SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI ZADATAK br. 1540

OPTIMIZACIJA IMPLEMENTACIJE PROJEKCIJSKOG RAVNINSKOG PRESLIKAVANJA

Petra Bosilj

Zagreb, svibanj 2010.

Autorica rada zahvaljuje se svim kolegama koji su svojim komentarima učinili ovaj rad kvalitetnijim, a posebice kolegi Davoru Cihlaru na mnogim sugestijama pri prijevodu i nesebičnu pomoć pri uređivanju rada.

Sadržaj

1. Uvod

Cilj ovog rada bio je napraviti brzu i optimiziranu implementaciju projekcijskog ravninskog preslikavanja uz ograničenje na vrstu ulaznih slika. Ostvarena implementacija radi samo na sivim slikama, gdje se siva razina pojedinog slikovnog elementa kreće u rasponu od 0 do 255, odnosno veličina jednog piksela je 8 bita. Zadatku se pristupilo izradom izravne implementacije algoritma i mjerenjem vremena njezina izvršavanja. Tako izmjerena vremena korištena su kao referentne brzine izvođenja pri daljnjoj analizi ubrzanja algoritma.

Konačno ubrzanje se postiglo isprobavanjem raznih koncepata i tehnika te međusobnom kombinacijom promjena koje su smanjile vrijeme izvođenja.

Projekcijsko ravninsko preslikavanje, odnosno perspektivna transformacija, ima mnogo primjena na području računalnog vida. Neke od njih uključuju uklanjanje izobličenja nastalih zbog perspektivne slike (*engl. perspective imaging*, npr. fotografija snimljena pod nekim kutem u odnosu na objekt) koja mijenja odnos oblika i pravaca, simuliranje rotacije kamere, iscrtavanje sjene osvjetljenog objekta [1] i dobivanje zrcalnog odraza slike.

Implementacija okviru ovog rada razvijena je u ljusci *cvsh2* (Computer Vision Shell) i trenutno se koristi u implementaciji pronalaženja, prepoznavanja i praćenja razdjelne linije na prometnici [13]. Da bi se postpak praćenja linije moglo uspješno provesti, potrebno je iz ulazne snimke snimljene iz perspektive vozača inverznom perspektivnom transformacijom dobiti snimke iz ptičje perspektive.

Postupak i svojstva projekcijskog ravninskog preslikavanja kao i posebni slučaj afine transformacije objašnejna su u odjeljku 2, nakon čega u odjeljku 3 slijedi kratki teoretski pregled koncepata i tehnika korištenih za ubrzanje osnovne implementacije. U odjeljku 4 sažeto je opisano razvojno okruženje kao i sve potrebne postavke za postizanje najbolje performanse. Primjena koncepata i tehnika objašnjenih u odjeljku 3 na osnovnu

implementaciju algoritma perspektivne transformacije prikazana je u odjeljku 5, dok se u idućem odjeljku, odjeljku 6 može vidjeti pregled postignutih rezultata. Na kraju rada izveden je zaključak (odjeljak 7), navedena korištena literatura (odjeljak 8) te sažetak najbitnijih točaka rada (odjeljak 9).

2. Projekcijsko ravninsko preslikavanje

2.1. Prikaz točaka i pravaca ravnine homogenom notacijom

Za razumijevanje postupka projekcijskog ravninskog preslikavanja, potrebno je biti upoznat sa homogenom reprezentacijom točaka i pravaca.

Svaka je točka u ravnini jednoznačno određena uređenim parom koordinata (x, y), zbog čega je uobičajeno poistovjetiti ravninu sa $\mathbb{R}^2$. Točku u ravnini tako možemo predstaviti sa $\mathbf{x} = (x, y)^\top$, pri čemu obje strane ove jednadžbe predstavljaju stupčani vektor.

Svaki pravac u ravnini možemo prikazati jednadžbom $ax + by + c = 0$. Različite pravce ravnine dobivamo mjenjajući parametre a, b i c. Zbog toga se pravci u homogenoj notaciji prirodno mogu predstaviti kao stupčani vektori $(a, b, c)^\top$. Može se primjetiti da ovaj prikaz pravca nije jednoznačan, pošto pravci $ax + by + c = 0$ i $(ka)x + (kb)y + (kc) = 0$ predstavljaju isti pravac za svaku konstantu k različitu od nule [1].

Kako bi se došlo do homogenog zapisa točke u ravnini, razmatra se sljedeće:

- točka $\mathbf{x} = (x, y)^\top$ leži na pravcu $\mathbf{l} = (a, b, c)^\top$ ako i samo ako $ax + by + c = 0$;
- ovo se može zapisati pomoću skalarnog produkta vektora koji prikazuje točku i homogenog prikaza pravca kao $(x, y, 1)(a, b, c)^\top = (x, y, 1)\mathbf{l} = 0$;
- vrijedi $(kx, ky, k)\mathbf{l} = 0$ za bilo koju konstantu k različitu od nule i pravac $\mathbf{l}$ ako i samo ako vrijedi $(x, y, 1)\mathbf{l} = 0$;
- prirodno je, zbog toga, skup vektora $(kx, ky, k)^\top$ smatrati različitim homogenim prikazima točke $(x, y)^\top$ iz $\mathbb{R}^2$ za različite vrijednosti parametra k.

Iz ovoga se može vidjeti da će točki sa homogenim koordinatama $\mathbf{x} = (x_1, x_2, x_3)^\top$ odgovarati točka $(x_1/x_3, x_2/x_3)^\top$ u $\mathbb{R}^2$, gdje x_3 predstavlja homogenu koordinatu točke $\mathbf{x}$.

Projekcijski prostor $\mathbb{P}^2$ može se opisati kao skup zraka u prostoru $\mathbb{R}^3$. Povuče li se pravac kroz ishodište prostora $\mathbb{R}^3$, točke koje se nalaze na tom pravcu mogu se predstaviti vektorima $k(x_1, x_2, x_3)$ koji se u projekijskom prostoru $\mathbb{P}^2$ preslikavaju u istu točku [1].

U nastavku su navedena još neka bitna svojstva pravaca i točaka prikazanih homogenom notacijom, odnosno u projekcijskom prostoru $\mathbb{P}^2$, čija se detaljna pojašnjenja mogu naći u [1]:

- točka **x** leži na pravcu **l** ako i samo ako vrijedi $\mathbf{x}^\top\mathbf{l} = 0$,
- da bi se jednoznačno odredili točka i pravac, potrebne su dvije vrijednosti (*x*- i *y*-koordinate točke odnosno dva nezavisna omjera, npr *a : b* i *b : c* kod pravca), zbog čega se kaže da točka i pravac imaju *dva stupnja slobode,*
- sjecište dva pravca **l** = (a, b, c) i **l'** = (a', b', c') može se izračunati kao $\mathbf{x} = \mathbf{l} \times \mathbf{l'}$, gdje $\mathbf{l} \times \mathbf{l'}$ predstavlja vektorski produkt dvaju vektora koji označavaju pravce. Bitno je napomenuti da uvrštavanjem vektora koji označavaju dva paralelna pravca dobiva točka **x** sa homogenom koordinatom jednakom 0, i takve točke nazivamo *idealnim točkama* ili točkama u beskonačnosti,
- pravac **l** koji prolazi točkama **x** i **x'** može se odrediti kao $\mathbf{l} = \mathbf{x} \times \mathbf{x'}$.

2.2. Projekcijske transformacije

Prema definiciji [1], planarna projekcijska transformacija, ili *homografija*, je preslikavanje *h* iz $\mathbb{P}^2$ u $\mathbb{P}^2$ koje ima svoj inverz, takvo da tri točke **x**1, **x**2, i **x**3 leže na istom pravcu ako i samo ako *h(***x**1*), h(***x**2*)* i *h(***x**3*)* leže na istom pravcu.

Homografija je linearna transformacija nad točkama u homogenom zapisu koja se može predstaviti nesingularnom matricom dimenzija 3×3:

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} \sim \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad (2.1)$$

ili kraće zapisano:

$$\boldsymbol{x}' \sim H\,\boldsymbol{x} \quad (2.2) \quad .$$

Bitno je primjetiti da skaliranje matrice H proizvoljnim faktorom skaliranja različitim od nule ne mijenja projekcijsku transformaciju. Upravo zbog tog svojstva matricu H nazivamo *homogenom matricom,* jer je, baš kao i kod homogene reprezentacije točke, bitan samo omjer elemenata matrice. Kako postoji osam međusobno nezavisnih omjera u matrici H, projekcijska transformacija ima osam stupnjeva slobode.

2.2.1. Afine transformacije

Afina transformacija u matričnom prikazu [1] zapisuje se kao:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \sim \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (2.3) \quad ,$$

ili kraće zapisano:

$$\boldsymbol{x}' \sim H_A \boldsymbol{x} \sim \begin{bmatrix} A & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \boldsymbol{x} \quad (2.4) \quad .$$

Iz zapisa (2.3) može se vidjeti da planarna afina transformacija ima šest stupnjeva slobode koji odgovaraju promjenljivim elementima matrice. Zapis (2.4) ukazuje na to da se afina transformacija sastoji od dva dijela: matrica *A* obavlja rotaciju i smik nad zadanom točkom, nakon čega se nad točkom vrši translacija za ***t***.

