

Universitatea Tehnică "Gh. Asachi" Iași
Facultatea de Automatică și Calculatoare

Domeniul: Calculatoare și Tehnologia Informației
Specializarea: Tehnologia Informației

Lucrare de licență

Identificarea obiectelor în secvențe video utilizând puncte de interes

Absolvent

Carata Lucian

Coordonator științific
Prof. Dr. Ing. Vasile Manta

Iași, 2009

Programs must be written for people to read, and only incidentally for machines to execute.

Programele trebuie scrise mai întâi pentru a fi citite de oameni și doar apoi pentru a fi executate de mașini.

— Abelson and Sussman

Cuprins

Cuprins	i
1 Introducere	1
1.1 Recunoașterea automată a obiectelor	1
1.2 Formularea și abordarea temei	2
2 Noțiuni teoretice	5
2.1 Abordări ale temei în literatura de specialitate	5
2.2 Identificarea trăsăturilor	7
2.2.1 Detectorul Harris	7
2.2.2 Detectorul SIFT	9
2.3 Identificarea trăsăturilor în timp real	13
2.3.1 Detectorul SURF	13
2.3.2 Alegerea parametrilor de rulare ai algoritmilor	18
2.4 Potrivirea trăsăturilor	19
3 Proiectarea aplicației	21
3.1 Descrierea aplicației	21
3.2 Obiective	21
3.3 Constrângeri	22
3.4 Arhitectura generală a proiectului software	23
3.5 Descrierea detaliată a componentelor	24
3.5.1 Nivelul de abstractizare a achiziției imaginilor	24
3.5.2 Nivelul de prelucrare a imaginilor	26
3.5.3 Nivelul de administrare a resurselor de procesare	27
3.5.4 Interfața grafică	29
3.6 Metriki de proiectare	29
4 Implementarea aplicației	31
4.1 Descriere generală	31

4.1.1	Identificarea provenienței unui fragment de cod	32
4.2	Biblioteci externe utilizate	33
4.2.1	Motivație	33
4.2.2	Biblioteci externe	33
4.3	Soluții de implementare	34
4.3.1	Tipuri utilizate de algoritmul SURF	34
4.3.2	Imagini Integrale	35
4.3.3	Implementarea algoritmului SURF	37
4.3.4	Paralelizarea procesării	40
4.4	Probleme apărute în dezvoltare. Soluții propuse	42
4.4.1	Rularea pe mai multe fire de execuție, fără a utiliza primitive de sincronizare	42
4.4.2	Afișarea în interfață grafică a unui număr foarte mare de obiecte	43
4.4.3	Deadlock-uri în codul DirectShow	44
4.4.4	Ordinea obținerii rezultatelor procesării	45
4.5	Interfața cu utilizatorul	45
5	Compilarea și testarea aplicației	49
5.1	Compilarea aplicației	49
5.1.1	Dependențe	49
5.2	Testarea unităților	50
6	Rezultate și Concluzii	53
6.1	Rezultate experimentale	54
6.2	Concluzii	58
6.2.1	Directive de cercetare	59
	Bibliografie	61
A	Diagrame UML	65
B	Filtre utilizate în algoritmul SURF	71
C	Codul aplicației (parțial)	75
	Lista de simboluri și prescurtări	99
	Listă de figuri	100
	Glosar	103

Capitolul 1

Introducere

1.1 Recunoașterea automată a obiectelor

Identificarea și recunoașterea automată a unor obiecte, în imagini statice sau secvențe video, este îndelung studiată în grafica pe calculator, prezentând interes din perspectiva dificultăților întâmpinate în rezolvarea problemei de sistemele de calcul în comparație cu sistemele biologice, dar și datorită aplicabilității în domenii din cele mai diverse.

Astfel, primele aplicații s-au conturat în mediul industrial, pentru inspectarea automată a produselor de pe liniile de fabricație (de exemplu, identificarea defectelor unor plăci integrate - lipituri incorecte, plasari incorecte de componente etc). Totuși, mediul de recunoaștere în aceste cazuri este unul controlat, existând limite stricte între care recunoașterea se realizează cu succes. Mai mult, obiectele pentru care se realizează identificarea au caracteristici bine cunoscute. Pornind de aici, s-au dezvoltat metode care încearcă să elimine cât mai multe dintre restricții, și să permită recunoașterea în cazul general. Aceste abordări largesc gama de aplicații și în zona utilizatorilor obișnuiți, pentru îmbunătățirea următoarelor generații de motoare de căutare, programe de chat sau de supraveghere a locuințelor. Desigur, domenii precum medicina (recunoașterea sau numărarea celulelor de un anumit tip), robotica (dezvoltarea unor roboți care să interacționeze cu mediul înconjurător folosind "vederea artificială") utilizează și ele recunoașterea obiectelor ca subproblemă. Aplicații similare există în domeniul militar (recunoașterea unor dispozitive suspecte în aeroporturi, identificarea persoanelor pe baza infățișării).

