len(args) < 3:
		print "Uporaba: " + args[0] + " <eval_data> <seq_data> [<tolerancija=" + str(tolerancija) + "px>]"
		sys.exit(-1)

	if len(args) == 4:
		tolerancija = int(args[3])

	# Eval data
	deval = open(args[1])
	evalStr = deval.read()
	evalPattern = re.compile("(\d+) (\d+) (\d+) (\d+) (\d+) (\d+)")

	evalStrTpls = evalPattern.findall(evalStr)
	currTime = evalStrTpls[0][0]
	eval = []
	single = []
	for e in evalStrTpls:
		elem = (int(e[1]), int(e[2]), int(e[3]), int(e[4]), int(e[5]))
		if not (e[0] == currTime):
			eval.append(single[:])
			single = []
			currTime = e[0]

		single.append(elem)

	if not (len(single) == 0): eval.append(single[:])

	# Seq data
	seq = open(args[2])
	lines = seq.readlines()

	groupPattern = re.compile("(x=\d+,y=\d+,w=\d+,h=\d+)")
	koordsPattern = re.compile("x=(\d+),y=(\d+),w=(\d+),h=(\d+)")
	koords = []
	for l in lines:
		mg = groupPattern.findall(l)
		kgroup = []
		for g in mg:
			mo = koordsPattern.match(g)
			k = mo.groups()
```

```
                    kint    =    (int(k[0]),int(k[1]),int(k[0])+int(k[3]),int(k[1])
+int(k[2]))
                    kgroup.append(kint)
              if not (len(kgroup) == 0): koords.append(kgroup[:])

       # Usporedba
       if not (len(koords) == len(eval)):
              print "Nesklad u podatcima! seq datoteka i eval datoteka nemaju
jednak broj slika!"
              print "seq: " + str(len(koords)) + ", eval: " + str(len(eval))
              sys.exit(-1)
       samplesNum = len(koords)
       FP = 0
       FN = 0
       TP = 0
       for i in xrange(0, samplesNum):
              skoord = koords[i]
              ekoord = eval[i]
              TPlist = []
              for j in xrange(0, len(ekoord)):
                    if ekoord[j][0] == 0 and ekoord[j][1] == 0 and ekoord[j][2] ==
0 and ekoord[j][3] == 0:
                           TPlist.append(ekoord[j])
                           continue

                    for sk in skoord:
                           if usporedi(sk, ekoord[j], tolerancija):
                                  TPlist.append(ekoord[j])
                                  TP += 1
                                  skoord.remove(sk)
                                  break
              for ek in TPlist:
                    ekoord.remove(ek)
              FN += len(skoord)
              FP += len(ekoord)

       P = TP*1.0/(TP+FP)
       R = TP*1.0/(TP+FN)
       if P == 0 and R == 0: F1 = 0
       else: F1 = 2*P*R/(P+R)

       print "Number of images: " + str(samplesNum)
       print "True positive: " + str(TP)
       print "False positive: " + str(FP)
       print "False negative: " + str(FN)
       print "Precision: " + str(P)
       print "Recall: " + str(R)
       print "F1: " + str(F1)