Nakon afine transformacije, omjeri duljina kao i kutevi između pravaca nisu sačuvani, no tri bitna invarijantna svojstva afine transformacije [1] su:

i. **Paralelni pravci**. Sjecište paralelnih pravaca može se prikazati kao $(x_1, x_2, 0)^\top$ i predstavlja idealnu točku, odnosno točku u beskonačnosti. Nakon transformacije, paralelni pravci preslikavaju se u točke čije je sjecište i dalje u beskonačnosti tj. u paralelne pravce.

ii. **Omjer duljina segmenata paralelnih pravaca**. Kako faktor skaliranja dužine segmenta pravca ovisi samo o kutu između pravca i smjera skaliranja, skaliranje je zajedničko svim pravcima koji imaju isti smjer. Zbog toga se

skaliranje poništava kod omjera duljina segmenata paralelnih pravaca.

iii. **Omjer površina**. Može se pokazati [1] da se sve površine skaliraju istim faktorom, preciznije $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{21} \end{vmatrix}$. Isto kao i kod segmenata paralelnih pravaca, zajednički faktor se poništava kod računanja omjera površina.

2.3. Interpolacija

Područje matematike, numerička analiza, definira interpolaciju kao metodu izračunavanja novih vrijednosti funkcija iz poznatog diskretnog skupa zadanih vrijednosti. Diskretni skup vrijednosti najčešće se dobiva uzorkovanjem ili eksperimentalno te se interpolacija koristi kao jedna od metoda za konstruiranje funkcije koja što bolje prati zadane vrijednosti [2].

U okviru ovog rada, interpolacija je korištena za izračunavanje vrijednosti piksela sa realnim koordinatama koji u stvarnosti ne postoje iz poznatih vrijednosti susjednih piksela slike.

2.3.1. Linearna interpolacija

Linearna interpolacija jednostavan je oblik interpolacije namjenjen za interpolaciju točaka sa jednom prostornom dimenzijom, tj. za procjenu vrijednosti funkcije ovisne o samo jednom parametru [3]. Uz poznate vrijednosti funkcije *f(x)* u dvije točke, x_0 i x_1, pri čemu je $f(x_0)=y_0$ i $f(x_1)=y_1$, procjena vrijednosti u proizvoljnoj točki *x* može se izračunati po formuli:

$$p(x)=f(x_0)+\frac{f(x_1)-f(x_0)}{x_1-x_0}(x-x_0) \quad (2.5) \ ,$$

gdje *p(x)* označava polinom linearne interpolacije.

Iz formule (2.5) može se vidjeti da ukoliko su nam poznate vrijednosti dviju točaka (x_0, y_0) i (x_1, y_1), rezultat linearne interpolacije nad tim točkama je dužina između te dvije točke.

Linearna interpolacija nad skupom točaka definirana je kao konkatenacija linearnih interpolacija između svakog para točaka [3]. Rezultat linearne interpolacije u tom je slučaju neprekinuta krivulja sa prekidima u derivaciji u zadanim točkama.

2.3.2. Bilinearna interpolacija

Bilinearna interpolacija je proširenje linearne interpolacije za interpoliranje vrijednosti funkcija dvije varijable [4]. Bilinearna interpolacija provodi se tako da se prvo provede linearna interpolacija po jednoj koordinati, a zatim po drugoj. Iako je svaka od interpolacija po koordinatama linearna, bilinearna interpolacija kao cjelina nije linearna već kvadratna pošto je nastala kao produkt dviju linearnih funkcija.

Formulu za linearnu interpolaciju može se također napisati kao:

$$f(x) \approx \frac{x_1 - x}{x_1 - x_0} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) \quad (2.6)$$

Da bi se dobila formula za bilinearnu interpolaciju [4] u točki $P = (x, y)$ uz poznate vrijednosti funkcije f u točkama $Q_{00} = (x_0, y_0)$, $Q_{01} = (x_0, y_1)$, $Q_{10} = (x_1, y_0)$, $Q_{11} = (x_1, y_1)$, prvo se provodi linearna interpolacija po x-osi:

$$f(R_0) \approx \frac{x_1 - x}{x_1 - x_0} f(Q_{00}) + \frac{x - x_0}{x_1 - x_0} f(Q_{10}) \quad (2.7) \ ,$$

gdje je $R_0 = (x, y_0)$,

$$f(R_1) \approx \frac{x_1 - x}{x_1 - x_0} f(Q_{01}) + \frac{x - x_0}{x_1 - x_0} f(Q_{11}) \quad (2.8) \ ,$$

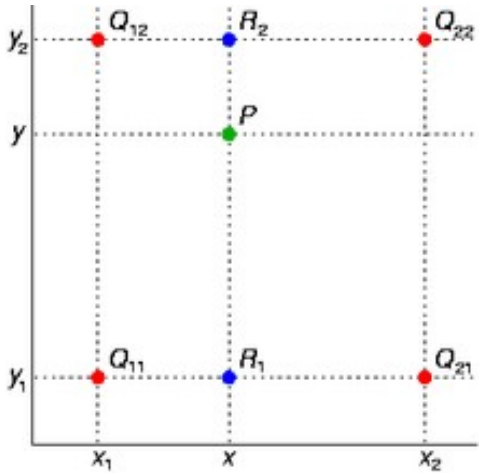
gdje je $R_1 = (x, y_1)$.

Nakon toga provodi se linearna interpolacija po y-osi:

$$f(P) \approx \frac{y_1 - y}{y_1 - y_0} f(R_0) + \frac{y - y_0}{y_1 - y_0} f(R_1) \quad (2.9) \ .$$

Uvrštavanjem vrijednosti $f(R_0)$ i $f(R_1)$ u gornju formulu, dobivamo željenu procjenu za točku P:

$$f(x,y)\approx\frac{f(Q_{00})}{(x_1-x_0)(y_1-y_0)}(x_1-x)(y_1-y)+\frac{f(Q_{10})}{(x_1-x_0)(y_1-y_0)}(x-x_0)(y_1-y)+\frac{f(Q_{01})}{(x_1-x_0)(y_1-y_0)}(x_1-x)(y-y_0)+\frac{f(Q_{11})}{(x_1-x_0)(y_1-y_0)}(x-x_0)(y-y_0) \quad (2.10).$$



Slika 2.1: Bilinearna interpolacija u točki P

Točke nad kojima se provodi postupak bilinearne interpolacije u formulama (2.7)-(2.10) mogu se vidjeti na *Slici 2.1*.

U okviru ovog rada, bilinearna interpolacija korištena je isključivo na susjednim pikselima, odnosno na takvim točkama za koje uvijek vrijedi $x_1 = x_0 + 1$ i $y_1 = y_0 + 1$, a točka čija se vrijednost procjenjivala uvjek je bila u intervalu $(x, y) \in [(x_0, y_0), (x_1, y_1)\rangle$. Zbog toga, bilo je moguće koristiti pojednostavljenu formulu za bilinearnu transformaciju:

$$f(x,y)\approx f(Q_{00})(1-\{x\})(1-\{y\})+f(Q_{10})\{x\}(1-\{y\})+f(Q_{01})(1-\{x\})\{y\}+f(Q_{11})\{x\}\{y\} \quad (2.11),$$

pri čemu je oznaka {a} definirana kao $\{a\}=a-\lfloor a\rfloor$.

3. Pregled koncepata korištenih za oblikovanje ubrzane izvedbe

Kako bi se postiglo ubrzanje osnovne implementacije projekcijskog ravninskog preslikavanja, isprobani su različiti koncepti i tehnike koji će biti objašnjeni u nastavku ovog poglavlja.

3.1. Vektorske instrukcije

SIMD računala prema Flynnovoj klasifikaciji arhitektura računala označavaju razred paralelnih računala. Kratica dolazi od engleskog izraza *Single instruction multiple data*. Razvojem računalnih igara koje simuliraju stvarni vremenski tok, proizvođači mikroprocesora uvode podršku za SIMD instrukcije na osobna računala. Prvi široko upotrebljavan SIMD instrukcijski skup bilo je Intelovo proširenje arhitekture x86 poznato pod nazivom MMX, instrukcijski skup temeljen na mikroprocesoru Intel 8086. Nove SIMD arhitekture mogu se smatrati kratkim vektorskim arhitekturama (*engl. short-vector arhitectures*) jer su orijentirane na rad sa vektorima od dva ili četiri elementa. Moderna računala često su višeprocesorska MIMD računala (*engl. Multiple instruction, multiple data*) gdje svaki procesor može izvršavati kratke vektorske SIMD instrucije [6].