Problemele cele mai mari în identificarea și recunoașterea obiectelor apar datorită variațiilor din mediul în care obiectul este plasat (culoare, lumină, umbre, reflexii, ocluzionarea obiectului țintă de către alte obiecte). De asemenea, apar dificultăți și datorită diferențelor de poziționare și perspectivă între reprezentan-

rea inițială a obiectului care se dorește a fi identificat (de cele mai multe ori, o fotografie a respectivului obiect) și situația reală în care se încearcă identificarea acestuia (când el poate fi "privit" la o altă scală, rotit sau dintr-un punct de vedere diferit).

Au fost găsite mai multe abordări pentru rezolvarea acestor probleme, majoritatea detectând în fiecare imagine anumite zone caracteristice, invariante la modificări ale parametrilor de mediu/perspectivă. Se realizează apoi o potrivire între ele și o bază de date în care au fost anterior reținute caracteristicile obiectelor căutate. Diferențele între metode se referă la modalitatea de detecție a zonelor, la forma lor (punkte, arii din imagine) și la informațiile reținute pentru fiecare zonă în parte astfel încât ea să poată fi regăsită într-o nouă imagine și atribuită ca apartinând obiectului căutat.

La momentul actual, tehniciile de recunoaștere nededicate permit o detecție cu un procentaj de reușită și repetabilitate a rezultatelor suficient de mare (tipic peste 80%) pentru a fi considerate aplicabile cu succes în aplicații de orice tip. Totuși, se pune problema selectării unor metode cât mai eficiente, care să poată fi aplicate "în timp real".

1.2 Formularea și abordarea temei

Recunoașterea unor obiecte (furnizate ca imagini, drept date de intrare), într-o secvență video live sau filmată anterior (video salvat pe hdd), presupune detectarea existenței obiectelor și identificarea poziției acestora în fiecare cadru (eng. frame), urmată de "adnotarea" cadrului în zona obiectului recunoscut. Atât inițial, pentru imaginile ce definesc obiectele, cât și pentru secvența video, se aplică aceeași algoritmi de determinare a zonelor caracteristice. Apoi, se realizează o potrivire între rezultatele obținute pentru cadrul curent și modelul determinat pentru fiecare dintre obiecte. În măsura în care există corespondențe (în cadrul curent există zone similare cu cele ale obiectului), se stabilește prezența obiectului, precum și poziția acestuia. Se dorește o variație cât mai mică a rezultatelor la schimbări de scală, rotații, modificări ale perspectivei 3D și a luminozității ambientale, urmărind în același timp o repetabilitate crescută a experimentelor. De asemenea, este de preferat să nu se impună restricții legate de modul în care sunt capturate imaginile sau de calibrarea anterioară a dispozitivelor (camere video, aparate foto digitale).

Dacă drept zone caracteristice se folosesc puncte de interes, există 2 pași generali în analiza fiecărei imagini prelucrate:

1. *Localizarea*: determinarea automată a poziției punctelor de interes (în două imagini ale aceluiași obiect, se dorește ca punctele de interes să fie localizate în aceeași zonă relativ la obiect - Figura 1.1)

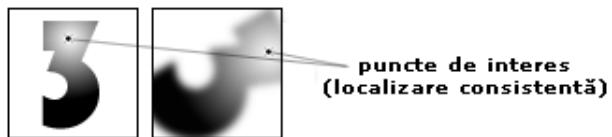


Figura 1.1: *Localizarea punctelor de interes*: rulând în mod independent algoritmul pe două imagini ale aceluiași obiect în situații diferite, se dorește ca punctele de interes să fie identificate în aceleasi poziții relativ la obiect

2. *Descrierea*: fiecărui punct de interes determinat anterior îi sunt asociate o mulțime de date rezultate din analiza imaginii, astfel încât el să poată fi identificat cu un grad ridicat de individualitate în comparație cu restul punctelor de interes (Figura 1.2). În imagini diferite ale aceluiași obiect, vectorul obținut trebuie să fie invariant la modificări ale mediului (luminozitate) sau la transformări affine asupra obiectului (translații, rotații, scalări), pentru a asigura o recunoaștere adecvată (căutarea se realizează pe baza descriptorilor).

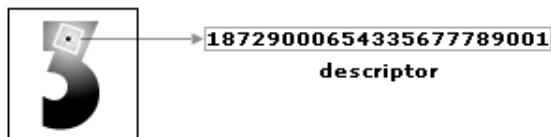


Figura 1.2: *Descrierea punctelor de interes*: Asocierea de informații pentru identificare, considerând vecinătatea punctului de interes.

Pentru fiecare dintre acești pași, există diferenți algoritmi, unii asigurând o "acoperire" mai bună a obiectelor cu puncte de interes, alții concentrându-se pe stabilitatea trăsăturilor determinante sau pe eficiența computațională. Desigur, trebuie realizat un compromis astfel încât să se ajungă la o soluție acceptabilă pentru cât mai multe aplicații. De menționat că algoritmii de localizare și cei de descriere pot fi aleși în mod independent, dar, o analiză a performanțelor nu poate fi realizată decât la nivelul efectului celor 2 pași, în mod secvențial.