if __name__ == "__main__":
       main(sys.argv[:])
```

kod skripte eval.py

6.2 Učenje detektora za trokutaste i okrugle znakove

U sklopu ovog projekta korištena je biblioteka *OpenCV*, čiji glavni cilj jest pronaći tražene objekte i evaluirati uspješnost njihovog pronalaska. Biblioteka *OpenCV* sadrži gotove metode za detekciju objekata Viole i Jonesa i klasifikatore za pronalaženje ljudskog lica. Slijedi procedura za izradu klasifikatora:

1. Da bi se klasifikator izgradio potrebno je sastaviti dvije skupine slika, odnosno primjera. Dakle, potrebni su nam takozvani pozitivni primjeri (skup slika na kojima se pojavljuju traženi objekti) i negativni primjeri ili pozadine (sličan skup slika na kojem se ne pojavljuje traženi objekt). Cilj je da klasifikator „nauči“ razliku između nečega što jest znak i svega ostalog što se nalazi u pozadini.
2. Pozitivni uzorci za ispitivanje i testiranje uzeti su iz različitih videa.
3. Potrebno je isjeći i skalirati skup pozitivnih znakova za učenje. To se radi pozivom programa: `opencv_createsamples.exe`. Poziv može biti: `opencv_createsamples.exe –info trokuti.dat –vec trokuti.vec`
4. Rezultat poziva naredbe `opencv_createsamples.exe` je datoteka tipa *.vec* koja sadrži slike zapisane vektorskim formatom
5. Poziva se *opencv_haartraining.exe* koji prima sljedeće parametre:
    - datoteka tipa *.vec* pozitivnih primjera
    - popi pozadina
    - broj koliko se pozitivnih primjera koristi
    - parametar: koliko želimo da se iz tih pozadina napravi negativnih primjera
    - parametar: 'način na koji program uči':
        - *nstage* (eng. *number of stages*)
        - *mhr* (eng. *minimum hit rate*) – npr.: mhr=0,995, što znači da za svaki *stage* (kaskadu) 99,5% pozitivnih primjera mora algoritmom biti potvrđeno kao pozitivni primjeri
        - *mfa* (eng. *maximum false alarm*) – npr. ako je mfa=0.4, to znači da najviše 40% negativnih primjera (eng. *false*

positive-a) smije biti označeno

6. OpenCV u svakoj kaskadi izgenerira upravo onoliko uzoraka koliko je potrebno da se zadovolje parametri *mhr* i *mfa*
7. Ukoliko nije moguće generirati toliko uzoraka učenje se prekida i preostanu samo dotad izgenerirane kaskade.

6.3 Ručno označavanje znakova za evaluaciju

Za evaluaciju klasifikatora (čija je izgradnja opisana u prethodnom potpoglavlju) potreban je skup pozitivnih primjera. Skup pozitivnih primjera dobiven je alatom za označavanje *Marker.* Alat *Marker* omogućava označavanje položaja i tipova prometnih znakova u video snimkama. Može se zajedno s uputama preuzeti s adrese: *http://www.zemris.fer.hr/~ssegvic/mastif/results_hr.shtml.* Da bi se *Marker* mogao pokrenuti, potrebno je instalirati *Java Runtime Environment* (JRE) i *Windows Media Player 11*.

Prilikom označavanja znakova korišten je video *090722_1_ipv.wmv*. Video traje 1 sat, 28 minuta i 16 sekundi i veličine je 1.89GB. Snimljen je iz vozila koje se kreće kolnikom (vozilo se ne vidi). Popis znakova koji su označavani na videu nalazi se na adresi: *http://narodne-novine.nn.hr/clanci/sluzbeni/288185.html.* Klikom na različite kategorije znakova mogu se pogledati slike opisanih znakova. Važno je pročitati opis znaka, kako bi se stekao dojam o tome što sve neki znak mora sadržavati da bi se mogao identificirati kao taj znak. To se najbolje vidi na primjeru raznih tabla, odnosno znakova kategorije E.

Slijedi procedura za prevođenje i pokretanje *Markera* iz naredbenog retka [13]. Dakle, desni klik na direktorij *marker* i *Open Command Window Here.* Prije prevođenja potrebno je na računalo instalirati Javu i dodati javu i javac u *path.* To se, ukoliko je Java instalirana u direktorij `C:\Program Files\Java\jdk1.6.0_12`, radii na slijedeći način: `set PATH="C:\Program Files\Java\jdk1.6.0_12\bin";"C:\Program Files\Java\jdk1.6.0_12\jre\bin";%PATH%.`

Slijed naradba za prevođenje:

- `mkdir bin`

- `javac -extdirs ./lib/ -sourcepath src -d bin src/hr/fer/zemris/cv/marker/Main.java`
- `cp src/hr/fer/zemris/cv/marker/messages.properties bin/hr/fer/zemris/cv/marker/`

Marker se se pokreće naredbom: `java -cp lib -cp bin. hr.fer.zemris.cv.marker.Main.`

Slijedi postupak za označavanje znakova. Nakon pokretanja *Markera* potrebno je učitati video (*Datoteka -> Otvori*...) i pritisnuti tipku Play. Čim se uoči znak, potrebno je pauzirati video i znak označiti. Znak se označava odabirom tipke na kojoj je nacrtana olovka i od velike je važnosti da se označi precizno – od ruba do ruba, nikakav slobodni prostor nije dozvoljen. Znak treba označiti četiri puta, prvi put kad se tek uoči (kad možemo razaznati o kojem se znaku radi) i zadnji put prije nego što nestane iz vidokruga, no mora se cijeli vidjeti. Između prvog i zadnjeg okvira znak je potrebno označiti još dva puta i to na pravilnim razmacima. Nakon označavanja znaka, potrebno je dva puta kliknuti na okvir i odabrati o kojem se znaku radi. Oznake se spremaju na način da se odabere *Datoteka->Spremi Oznake*. U slučaju da video nije moguće označiti do kraja naziv spremljenih oznaka treba završavati s *_partial.*

Marker zapisuje oznake u jednom od dva formata: *seq* ili *vse*. Prethodnom se radnjom oznake spremaju u .vse formatu, čiji je oblik sljedeći: [F775]:A05@(x=634,y=209,w=73,h=72) što opisuje *brojFramea(775)/ vrstaPrometnogZnaka(A05)/koordinate(gornja lijeva)/visina/širina*. Marker zna navedenu informaciju pretvoriti u .seq format.

Nakon što se označi čitav video, potrebno je napraviti .bmp slike koje će sačinjavati skup pozitivnih primjera. U ovom slučaju nije bilo moguće cijeli video označiti odjednom, rezultat označavanja bile su tri .vse datoteke, koje su kasnije prilikom stvaranja .bmp slika zamijenjene jednom .vse datotekom koja sadrži sve podatke. Na videu su označeni svi znakovi koji se pojavljuju, to su znakovi u kategorijama od A do E.

Evaluacija se vrši samo na skupu koji sadrži okrugle i trokutaste znakove i njih je potrebno izdvojiti. Zato je potrebno odabrati *Pretraga – Traži...* i pod *Regularni izraz* napisati kategoriju znakova koja se traži. U ovom slučaju radi se o cijeloj kategoriji A te

o znakovima od B02 do B43 i treba kliknuti *Dodaj*. Pod okruglim znakovima smatraju se okrugli znakovi kategorije B koji imaju crveni obrub. Zatim se pod *Datoteke* dodaje datoteka tipa *.vse* s oznakama i nakon pritiska tipke *Dodaj,* pritišće se tipka *Traži.* Dva puta je potrebno kliknuti na ispis pod *Pretraga,* npr.: *Kategorija A: 344 objekata*. U novootvorenom prozoru potrebno pritisnuti tipku *Spremi.* Rezultat su *.bpm* slike s označenim znakovima koje se koriste prilikom evaluacije te datoteka tipa *.seq*.

6.4 Evaluacija detektora za trokutaste i okrugle znakove

Evaluacija detektora vrši se pomoću *preformance* usluge. Najvažniji parametri koje prima program *opencv_performance* su slijedeći[14]:

1. Klasifikator koji nastaje učenjem
2. Popis slika za evaluaciju koje su napravljene korištenjem programa *Marker* (*Marker* napravi datoteku tipa *.seq*, iz koje se uz pomoć programa napravljenog u jeziku *Python* generira datoteka tipa *.dat* kojeg prihvaća *OpenCV*). Kôd programa je prikazan na *Slici 6.1.*
3. Početni uvjeti – početna veličina odsječka na kojemu se traži znak
4. Parmetar koji pokazuje za koliko postotaka smije izračunati znak biti pomaknut od onog znaka koji je označen u *Markeru*, kako bi se on prihvatio kao stvaran znak („pravi pozitiv“)
5. Parametar koji pokazuje koliko puta smije izračunati znak biti veći ili manji od onog znaka koji je označen u *Markeru*, kako bi se prihvatio kao stvaran znak („pravi pozitiv“)
6. Parametar koji predstavlja koliko puta dio slike mora biti potvrđen kao pozitiv da bi se registrirao kao pozitivan ishod traženja