U okviru ovog rada razmatrano je korištenje SIMD proširenja SSE2, koje je prvi puta predstavio Intel 2001. godine u prvoj verziji procesora Pentium 4. SSE2 nastao je kao proširenje na instrukcijski skup SSE, napravljen da potpuno istisne i zamjeni proširenje MMX[7]. Neke od tipičnih SSE2 instrukcija su `ADDPD`, `DIVPD`, `MULPD` za paralelno zbrajanje, dijeljenje i množenje četiri realna broja jednostruke preciznosti (*engl. float)*, kao i one malo "naprednije" poput `MINPD` i `SQRTPD` za određivanje najmanjeg realnog broja i korjenovanje vektora relanih brojeva jednostruke preciznosti.

Jedan od načina korištenja proširenja SSE2 je korištenje intrinzika (*engl. Intrinsic*). Intrinzici su funkcije poznate prevodiocu (pa su stoga i ovisne o prevodiocu) koje se

prevode direktno u najčešće jednu vektorsku instrukciju. Intrinzici, ili intrinzične funkcije su učinkovitije od običnih poziva funkcija, zbog toga što nije potrebno povezivanje pri pozivu funkcije – prevodioc direktno mapira intrinzičnu funkciju u niz asemblerskih istrukcija [8]. Intrinzici korišteni u okviru rada pripadaju Microsoft Visual C++ prevodiocu.

Tablica 3.1: primjer intrinzične funkcije

mnemonik	objašnjenje i povratna vrijednost

`__m128 _mm_set_ps (float z, float y, float x, float w);`	Postavlja četiri realne komponente jednostruke preciznosti u varijablu tipa `__mm128`, koja odgovara jednom XMM[0-7] registru, na četiri vrijednosti zadane parametrima:
	`r0 := w`
	`r1 := x`
	`r2 := y`
	`r3 := z`

Primjer intrinzične funkcije kojom se podatci pripremaju za obradu pomoću drugih SSE intrinzika može se vidjeti u *Tablici 3.1.* Varijable tipa `__mm128` koje odgovaraju jednom XMM registru na ovaj se način pripremaju za daljnju obradu drugim intrinzicima. Nad njima se nakon postavljanja vrijednosti mogu izvoditi aritmetičke i logičke operacije koje se nakon obrade ponovno spremaju u varijable primitivnog tipa.

3.2. OpenMP

Kao što je već spomenuto u odjeljku 3.1, današnja računala često su višeprocesorska, što čini zanimljivom mogućnost uvođenja višedretvenosti u implementaciju. OpenMP (*Open Multi-Processing*) je aplikacijsko programsko sučelje (*engl. application programming interface*, u daljnjem tekstu *API*) koje podržava višeprocesorsko programiranje sa dijeljenom memorijom u raznim programskim jezicima, koji uključuju C i C++, na platformama Microsoft Windows i Unix [10]. Sastoji se od skupa direktiva prevodiocu, biblioteka i varijabli okoline koje omogućavaju paralelizaciju dijelova implementacije. Da bi se omogućio rad sa OpenMP API-jem, potrebno je uključiti zaglavlje *"omp.h".* OpenMP API omogućuje kako paralelizam na razini podataka *(engl. data parallelism*), tako i paralelizam na razini zadataka (*engl. task parallelism)* [9]. Pod pojmom *paralelizam na razini podataka* podrazumjeva se

paralelizacija izračuna, odnosno raspodjela podataka na razne čvorove za paralelni izračun – procesore [11], dok se pod pojmom *paralelizam na razini zadataka* podrazumjeva paralelizacija programskog koda na način da svaki procesor izvodi svoju dretvu nad (tipično) različitim podatcima, pri čemu dretve mogu međusobno komunicirati razmjenjujući podatke [12].

Tablica 3.2: Primjer paralelizacije programskog koda OpenMP API-jem

primjer programskog koda	objašnjenje

`int main(int argc, char *argv[]) {`
`  const int N = 100000;`
`  int i, a[N];`
 	
`  #pragma omp parallel for`
`  for (i = 0; i < N; i++)`
`  a[i] = 2 * i;`
 	
`  return 0;`
`}`
	• izvršavanje for petlje za inicijalizaciju velikog niza izvršava se paralelno, u više dretvi
	• direktive prevodiocu u jezicima C/C++ nazivaju se *pragmama*, a one specifične za OpenMP imaju oblik:
	`#pragma omp <rest of pragma>`

U *Tablici 3.2* moguće je vidjeti primjer paralelizma na razini podataka. Na pragmu `#pragma omp parallel for` mogli su se još dodati atribut `num_threads(n)` kojim se određuje broj stvorenih dretvi i `schedule(type)` kojim se bira način određivanja rasporeda poslova (*engl. scheduling*) između *static, dynamic* i *guided*[1] načina [9]. Postoje još i atributi kojima se sprečava "stanje utrke" (*engl. race conditions*). Stanje utrke je izvorno dobilo ime po stanju u računalnom sustavu u kojem se podaci šire i putuju kroz logičke sklopove tako brzo da ne mogu biti upravljani taktnim signalima, no u ovom kontekstu označava stanje u kojem više dretvi pokušava istovremeno pristupiti istim podatcima i pa se može reći da se dretve "natječu" za pristup. Atributi ovog tipa uključuju: atribute za djeljenje podataka, atribute za sinkronizaciju, za kontrolu uvjetnih naredbi, inicijalizacije varijabli u paralelnim blokovima, kopiranje podataka, i druge [9].

1 *static* način rasporeda poslova određuje iteracije koje će svaka dretva izvršiti prije izvršavanja iteracija petlje, *dynamic* način rasporeda poslova dodjeljuje određenu količinu iteracija malom broju dretvi, dok se kod *guided* načina rasporeda poslova velik dio sljednih iteracija dodjeljuje svakoj dretvi dinamički.

3.3. Blocking

Engleski izraz *blocking* označava tehniku za poboljšanje lokalnosti memorijskih pristupa, temeljenoj na obrađivanju podmatrica glavne matrice pri izvedbi matričnog algoritma.

Memorijske strukture u današnjim računalnim sustavima mogu se na temelju kapaciteta i brzine memorije svrstati u hijerarhijsku strukturu [14]:

- **Lokalna memorija** ili **registri**. Brzina istog reda veličine kao i sklopovi procesora, najčešće 8 do 64 registra, no moguće čak i do nekoliko stotina.
- **Priručna *(engl. chache)* memorija**. Ne mora biti sporija od lokalne memorije, većeg kapaciteta najčešće od 4 K bajta do 64 K bajta.
- **Glavna, radna** ili **primarna memorija**. Reda veličine *M bajta* i sporija u odnosu na priručnu memoriju.
- **Sekundarna memorija**. Vrlo velikog kapaciteta (red veličine *G bajta*) i relativno spora u odnosu na primarnu memoriju (obično diskovna memorija).

Pravilnom uporabom ovakve memorijske hijerarhije, omogućava se primarnoj memoriji da se prividno pojavljuje kao memorija kapaciteta sekundarne memorije, a brzine jednake ili gotovo jednake najbržoj memoriji u memorijskoj hijerarhiji: centralnoj procesorskoj jedinici [14].

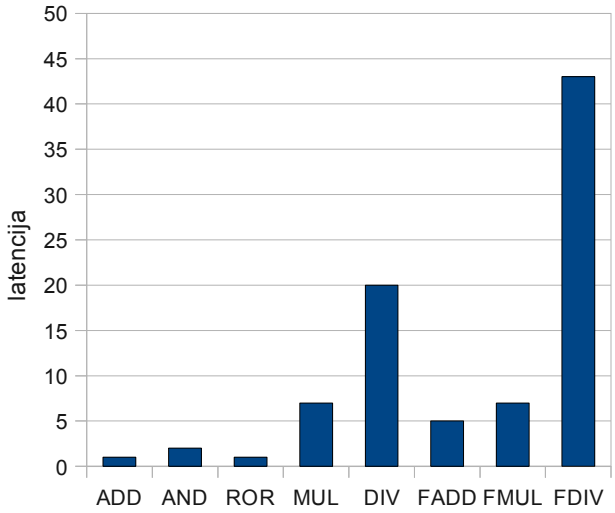
Tehnikom *blockinga* pokušavaju se maksimalno iskoristiti svojstva *priručne memorije* pažljivim odabirom redosljeda pristupa podataka u glavnoj memoriji. Tipično je priručna memorija 5 ili više puta brža od glavne memorije. U nju se pohranjuju tekući aktivni segmenti programa i podataka [14]. *Lokalnost pristupa* ili princip lokalnosti odnosi se na pojavu učestalosti pristupa istim ili bliskim vrijednostima ili memorijske lokacijama. Dva osnovna tipa lokalnosti pristupa su *vremenska lokalnost* i *prostorna lokalnost*. Vremenska lokalnost odnosi se na ponovno upotrebljavanje određenog računalnog resursa u relativno kratkim vremenskim periodima, dok se prostorna lokalnost odnosi na upotrebu podatkovnih elemenata koji se nalaze na bliskim memorijskim lokacijama. Posebna vrsta prostorne lokalnosti, *sekvencijalna lokalnost* javlja se kada su podatkovni

elementi uređeni i pristupa im se linearno, poput čitanja elemenata jednodimenzionalnog polja podataka [15].

Bolje iskorištavanje priručne memorije postiže se smanjivanjem broja *promašaja*, odnosno, broja pristupa memorijskoj lokaciji čija kopija nije pohranjena u priručnoj memoriji (memoriji velike brzine), pa se stoga mora dohvaćati iz glavne memorije (koja je male brzine). Tehnika *blocking* to postiže iskorištavanjem svojstva vremenske lokalnosti pristupa podatcima [16]. Osnovna ideja je da se matrica, umjesto redak po redak (ili stupac po stupac), podijeli na manje cijeline, podmatrice koje se zatim sekvencijalno obrađuju. Na taj način se pokušava postići da se za čitavo vrijeme obrade podmatrice odnoslo *bloka* podataka koji se obrađuje svi potrebni podatci cijelo vrijeme nalaze u priručnoj memoriji. Detaljnije objašnjenje i primjer korištenja *blocking* tehnike može se naći u [16]

3.4. Strength reduction

Strength reduction je optimizacija prevodioca pri kojoj se skupe operacije zamjenjuju ne zamjenjuju sa direktnim asemblerskim instrukcijama već sa ekvivalentnim, ali manje složenim instrukcijama.



Grafikon 3.1: Usporedba brzine pojedinih asemblerskih iz Pentium 4 porodice procesora

Kao što se može vidjeti iz *Grafikona 3.1,* neke se matematičke operacije prevode u spore asemblerske instrukcije. U usporedbi s jednostavnim aritmetičkim operacijama (`ADD`, `FADD`), logičkim operacijama (`AND`) i operacijama za manipulaciju bitovima (`ROR`), sporim operacijama možemo smatrati operacije množenja (`MUL`, `FMUL`) i dijeljenja (`DIV`, `FDIV`). Također se može primjetiti da su operacije nad realnim brojevima (`FADD`, `FMUL`, `FDIV`) sporije od operacija nad cijelim brojevima (`ADD`, `MUL`, `DIV`). Najčešći primjeri *strength reductiona* uključuju mijenjanje množenja unutar petlje zbrajanjem te potenciranja množenjem. Takvo množenje često se pojavljuje pri pristupu nizovima unutar petlji. Iako prevodioc najčešće može dobro uočiti takve situacije i smanjiti složenost operacija uvođenjem zamjenskih instrukcija, ponekad se ubrzanje može postići primjenjivanjem sličnog principa izbjegavanja sporih operacija pri implementiranju postupka u jeziku više razine (C/C++).

4. Pregled korištene programske podrške

Implementacija algoritma projekcijskog ravninskog preslikavanja razvijana je u okviru ljuske *cvsh2* u programskom okruženju Microsoft Visual Studio. U nastavku poglavlja nalaze se upute za podešavanje razvojnog okruženja da bi se projekt mogao pokrenuti i kao i postavke pod kojima se postiže maksimalna performansa pri testiranju. Nakon toga slijede kratke upute za korištenje ljuske *cvsh2*.

4.1. Podešavanje razvojnog okruženja

Ljuska *cvsh2* koristi komponente iz skupa biblioteka Boost. Skup biblioteka Boost zamišljen je kao svjetski repozitorij biblioteka razreda u jeziku C++, zasnovan na ideji da programeri dijele svoje vlastite biblioteke pa na taj način nastaje repozitorij gdje mogu pronaći biblioteke koje su im potrebne [16]. Sve biblioteke u skupu Boost su standardizirane, odnosno napsane po određenim pravilima. Popis smjernica i zahtjeva za biblioteke uključene u skup Boost mogu se naći na [17]. Potreban skup biblioteka potrebno je preuzeti sa stranica *http://www.boost.org* i instalirati na računalo. Nakon toga u programskom okruženju Microsoft Visual Studio pod stavku *Project → Properties → C/C++ → General → Additional Include Directories* i *Project → Properties → Linker → General → Additional Library Directories* treba dodati putanju do instaliranih komponenti [18].

Kako bi se omogućio prikaz obrade videa, mora se postići kompatibilnost sa datotekama formata *.wmv* (*Windows Media Video*). Zbog toga je potrebno preuzeti *Windows Media Format SDK 11* sa stranica http://msdn.microsoft.com/en-us/windows/bb190307.aspx te dodati putanju do instalacije u programsko okruženje Microsoft Visual Studio na isti način kao i za skup biblioteka *Boost*. Dodatno, pod stavku *Input → Linker* potrebno je dodati parametre "*vfw32.lib"* i *"wmvcore.lib"* [18].

Tablica 4.1: Postavke parametara za Microsoft Visual Studio

parametar	vrijednost
Optimization → Optimization:	Maximize Speed (/O2)
Optimization → Enable Intrinsic Functions:	No
Optimization → Favor Size or Speed:	Favor Fast Code (/Ot)
Optimization → Omit Frame Pointers:	Yes (/Oy)
Code Generation Optimization → Enable Enhanced Instruction Set:	Streaming SIMD Extension 2 (/arch:SSE2)
Code Generation Optimization → Floating Point Mode:	Fast (/fp: fast)

Također, radi postizanja najboljih performansi, potrebno je podesiti parametre pod *Project → Properties → C/C++* na vrijednosti navedene u *Tablici 4.1.*

4.2. Rad sa ljuskom *cvsh2*

Ljuska *cvsh2* namjenjena je za rad preko komandne linije, pa je stoga u programskom okruženju Microsoft Visual Studio potrebno podesiti odgovarajuće argumente naredbenog retka. Argumenti naredbenog retka upisuju se pod *Project → Properties → Debugging → Command Arguments*. Ljuska *cvsh2* pri tome služi kao unificirano sučelje za rad i testiranje algoritama iz područja računalnog vida, obavlja poslove dobavljanja slike za korisnički kod te iscrtavanja rezultata [19].

Tablica 4.2: Parametri za rad sa ljuskom cvsh2

parametar	oblik
ulazna datoteka ili drugi izvor	-sf=<putanja_i_ime_datoteke> -sf=test
algoritam se želi primjeniti na dani ulaz	-a=alg_pbosilj
odredišna datoteka s rezultatima obrade ili interaktivni rad	-df=<putanja_i_ime_datoteke> -i -i="početne naredbe"
konfiguracija parametara algoritma – veličine izvorišne slike (opcionalno)	-c="400 400"

Prilikom pokretanja ljuske *cvsh2* potrebno je algoritmu predati parametre navedene u *Tablici 4.2.*

Tablica 4.3: Naredbe za interaktivni način rada

naredba	objašnjenje
p n *x*	obrada narednih *x* sličica (*engl. frame*) u videu
p a *x*	obrada sličice videa sa rednim brojem *x*
s n *x*	prikaz narednih *x* sličica videa bez obrade
s a *x*	prikaz sličice videa sa rednim brojem *x*
help	prikaz mogućih naredbi
quit	Izlaz iz aplikacije

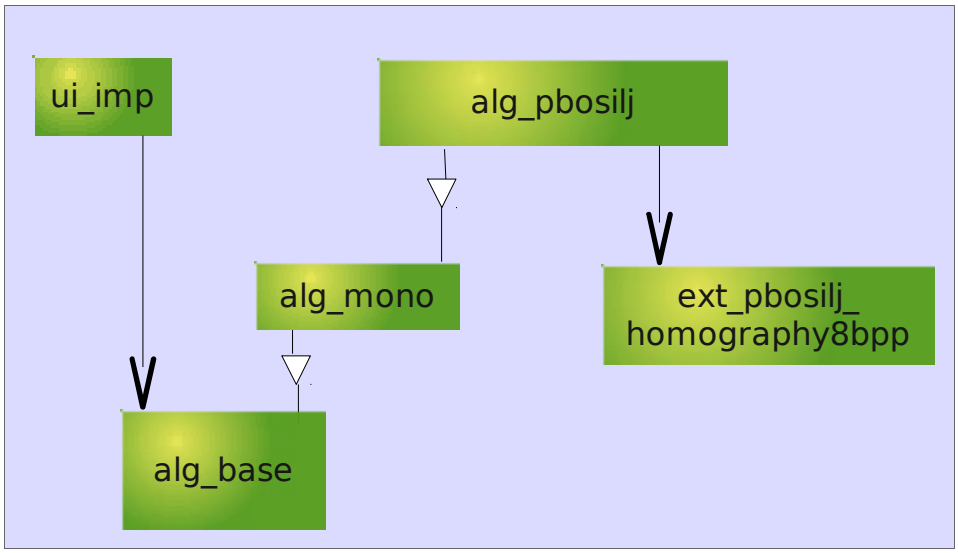
U *Tablici 4.3.* prikazane su naredbe koje se koriste u interaktivnom načinu rada i u sklopu parametra `-i="početne naredbe"` pri pokretanju programa. Ukoliko se ove naredbe koriste prilikom rada, unose se direktno i bez navodnika.