```
#!/usr/bin/python
import re
import sys
reLine = re.compile(r'^\[.*(A.bmp)\]:(.*)')
reSign = re.compile(r'.*@[^\d]*(\d+)[^\d]*(\d+)[^\d]*(\d+)[^\d]*(\d+)')
for line in sys.stdin.readlines():
    m = reLine.match(line)
    if m:s
        strSigns = m.group(2)
        vecSigns = strSigns.split('&')
        lineOut = '%s %d' % (m.group(1), len(vecSigns))
        for sign in vecSigns:
            m = reSign.match(sign)
            lineOut = lineNumber + ' %d %d %d' % (
             int(m.group(1)), int(m.group(2)),
             int(m.group(3)), int(m.group(4)))
        print lineOut
```

Slika 6.1: Skripta za konverziju datoteke tipa .seq u datoteku tipa .dat

Oblik datoteke tipa *.seq* je sljedeći: *kazaloDoBmpSlike/vrstaPrometnogZnaka/koordinate(gornja lijeva)/visina/širina*, npr.: `[D:\My Documents\5.semestar\Projekt\trokuti_test\A33_0034.bmp]: A04@(x=577,y=218,w=30,h=30)&A33@(x=576,y=248,w=32,h=31).` Na ovom se primjeru vidi da na istoj slici mogu biti označena dva znaka.

Datoteka tipa *.dat* datoteka popis slika u obliku: *imeSlike brojZnakovaNaSlici koordinataXGornjegLijevogKuta koordinataYGornjegLijevogKuta širinaZnaka visinaZnaka*, npr.: `B31_0188.bmp 1 603 211 49 56.`

Slijedi opis načina rada programa `opencv_performance.exe`.
Provjerava se gornji lijevi dio slike na kojoj se traži znak, početne veličine zadane parametrima. Veličina početnog okvira mora biti veća ili jednaka veličini pozitivnih slika na kojima je naučen klasifikator. U ovom slučaju to je 24x24 piksela.

Uz pomoć parametara naučenoga klasifikatora provjerava se nalazi li se u nekom okviru znak ili ne. Okvir se nakon provjere pomiče jedan piksel udesno pa na kraju retka jedan piksel prema dolje, itd. sve dok se ne pretraži čitava slika. Nakon što se pretraži čitava slika, veličina prozora se pomnoži veličinom parametra *sf* (eng. *scale factor*) i pretraga počinje iznova.

Naposljetku se izračunaju koordinate znaka na temelju koordinata grupe prozora u kojima je znak pronađen. Izgenerira se *.bmp* slika i na njoj crvenim okvirom označi gdje je izračunato da se znak nalazi.

Rezultati evaluacije koji se mogu vidjeti u *Tablici 6.2*. Inzistiranje na vrlo velikom *mhr*-u(99.9%) rezultiralo je dobrim postotkom pronalaska znakova, no zbog iznimno velikog broja pronalaska znakova tamo gdje ih nema, pokazao se kao ne tako dobar izbor. Drugi nedostatak ovakvog parametra je dugo vrijeme učenja.

Smanjenje *mhr*-a na 99% dalo je sljedeću promjenu u rezultatima: što je *mfa* bio veći to su rezultati bili sličniji, a što je bio manji to su rezultati bili lošiji, gdje pod „lošije“ rezultate smatramo povećani broj lažnih pozitiva uz manji postotak detekcije.

Primjer poziva *performance* programa:

opencv_performance.exe -data klasifikator1 -info trokuti.dat

Naravno, evaluacija je bolja ili lošija, ovisno o parametrima koji se zadaju. Evaluira se na raznim klasifikatorima nastalim prilikom učenja. Parametri se mijenjaju kako bi se postigli što bolji rezultati s određenim klasifikatorima.

Inzistiranje na vrlo velikom *mhr*-u(99.9%) rezultiralo je dobrim postotkom pronalaska znakova, no zbog iznimno velikog broja pronalaska znakova tamo gdje ih nema, pokazao se kao ne tako dobar izbor. Drugi nedostatak ovakvog parametra je dugo vrijeme učenja.

Smanjenje *mhr*-a na 99% dalo je sljedeću promjenu u rezultatima: to je *mfa* bio veći to su rezultati bili sličniji, a što je bio manji to su rezultati bili lošiji, gdje pod „lošije“ rezultate smatramo povećani broj lažnih pozitiva uz manji postotak detekcije.