5. Programska implementacija

U ovom poglavlju opisana je struktura implementacije projekcijskog ravninskog preslikavanja, detaljnije pojašnjena implementacija algoritma kao i sve promjene koje su uvedene nakon osnovne verzije.

5.1. Struktura implementacije

Razred koji predstavlja algoritam za obradu slike koji nad zadanom ulaznom slikom provodi projekcijsku ravninsku transformaciju zadanu korisničkim ulaznim podatcima (klikovima miša) nalazi se u datoteci *alg_pbosilj.cpp*. Sučelje samog algoritma koji provodi inverz zadane homografije nad zadanom slikom nalazi se u datoteci *ext_pbosilj_homography8bpp.hpp*, dok se sama implementacija algoritma nalazi u datoteci *ext_pbosilj_homography8bpp.cpp*.



Slika 5.1: Grafički prikaz ovisnosti komponenti

Ovisnosi komponenata mogu se vidjeti na *Slici 5.1*. Komponenta *alg_base* je apstraktni razred koji definira izgled svih algoritama korištenih unutar ljuske, dok je *alg_mono* također apstraktni razred koji opisuje jedan od tipova algoritama koje je moguće implementirati unutar ljuske. Razred *ui_imp* opisuje implementaciju korisničkog sučelja. Iz slike je vidljivo da ne postoji direktna ovisnost između glavne funkionalnosti ljuske i implementacije samog algoritma.

```
typedef boost::numeric::ublas::matrix<double> Homography;
class alg_pbosilj:
    public alg_mono1
  {
private:
  std::vector<int> t_;
  int dstWidth_;
  int dstHeight_;

  math::Point2D ptBl_;
  math::Point2D ptBr_;
  math::Point2D ptTl_;
  math::Point2D ptTr_;
  Homography H_;
  win_ann ann_;
public:
  alg_pbosilj(const CtorArg&);
  virtual ~alg_pbosilj();
public:
  virtual char const* name(){return s_name();}
  static char const* s_name(){return "pbosilj";}
public:
//mandatory operations
  virtual void process(
    const img_wrap& src,
    const win_event_vectorAbstract& events,
    int msDayUtc,
    img_wrap& dst);
  virtual win_ann_abstract const* pAnnDst(int id);
public:
  virtual void getCurrentConfig(std::string& str);
  virtual void getConfigDescription(std::string& str);
  virtual void config(char const*);
  virtual void profile(ui_reportAbstract& );
private:
  void dumpParams(std::ostream& os, char const*);
  void recalculate(int w, int h);
};
```

Slika 5.2: Razred `alg_pbosilj`

Iz priloženog isječka koda može se vidjeti da razred *alg_pbosilj* sadržava sve podatke potrebne za obradu slike, kao i metode potrebne za fleksibilno upravljanje procesom obrade. Privatne članske varijalbe `dstWidth_` i `dstHeight_` određuju veličinu ishodišne slike, dok varijable tipa `math::Point2D` predstavljaju točke u kojima su zapisani korisnički ulazni podatci odnosno koordinate kuteva stare slike na području nove slike, iz kojih se izračunava matrica homografije `Homography H_`.

Tablica 5.1: Najbitnije metode razreda `alg_pbosilj`

metoda	opis
`virtual char const* name();` `static char const* s_name();`	Metode koje definiraju ime algoritma. Ovo ime potrebno je navesti pri pokretanju ljuske *cvsh2*.

`virtual void process(`
`  const img_wrap& src,`
`  const win_event_vectorAbstract& events,`
`  int msDayUtc,`
`  img_wrap& dst);`
	Najbitnija metoda razreda. Ovom metodom pokreće se jedna iteracija obrade slike.

`virtual void getCurrentConfig(std::string&);`	Metoda za ispis trenutnih postavki algoritma
`virtual void getConfigDescription (std::string&);`	Metoda koja ispisuje format konfiguraijskog stringa
`virtual void config(char const*);`	Metoda za rekonfiguriranje postavki algoritma (veličine odredišne slike).
`void dumpParams(std::ostream& os, char const*);`	Metoda za ispis postavki algoritma.
`virtual void profile(ui_reportAbstract& );`	Metoda koja pomaže pri mjerenju brzine izvršavanja implementacije.

Pregled najbitnijih metoda razreda *alg_pbosilj* dan je u *Tablici 5.1*, gdje se mogu vidjeti prototipovi metoda kao i kratki opis zadatka svake pojedine metode.

Sučelje algoritma u datoteci *ext_pbosilj_homography8bpp.hpp* definira samo jednu funkciju u okviru prostora imena *ext_pbosilj*. Funkcija definirana u okviru sučelja, `void homography8bpp (const img_wrap& src, const boost::numeric::ublas::matrix<double>& Hi, img_wrap& dst)`, kao parametre prima reference na izvorišnu sliku, matricu transformacije i odredišnu sliku.

5.2. Implementacija algoritma

Sama implementacija algoritma nalazi se u datoteci *ext_pbosilj_homography8bpp.cpp*. U okviru funkcije `homography8bpp` definirane u sučelju rade se potrebne pripremne radnje – ukoliko slika predana algoritmu nije siva, slika se transformira u sivu sliku. Također, izdvajaju se veličine izvorišne i odredišne slike, pošto je algoritam napisan na način da može raditi sa izvorišnim i odredišnim slikama različitih dimenzija. Nakon obavljanja pripremnih ranji, poziva se implementacija algoritma koja se nalazi u neimenovanom prostoru imena u funkciji `void myHomography8bpp`.

```
void myHomography8bpp(
   unsigned char const* psrc,
   int width, int height,
   const boost::numeric::ublas::matrix<double>& Hi,
   unsigned char* pdst){

   double h[3][3] = {{Hi(0,0), Hi(1,0), Hi(2,0)}, {Hi(0,1), Hi(1,1), Hi(2,1)},
                     {Hi(0,2), Hi(1,2), Hi(2,2)}};

   for (int y=0; y < height; ++y){
      double xh_12 = h[1][0] * y + h[2][0];
      double yh_12 = h[1][1] * y + h[2][1];
      double th_12 = h[1][2] * y + h[2][2];

      for (int x=0; x < width; ++x){
         double xh = h[0][0] * x + xh_12;
         double yh = h[0][1] * x + yh_12;
         double th = h[0][2] * x + th_12;

         double x_ = xh/th;
         double y_ = yh/th;
         int x_floor_ = (int)x_;
         int y_floor_ = (int)y_;
         double dx = x_ - x_floor_;
         double dy = y_ - y_floor_;
         int y_Pomak = y_floor_ * width;

         if ( x_ >= 0.f && y_ >= 0.f &&
              x_ < ((double)width-1) && y_ < ((double)height-1)) {
            double result = dx *    dy    * psrc [x_floor_ + 1 + y_Pomak + width] +
                            dx * (1 - dy) * psrc [x_floor_ + 1 + y_Pomak        ] +
                     (1 - dx) *    dy    * psrc [x_floor_     + y_Pomak + width] +
                     (1 - dx) * (1 - dy) * psrc [x_floor_     + y_Pomak        ];
            pdst[x + width * y] = (int)(result + 0.5);
         }
         else{
            pdst[x + width * y] = 0;
         }
      }
   }
   return;
}
```

Slika 5.2: Prva implementacija algoritma homografije

Početna i osnovna implementacija homografije može se vidjeti na *Slici 5.3.* Ovaj kod korišten je kao baza na kojoj su se testirali razni koncepti korišteni za ubrzanje unoseći promjene u kod. Promjene koje su donjele ubrzanje zadržane su u kodu i na taj je način dobivena konačna, optimizirana verzija implementacije.

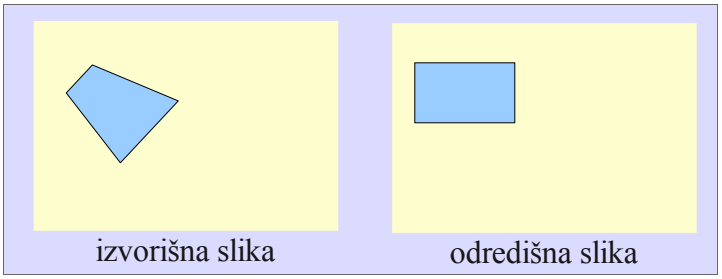
Za testiranje koncepta vektorskih instrukcija i ubrzanja koje bi se na taj način moglo postići kod osnovne funkcije prekopiran je u novu funkciju, `myHomography8bppSSE`, te je mjenjan liniju po liniju na način da koristi intrinzične funkcije.

Tablica 5.2: Primjer koda napisanog pomoću intrinzičnih funkcija

osnovna implementacija	intrinzične funkcije

`for (int y=0; y < height; ++y){`
`  double xh_12 = h[1][0] * y + h[2][0];`
`  double yh_12 = h[1][1] * y + h[2][1];`
`  double th_12 = h[1][2] * y + h[2][2];`
 	
`  for (int x=0; x < width; ++x){`
`  double xh = h[0][0] * x + xh_12;`
`  double yh = h[0][1] * x + yh_12;`
`  double th = h[0][2] * x + th_12;`
`  ...`
`  }`
`...`
`}`
	`for (int y=0; y < height; ++y){`
	`  __m128 y_v = _mm_set1_ps((float)y);`
	`  y_v = _mm_mul_ps(h1_v, y_v);`
	`  y_v = _mm_add_ps(h2_v, y_v);`
	`  for (int x; width; ++x){`
	`  __m128 results_v = _mm_set1_ps((float)x);`
	`  results_v = _mm_mul_ps(h0_v, results_v);`
	`  results_v = _mm_add_ps(y_v, results_v);`
	`  ...`
	`  }`
	`...`
	`}`

U *Tablici 5.2*. može se vidjeti primjer koda promjenjen na ovaj način. Da bi se mogle koristiti intrinzične funkcije, potrebno je koristiti tip podataka __m128 koji sadrži četiri podatka tipa float poravnata na parne adrese u memoriji i služi za direktno predstavljanje xmm registara. U ovom primjeru može se vidjeti primjer intrinzičnih funkcija za postavljanje sva četiri podatka jednostruke preciznosti registra na istu vrijednost (`_mm_set1_ps()`), za paralelno množenje (`_mm_mul_ps()`) i za paralelno zbrajanje podataka (`_mm_add_ps()`). Nakon testiranja ustanovljeno je da je kod koji koristi intrinzične funkcije sporiji od osnovne verzije koda. Vjerojatan razlog tome je taj što, ako se uključi opcija prevodioca da koristi vektorske asemblerske instrukcije, prevodioc uočava prilike za optimizaciju koda mnogo bolje nego što je to programer sposoban. Iz razloga što ovaj pristup nije donio nikakvo ubrzanje, ideja korištenja vektorskih instrukcija i intrinzičnih funkcija pri daljnjem je radu odbačena i sva daljnja poboljšanja rađena su nad osnovnom verzijom implementacije.



Slika 5.3: Shematski prikaz odgovarajućih područja na izvorišnoj i odredišnoj slici

Još jedan od koncepata kojeg se pokušalo uvrstiti u rješenje, ali nije donio nikakvo ubrzanje je *blocking*. Na *Slici 5.3.* može se vidjeti shematski prikaz označenog područja izvorišne slike koje se preslikava u označeno područje odredišne slike. Glavna ideja iza primjene ovog koncepta bila je u tome da, ukoliko se pikseli odredišne slike izračunavaju u malim blokovima, "poligoni" relevantnih podataka na izvorišnoj slici postat će dovoljno mali da bi mogli čitavi stati u priručnu memoriju.

Tablica 5.3: Uvođenje blockinga u osnovnu implementaciju

osnovna implementacija	uvođenje blockinga

`for (int y=0; y < height; ++y){`
`  ...`
`  for (int x=0; x < width; ++x){`
`  ...`
`  }`
`  ...`
`}`
	`#ifdef BLOCKING`
	`  int chunkX = 128;`
	`  int chunkY = 128;`
	`#else`
	`  int chunkX = width;`
	`  int chunkY = height;`
	`#endif`
	`  for (int y=0, yIter=1; y < height; ++yIter){`
	`  for (; y < std::min(height, yIter*chunkY); ++y){`
	`  for (int x=0, xIter = 1; x < width; ++xIter){`
	`  for (; x < std::min(width, xIter*chunkX); ++x){`
	`  ...`
	`  }`
	`  }`
	`  ...`
	`}`
	`}`

Način uvođenja *blockinga* u osnovnu implementaciju može se vidjeti u *Tablici 5.3*, pri čemu se sam postupak blokinga aktivira definiranjem konstante prevodioca BLOCKING. Problem sa ovim pristupom je da bi se veličina bloka trebala prilagoditi veličini priručne memorije, što je nemoguće podesiti za izvorišnu sliku. Također, čak i sa dovoljno malim

blokovima u odredišnoj slici, za pretpostaviti je da su odgovarajući dijelovi izvorišne slike previše nepravilni da bi se postigao ikakav vidljivi učinak.

U kod je tijekom procesa razvoja uvedena i jedna konceptualna izmjena. Osnovna verzija implementacije pretpostavljala je istu veličinu izvorišne i odredišne slike (parametri *width* i *height*), dok završna verzija koda radi sa razvojenim dimenzijama izvorišne (parametri *srcWidth* i *srcHeight*) i odredišne (parametri *dstWidth* i *dstHeight*).

Tablica 5.4: Primjeri uvođenja strength reductiona

stari kod	**novi kod**

x_ = xh/th;
y_ = yh/th;
	`double` th_rcp = 1/th;
	x_ = xh*th_rcp;
	y_ = yh*th_rcp;

U starom kodu nije korišteno.	`unsigned char const*` pSrcBase = psrc + y_floor * srcWidth + x_floor;

`int y_Pomak = y_floor_ * width;`
`result=dx * dy * psrc [x_floor_ + 1 + y_Pomak + width] +`
`dx * (1 – dy) * psrc [x_floor_ + 1 + y_Pomak] +`
`(1 - dx) * dy * psrc [x_floor_ + y_Pomak + width] +`
`(1 - dx) * (1 - dy) * psrc [x_floor_ + y_Pomak];`
	result = dxdy*pSrcBase[1+srcWidth];
	result += (dx - dxdy) * pSrcBase[1];
	dxdy = dy - dxdy;
	result += dxdy * pSrcBase[srcWidth];
	result += (1 -dx -dxdy)*pSrcBase[0];

`xh_12 = h[1][0] * y + h[2][0];`
`yh_12 = h[1][1] * y + h[2][1];`
`for (int x=0; x < dstWidth; ++x){`
`xh = h[0][0] * x + xh_12;`
`yh = h[0][1] * x + yh_12;`
`...`
`}`
	`xh_12 = h[1][0] * y + h[2][0];`
	`yh_12 = h[1][1] * y + h[2][1];`
	`xh = xh_12;`
	`yh = yh_12;`
	`for (int x =0; x < dstWidth; ++x){`
	`xh += h[0][0];`
	`yh += h[0][1];`
	`...`
	`}`

Neki od uspješnih primjera korištenja *strength reductiona* mogu se vidjeti u *Tablici 5.4*. Iz danih primjera može se vidjeti da je fokus pri tome bio na smanjenju broja naredbi djeljenja i množenja te kompliciranih memorijskih pristupa (pristup polju može se također promatrati kao množenje). Također je bitno napomenuti da je tehnike *strength reductiona* imalo smisla koristiti samo unutar unutarnje petlje (petlje po x-u), jer bi ubrzanje dobiveno na ovaj način izvan te petlje bilo zanemarivo u odnosu na vrijeme izvođenja unutarnje petlje.

Koncept višedretvenosti također je uspješno uveden u završnu verziju koda koristeći aplikacijsko programsko sučelje OpenMP. Da bi se *pragme* OpenMP-a mogle primjeniti, bilo je potrebno cijelo tijelo vanjske petlje (petlju po x-u i pripremne izračune) odvojiti u

posebnu funkciju, koja se tada kao jedina naredba pozivala unutar vanjske petlje.

Osim koncepata koji su primjenjivi na cijeli algoritam, implementacija je prilagođena i da prepoznaje posebni slučaj projekcijskog ravninskog preslikavanja, afinu transformaciju. Postupak izračunavanja vrijednosti piksela odredišne slike prilikom afine transformacije je jednostavniji nego prilikom izračunavanja po općenitom algoritmu. Na taj je način, u nekim posebnim slučajevima, bilo moguće izbjeći dodatne operacije množenja i dijeljenja.