Tablica 6.2: Rezultati evaluacije

n	nneg	nstage zadan	nstage napravljen	mhr	mfa	broj slika pozadina	pronađen i pozitivi	promašeni	lažni pozitivi	% pronađenih	n
1	2000	14	12	0,995	0,4	230	931	169	763	84,64	1
2	2000	14	12	0,999	0,5	230	1060	40	10654	96,36	2
3	10000	14	13	0,995	0,5	711	1073	27	3398	97,55	3
4	10000	18	13	0,995	0,6	711	1072	28	14414	97,45	4
5	10000	14	13	0,995	0,4	711	1055	45	866	95,91	5
6	10000	14	13	0,99	0,4	711	1020	80	914	92,73	6
7	10000	18	13	0,99	0,6	711	1070	30	15708	97,27	7
8	10000	14	13	0,99	0,5	711	1055	45	6671	95,91	8
9	?	?	30	?	?	?	883	217	488	80,27	9

Usporedbom klasifikatora 1 i 5 dolazi se do sljedećeg zaključka: povećanje broja slika pozadina odnosno „neznakova“ s 230 na 711 i povećanja iz njih izvučenih primjera „neznakova“ s 2000 na 10000 značajno je popravilo postotak pronalaska znakova. Broj lažnih pozitiva malo je porastao.

Zbog toga što program izračunava pozicije znakova na temelju grupe detekcija na mjestu na kojem se pretpostavlja da se nalazi znak, ponekad je bolje imati veći *scale factor*. Do ovoga se zaključka došlo primjenom *scale factor*-a, kao i do zaključka da klasifikatori koji inače pronalaze više lažnih pozitiva povećanjem *scale factor*-a mogu bolje pogoditi znak, što se vidi u *Tablici 6.3*. Kako se taj parametar smanjuje, pretraživanje dulje traje, dok za vrijednost 1,02 nema dovoljno memorije da se uopće pokrene evaluacija.

Tablica 6.3: Rezultati evaluacije klasifikatora tri i šest

n	sf	pronađeni pozitivi	promašeni	lažni pozitivi	%pronađenih
3	1,35	1069	31	6900	97,18
3	1,31	1079	21	8420	98,09
3	1,3	1079	21	7278	98,09
3	1,28	1073	27	8329	97,55
3	1,2	1073	27	3398	97,55
3	1,1	1038	62	17281	94,36
3	1,05	1001	99	28497	91,00
6	1,3	1000	100	750	90,91
6	1,2	1020	80	914	92,73
6	1,15	1053	47	1419	95,73
6	1,1	1066	34	1581	96,91
6	1,09	1067	33	1519	97,00
6	1,08	1063	37	1741	96,64

Promjenom parametara u *performance*-u možemo namjestiti koliko smije biti pomaknut okvir prepoznatoga znaka u odnosu na ono što smo označili kao znak. Ukoliko je cilj da se detektirani znak kasnije prepozna, vrlo je bitno da je znak vrlo dobro smješten u okvir. U tom slučaju okvir koji je ispustio npr. jedan ugao znaka ne može se smatrati dobro detektiranim znakom. Ukoliko je bitno samo postoji li na slici negdje znak, parametre veličine i pomaknutosti možemo postaviti puno slobodnije.

Znakovi traženi ovim klasifikatorima su pronađeni u barem dva okvira od četri, koliko puta je bilo označen isit znak. Znak najčešće nije prepoznat na onoj slici na kojoj je najudaljeniji od kamere. Često su znakovi bili proglašeni nedetektiranima zbog pogrešog grupiranja te je rezultantni okvir bio premali ili previše pomaknut u neku od strana. Primjećeno je da je rezultantni okvir znaka na desnom rubu slike ponekad pomaknut u lijevo u odnosu na rub ekrana. S desne strane okvir može doći do ruba, ali ne i preko njega.

Lažne detekcije, odnosno prepoznavanje znakova na mjestima na kojima se znakovi ne nalaze, događale su se iz više razloga. Na kolniku, koji se sive boje, znakovi su prepoznati zbog prirode algoritma koji detektira čak i minimalne razlike između

tamnog i svjetlog. Poneki krovovi prepoznaju se kao trokutasti znakovi. Najčešće krive detekcije se nalaze tamo gdje se ljudsko oko ne bi zabunilo, a to je u zelenilu krošanja.