Korištenje višedretvenosti i uvođenje posebnog načina računanja za afinu transformaciju moguće je vidjeti u završnoj verziji koda.

```
const double eps = 1e-12;

void myHomography8bpp(
  unsigned char const* psrc,
  int srcWidth, int srcHeight,
  const boost::numeric::ublas::matrix<double>& Hi,
  unsigned char* pdst,
  double dstWidth, double dstHeight){

  bool affine = false;
  double h[3][3] = {{Hi(0,0), Hi(1,0), Hi(2,0)}, {Hi(0,1), Hi(1,1), Hi(2,1)},
              {Hi(0,2), Hi(1,2), Hi(2,2)}};

  if (fabs(h[0][2]) < eps && fabs(h[1][2]) < eps){
    if (fabs(h[2][2]) < eps){
      throw std::runtime_error
            ("myHomography8bpp: got degenerate homography matrix on input");
    }
    else{
      h[0][0] /= h[2][2];
      h[1][0] /= h[2][2];
      h[2][0] /= h[2][2];
      h[0][1] /= h[2][2];
      h[1][1] /= h[2][2];
      h[2][1] /= h[2][2];
      affine = true;
    }
  }
    #pragma omp parallel for num_threads(8) schedule(dynamic)
  for (int y = 0; y < dstHeight; ++y){
    xLoop(h[0],h[1],h[2],y,affine, psrc, srcWidth, srcHeight, pdst, dstWidth, dstHeight);
  }
  return;
}
```

Slika 5.4: Završna implementacija algoritma homografije

```
inline void xLoop(double* h0, double* h1, double* h2, int y, bool affine,
  unsigned char const* psrc,
  int srcWidth, int srcHeight,
  unsigned char* pdst,
  int dstWidth, int dstHeight){

  double xh = h1[0] * y + h2[0];
  double yh = h1[1] * y + h2[1];
  double th;

  if (!affine){
    th = h1[2] * y + h2[2];
  }
  unsigned char* pBaseDst =&pdst[dstWidth*y];
  for (int x = 0; x < dstWidth; ++x){
    xh += h0[0];
    yh += h0[1];

    double x_,y_;
    if (!affine){
      th += h0[2];
      double th_rcp = 1/th;
      x_ = xh*th_rcp;
      y_ = yh*th_rcp;
    }
    else{
      x_ = xh;
      y_ = yh;
    }

    if ( x_ >= 0.0 && y_ >= 0.0 && x_ < ((double)srcWidth-1) && y_ < ((double)srcHeight-1)){
      int x_floor = (int)x_;
      int y_floor = (int)y_;
      double dx = x_ - x_floor;
      double dy = y_ - y_floor;
      unsigned char const* pSrcBase = psrc + y_floor * srcWidth + x_floor;

      double dxdy = dx * dy;
      double result = dxdy * pSrcBase [1 + srcWidth];
           result += (dx - dxdy) * pSrcBase[1];
           dxdy = dy - dxdy;
           result += dxdy * pSrcBase[srcWidth];
           result += (1 - dx - dxdy) * pSrcBase[0];
      pBaseDst[x] = (int)(result + 0.5);

    }
    else{
      pBaseDst[x] = 0;
    }
  }
}
```

Slika 5.5: Funkcija xLoop koja zamjenjuje unutrašnju petlju u završnoj implementaciji

Završna, optimizirana verzija homografije može se vidjeti na *Slici 5.4* i *Slici 5.5*. U ovoj se verziji mogu vidjeti sva do sada spomenuta ubrzanja, kao i razne ostale sitne promjene unešene u programski kod. Na *Slici 5.5* može se vidjeti unutrašnja petlja izdvojena u zasebnu funkciju zbog uvođenja paralelizacije pomoću OpenMP-a. Također se može vidjeti da se, za razliku od početne implementacije, u završnoj verziji provjerava da li dobivena transformacijska matrica predstavlja ispravno zadanu homografiju.

6. Rezultati i rasprava

Prilikom izrade rada, koristile su se dvije vrste evaluacije nastale implementacije: evaluacija točnosti i procjena brzine odnosno ubrzanja u odnosu na osnovnu verziju implementacije.

6.1. Evaluacija točnosti implementacije

Prilikom razvijanja programske implementacije, ispravnost rezultata procjenjivala se promatranjem rada programa i interakcijom sa programom preko korisničkog sučelja. Korisniku je omogućeno mišem odrediti položaj vrha po vrha područja iduće projekcije, određujući na taj način homografiju za idući korak.

Kako se na taj način nije moglo provjeriti radi li implementacija (ispravno ili uopće) nad svim ulaznim podatcima, provedeno je testiranje da se utvrdi hoće li implementacija funkcionirat nad velikim brojem slučajnih ulaznih podataka, bez provjere ispravnosti rezultata.

Dio programskog koda u datoteci *alg_pbosilj.cpp* koji učitava korisničke unose zanemaruje se i korisnički unosi simuliraju se slučajnim odabirom. Simulirani korisnički unosi zapisuju se u dnevničku datoteku.

Tablica 6.1: Kod za testiranje točnosti i funkcionalnosti implementacije

kod za učitavanje korisničkog unosa

```
for (int i=0; i<events.size(); ++i){
   const win_event_abstract& e(events[i]);
   if (e.type()==win_event_abstract::MouseLeft){
      math::Point2D pt(e.par1(), e.par2());
      if (0) ;
      else if (e.modifiers()==win_event_abstract::Ctrl)
         ptBl_=pt;
      else if (e.modifiers()==win_event_abstract::Shift)
         ptTl_=pt;
      else if (e.modifiers()==win_event_abstract::Shift+
                              win_event_abstract::Ctrl)
         ptBr_=pt;
      else if (e.modifiers()==0)
         ptTr_=pt;
   }
}
```

kod za simulaciju korisničkog unosa

```
if (randomTest){
   int eventNum = rand() % 4;
   math::Point2D pt(rand() % dstWidth,
                    rand() % dstHeight);
   if (eventNum == 0)
      ptBl_=pt;
   else if (eventNum == 1)
      ptTl_ = pt;
   else if (eventNum == 2)
      ptBr_ = pt;
    else
      ptTr_ = pt;

   FILE *log = fopen("log.txt", "a+");
   fprintf(log, "ptBl_ (%d %d), PtTl_ (%d %d),
                 ptBl_ (%d %d), PtTl_ (%d %d)\n",
                 (int)ptBl_.x(), (int)ptBl_.y(),
                 (int)ptTl_.x(), (int)ptTl_.y(),
                 (int)ptBr_.x(), (int)ptBr_.y(),
                 (int)ptTr_.x(), (int)ptTr_.y());
   fclose(log);
}
```

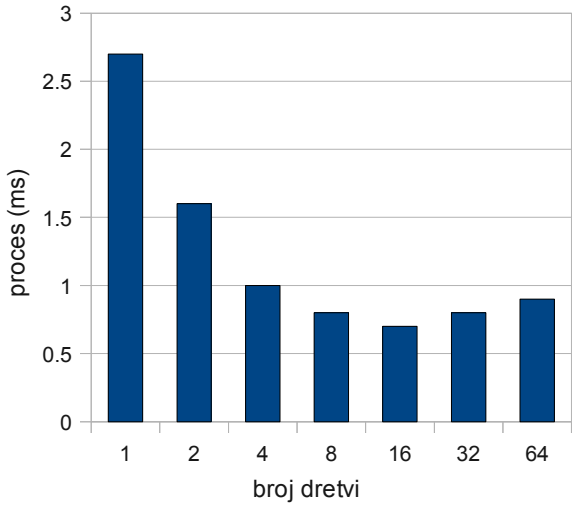
Programski kod koji učitava korisničke unose i kod koji ga simulira prilikom testiranja može se vidjeti na *Tablici 6.1.* Za simulaciju korisničkog unosa potrebno je također u konstruktor razreda *alg_pbosilj* dodati liniju `srand(time(NULL));`

Prilikom završnog tesitranja, provjereno je da implementacija funkcionira nad 7.585.817 različitih konfiguracija koje određuju homografiju prije nego što je prekinuto.

6.2. Procjena postignutog ubrzanja u odnosu na početnu verziju

Vremena za procjene ubrzanja implementacije dobivena su testiranjem na računalu sa procesorom i7-860 sa 4 jezgre i hyperthreadingom. Vrijeme koje se uspoređivalo je prosječno vrijeme izvršavanja procesa (transformacije) nad 5000 sličica (*engl. frame*), koje zanemaruje vrijeme dohvata i spremanja slike. Ovo vrijeme jedan je od podataka koje sučelje samo pruža.

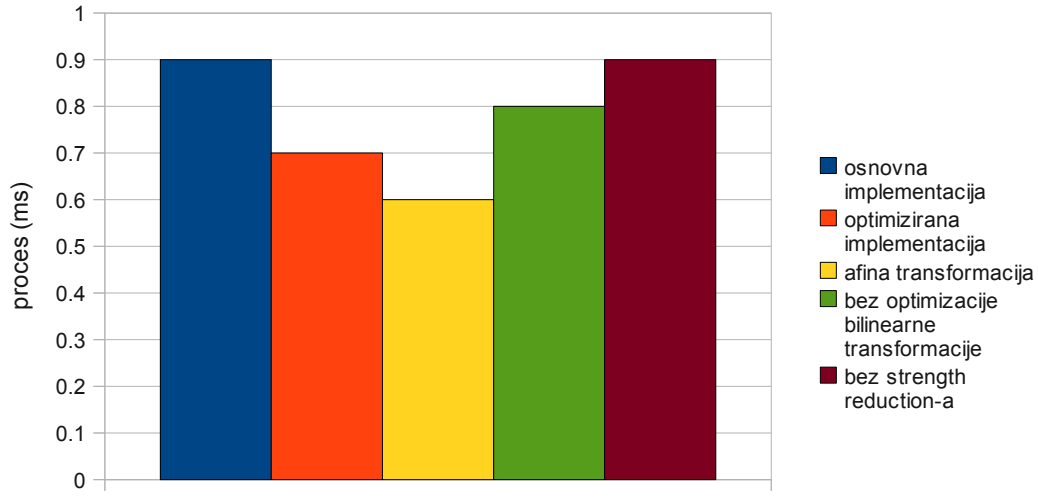
Prvo je pronađen broj dretvi koji daje najbolju performansu na ovom računalu Testiranje je rađeno sa ulaznim slikama veličine 2880x2048 piksela koje se preslikavaju u slike veličine 400x400 piksela.



Slika 6.1: Ovisnost brzine izvršavanja i broja dretvi

Kao što se može vidjeti iz *Slike 6.1.* koja prikazuje brzine izvršavanja najoptimiziranije verzije implmentacije algoritma projekcijskog ravninskog preslikavanja, na ovom računalu najbolja performansa postiže se sa 16 dretvi, i vrijeme izvršavanja je 0.7ms.

Testiranje brzine razvijene implementacije rađeno je također sa izvorišnim slikama veličine 2880x2048, a odredišnim 400x400. Ovo je testiranje rađeno tako da je algoritam pokrenut u 16 dretvi.



Slika 6.2: Usporedba brzine nekoliko verzija implementacije

Rezultati ovog testiranja mogu se vidjeti na *Slici 6.2,* na kojoj se može vidjeti da najveće ubrzanje unosi *strength reduction* (vidi se da je bez ključnih djelova u kojima je primjenjena tehnika *strength reductiona* implementacija najsporija), kao i brzina posebnog postupka za afinu transformaciju.

Također, ispitano je ubrzanje u ovisnosti o prethodno razvijenoj implementaciji projekcijskog ravninskog preslikavanja koja se može naći u datotekama *alg_homography_test.cpp* i *ipp_homography.cpp*. U odnosu na ovu verziju, nova se verzija izvršava čak 158 puta brže. Potrebno je napomenuti kako prethodno razvijena implementacija radi za sve formate ulaznih slika, a brzina izvođenja mjerena je nad slikama u boji pa se može pretpostaviti da bi ubrzanje bilo nešto manje ukoliko bi se brzina izvođenja mjerila na sivim slikama. Osim mogućnosti provođenja algoritma nad svim vrstama slika, razlog što je prethodna implementacija toliko spora je također i velika količina funkcijskih poziva – za svaki piksel odredišne slike na taj se način računa njegova vrijednost.

7. Zaključak

Izrada ovog rada omogućila je detaljnije upoznavanje s jednim od važnijih postupaka iz područja računalnog vida koji se koristi u okviru mnogih algoritama specifične namjene i prilikom rješavanja raznih problema.

Dobro poznavanje programskog jezika kao i znanja iz područja arhitekture računala omogućila su kvalitetnu izradu optimizirane implementacije projekcijskog ravninskog preslikavanja. Pri tome je najveće ubrzanje od čak 285.7% postignuto uvođenjem višedretvenosti korištenjem aplikacijskog programskog sučelja OpenMP. Primjetno ubrzanje također je postignuto korištenjem tehnika *strength reductiona*, koje su unijele ubrzanje od 28.6%. Predviđen je također i poseban postupak za slučaj afine transformacije, pri čemu se poseban slučaj izvršava 16.7% brže od općenitog.

Uz ubrzanje od čak 158 puta u odnosu na postojeću implementaciju algoritma projekcijskog ravninskog preslikavanja iz ljuske *cvsh2*, integracijom ove implementacije u ljusku omogućen je razvoj složenijih postupaka obrade slike i videa te postupaka iz područja računalnog vida za koji uz postojeću implementaciju ne bi mogli biti implementirani dovoljno efikasno za praktične primjene.

Budući rad mogao bi uključivati postupan razvoj optimiziranih implementacija algoritma za sve formate ulaznih slika korištenjem koncepata istraženih u okviru ovog rada ili prepravljanje razvijene implementacije na način da obrađuje sve formate ulaznih slika. Pri razvoju zasebnih optimiziranih implementacija bilo bi potrebno obratiti pozornost na mogućnost nepotrebnog dupliciranja programskog koda, dok bi se prilikom prepravljanja razvijene implementacije trebalo pripaziti da se zbog prevelike generalizacije razvijene implementacije ne izgubi na performansi.

8. Literatura

[1] Hartley, R., Zisserman, A. : Multilpe View Geometry in computer vision, Second Edition, 2004.

[2] Interpolation – Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Interpolation, 9. lipnja 2010.

[3] Linear Interpolation – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Linear_interpolation, 10. ožujka 2010.

[4] Bilinear Interpolation – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Bilinear_interpolation, 8. lipnja 2010.

[5] Flyn's taxonomy – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Flynn's_taxonomy, 14. svibnja 2010.

[6] SIMD – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/SIMD, 6. srpnja 2010.

[7] SSE2 – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/SSE2, 30. lipnja 2010.

[8] MMX, SSE and SSE2 intrinsics, http://msdn.microsoft.com/en-us/library/y0dh78ez(VS.80).aspx, 2010.

[9] OpenMP – Wikipedia, the free enyclopedia, http://en.wikipedia.org/wiki/OpenMP, 3. srpnja 2010.

[10] About OpenMP and OpenMP.org, http://openmp.org/wp/about-openmp/, 7. lipnja 2010.

[11] Data parallelism – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Data_parallelism, 4. lipnja 2010.

[12] Task parallelism – Wikipedia the free encyclopedia,

http://en.wikipedia.org/wiki/Task_parallelism, 27. svibnja 2010.

[13] Palašek, P., Pronalaženje, prepoznavanje i praćenje razdjelne linije iz perspektive vozača, Završni rad br. 1552, Fakultet elektrotehnike i računarstva, Zagreb, 2010.

[14] S. Ribarić, Arhitektura računala CISC i RISC, Zagreb, 1996.

[15] Locality of reference – Wikipedia the free encyclopedia, http://en.wikipedia.org/wiki/Locality_of_reference, 30. lipnja 2010.

[16] J. L. Hennessy, D. A. Patterson: Computer Arhitecutre, A Quantitative Approach, Third edition, 2003.

[17] B. G. Dawes, Proposal for a C++ Library Repository Web Site, 6. svibnja 1998.

[18] Guidelines – Boost C++ Libraries, https://svn.boost.org/trac/boost/wiki/Guidelines, travanj 2010.

[19] T. Benussi, P. Bosilj, M. Čuljak, D. Jurić, B. Miklenić, M. Morava, A. Trbojević, L. Zadrija: Pronalaženje i praćenje prometnih znakova, Dokumentaicija dodiplomskog projekta, Fakultet elektrotehnike i računarstva, Zagreb, 2010.

[20] A. Fog: Instruction Tables; Lists of instruction latencies, throughputs and micro- operation breakdowns for Intel, AMD and VIA CPUs , Copenhagen University College of Engeneering, 2010.

9. Sažetak / Abstract

U ovom radu razmatrana je primjena različite tehnike optimizacije pri razvoju implementacije algoritma projekcijskog ravninskog preslikavanja. Projekcijsko ravninsko preslikavanje važan je postupak iz područja računalnog vida sa raznim direktnim primjenama kao i primjenama u sklopu složenijih algoritama. Razmatrane tehnike uključuju uvođenje paralelizma, vektorske instrukcije, *blocking* i *strength reduction.* U obzir su također uzete i posebnosti samog algoritma.

Algoritam projekcijskog ravninskog preslikavanja, odnosno homografije, implementiran je u programskom jeziku C++. Zbog posebnosti algoritma u slučaju afine transformacije, ovaj se slučaj homografije obrađuje drugačije od općeg slučaja čime se postiže dodatno ubrzanje prilikom izvođenja posebng slučaja.

Ubrzanja u konačnoj veziji postignuta su uvođenjm paralelizma pomoću aplikacijskog programskog slučelja OpenMP te raznim primjenama tehnika *strength reductiona* koje se temelje na poznavanju područja arhitekture računala.

Ključne riječi: projekcijsko ravninsko preslikavanje, homografija, afina transformacija, računalni vid, obrada slike, optimizacija, paralelizam, OpenMP, vektorske instrukcije, *blocking, strength reduction*

Design of an optimized software implementation of projectiove planar mappings

This work considers various optimization tehniques in design of an implementation of projective planar mappings algorithm. Projective planar mappings are an important method in computer vision, with many direct appliances as well as usages in more complex algorithms. Considered tehniques include introducing parallelism into the implementation, vector instructions, blocking and strength reduction tehniques. Particularities of the given algorithm are also taken under consideration.

Projective planar mappings algorithm, also called homography, is implemented in programming language C++. Special case of homography, affine transformation, is proessed differently and thus faster then the general case.

The final version of the implementation relies on introducing parallelism with OpenMP application programming interface and various strength reduction techniques that consider different aspects of computer architecture for achieving the best performance.

Key words: projective planar mappings, homography, affine transformation, computer vision, image processing, optimization, parallelism, OpenMP, vector instructions, blocking, strength reduction