Važno je napomenuti kako je snimka za evaluaciju nastala na potpuno različitom mjestu u različito vrijeme od snimaka koje su upotrebljavane za učenje. Klasifikator 9, koji je pokazao najmanji postotak detekcije preuzeli smo gotov. Klasifikator koji je naučen na jednom dijelu snimke i evaluiran na drugom dijelu te iste snimke, pokazao je vrlo dobar postotak detekcije, preko 96%. Kada je za evaluaciju klasifikatora korišten drugi video, postotak detekcije je skromnih 80%.

7. Literatura

1. Pattern Recognition – OpenCV 2.0 C Reference; 11. prosinca 2009.; *Pattern Recognition – Object Detection*; *http://opencv.willowgarage.com/documentation/pattern_recognition.html?highlight=cvhaar#cvHaarDetectObjects*; 16. studeni 2009.
2. Viola P., Jones M.; Robust Real-time Object Detection; 13. srpnja 2001.; *Robust Real-time Object Detection; http://www.google.com/url?sa=D&q=http://research.microsoft.com/en-us/um/people/viola/pubs/detect/violajones_ijcv.pdf&usg=AFQjCNGFwYFO7R61Soi36xifNGBzZC7iFA*; 21. studeni 2009.
3. OpenMP - Wikipedia, the free encyclopedia; 20.11.2009.; *OpenMP http://en.wikipedia.org/wiki/OpenMP*; 23.11.2009.;
4. Babić, T., Programska implementacija pronalaženja objekata kaskadom boostanih Haarovih klasifikatora, Završni rad br. 806, Fakultet elektrotehnike i računarstva, Zagreb, 2009.
5. Čuljak, M.: Primjena strojno naućenih klasifikatora u računalnom vidu, Seminar, Fakultet elektrotehnike i računarstva, Zagreb, 2009.
6. D. Using the schedule Clause; *Using the schedule Clause*; *http://msdn.microsoft.com/en-us/library/9w1x7937.aspx*; 15.12.2009.
7. Suess M.; Parallel Programming Fun with Loop Carried Dependencies » Thinking Parallel; 2. svibnja 2007.; *Parallel Programming Fun with Loop Carried Dependencies*; *http://www.thinkingparallel.com/2007/05/02/parallel-programming-fun-with-loop-carried-dependencies/*; 16.12.2009.
8. Blaise B.; OpenMP; 30. siječnja 2009.; *OpenMP; https://computing.llnl.gov/tutorials/openMP/*; 16.12.2009.
9. Streaming SIMD Extension – Wikipedia, the free encyclopedia; 30.12.2009.; *Streaming SIMD Extension*; *http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions*; 02.01.2010.

10. x86 – Wikipedia, the free encyclopedia; 21. 12. 2009.; *x86;* *http://en.wikipedia.org/wiki/X86*, 02.01.2010.
11. SIMD - Wikipedia, the free encyclopedia; 12. 12. 2009; *SIMD*; *http://en.wikipedia.org/wiki/SIMD*, 02.01.2010.
12. Gary Bradski Adrian Kaehler. *Learning OpenCV.* Media, Sebastopol, 1. izdanje, 2008.
13. Fakultet Elektrotehnike i Računarstva. *MASTIF Marker – upute za korištenje,* 1. izdanje, 2009.
14. Naotoshi Seo; Tutorial: OpenCV haartraining (Rapid Object Detection With A Cascade of Boosted Classifiers Based on Haar-like Features); 14.08.2008; *Tutorial: OpenCV haartraining;* *http://note.sonots.com/SciSoftware/haartraining.html#mda65705*
15. OpenCV - Wikipedia, the free encyclopedia; 06.12.2008.; *OpenCV;* *http://en.wikipedia.org/wiki/OpenCV*
16. Živković, O., Optimiranje algoritama obrade slike vektorskim instrukcijama, Završni rad br. 756, Fakultet elektrotehnike i računarstva, Zagreb, 2009.
17. Kalafatić, Z., Šegvić, S., Dinamička analiza 3D scena, Prezentacija, Fakultet elektrotehnike i računarstva, Zagreb; *http://www.zemris.fer.hr/~ssegvic/pubs/das1.pdf*
18. D. Bučar, A. Bulović, S. Grčičić, J. Hucaljuk, B. Kovačić, P. Palašek, B. Popović, A. Sambol: Integriranje dodanih mogućnosti u programski sustav Marker, Projekt, Fakultet elektrotehnike i računarstva, Zagreb 2010.