Metodele ce au la bază trăsături identificate prin puncte de interes pot fi folosite și în alte aplicații, nu doar cea a recunoașterii. Astfel de algoritmi pot fi aplicați, de exemplu, ca prim pas în reconstruirea unor modele 3D ale obiectelor din imagini sau video, pentru calibrarea aparatelor foto digitale sau pentru crearea de panorame din secvențe de imagini. Prin urmare, sunt de utilitate mare implementările cât mai generale, flexibile, care să poată fi utilizate într-o gamă largă de aplicații sau teste comparative. Lucrarea de față se referă la detaliile unei astfel de implementări.

Capitolul 2 face o scurtă trecere în revistă a cercetărilor realizate în domeniu, punând accentul pe descrierea noțiunilor teoretice și a algoritmilor folosiți.

Capitolul 3 prezintă detaliile legate de proiectarea aplicației propuse, descriind structura detaliată a modulelor unei platforme software pentru prelucrarea fluxurilor de imagini și a submodulelor ce implementează algoritmii pentru detecția de obiecte.

Capitolul 4 detaliază implementarea aplicației construită pe baza platformei proiectate, prezentând atât secvențele de cod care implementează noțiunile teoretice prezentate, cât și problemele apărute pe parcurs. Pentru acestea, se descriu soluțiile propuse și motivația alegerii lor.

Capitolul 5 menționează dependențele diferitelor module ale proiectului de biblioteci externe. De asemenea, sunt prezentate pe scurt testele scrise pentru asigurarea calității programului și a verificării acestuia după compilare și linkeditare.

Capitolul 6 realizează o prezentare a rezultatelor experimentale obținute în urma aplicării algoritmului SURF, în cadrul aplicației descrise în această lucrare. Se evidențiază atât contribuțiile personale în comparație cu alte implementări disponibile, cât și o serie de direcții de studiu.

Capitolul 2

Noțiuni teoretice

2.1 Abordări ale temei în literatura de specialitate

În funcție de restricțiile aplicației practice în care este utilizată, recunoașterea obiectelor poate lua mai multe forme, de la simpla împărțire a imaginii în zone ce pot reprezenta obiecte (segmentare pe bază de culoare), la o recunoaștere completă, ce implică determinarea locației (x, y) a unui obiect, reconstituirea poziționării sale în spațiu (sau 2D în planul imaginii) și recunoașterea denumirii obiectului respectiv pe baza unor cunoștiințe anterioare ale sistemului.

Oricare ar fi gradul de complexitate, la modul general se pune problema ca pornind de la o matrice de pixeli (imaginea), să fie determinată o submulțime a acestora care reprezintă un obiect. Fără a scădea din generalitate, considerăm că obiectul este dat de o regiune contiguă de pixeli din imaginea originală.

O abordare directă a problemei, presupunând că deținem o imagine a obiectului, ar fi căutarea tuturor pixelilor săi într-o altă imagine dată. Îmbunătățiri ale acestei metode, caracterizată de potrivirea unor ”tipare” reprezentând obiectul în scene care îl conțin, au reprezentat începutul cercetării în domeniu (Figura 2.1a). Soluția (**template matching**), în forma ei inițială, este neficientă computațional și sensibilă atât la modificări ale mediului în care dorim să realizăm recunoașterea (luminozitate, reflexii) cât și la ocluzionări parțiale ale obiectelor. Pentru obținerea unei oarecare invariante, a fost propusă corelarea nivelurilor de gri din diverse zone ale imaginii reprezentând obiectul, cu zone din imagini care se presupune că îl conțin. [Ballard and Brown, 1982, Goshtasby et al., 1984] Aceste studii sunt făcute în contextul sistemelor de stereo-vizualizare, unde scena este fotografiată simultan din perspective diferite și se dorește determinarea unor corespondențe între imagini, evitând o calibrare anterioară sau cunoașterea geometriei epipolare a sistemului. Mai recent, există variante care propun modificări ale metodei pentru a o putea rula în timp real [Cole et al., 2004].

Pentru a depăși o parte din problemele metodei anterioare, se pleacă de la observația că pentru recunoașterea unui obiect nu este nevoie de toți pixelii săi, ci doar de o parte din aceștia, ce definesc forma specifică a obiectului sau caracteristici importante ale acestuia. Se realizează o sintetizare a informației din imaginea originală, făcându-se primul pas înspre reprezentarea respectivului obiect într-un mod abstract. Abordarea recunoașterii obiectelor prin potrivirea unor astfel de trăsături abstractive (**feature matching**) este cea de-a doua direcție de cercetare în domeniu. (Figura 2.1b)

Aplicarea algoritmilor de acest tip presupune marcarea trăsăturilor din imagine ca puncte de interes, având ca informație minimală locația, (x_t, y_t) . Există desigur și posibilitatea stocării unor date suplimentare precum orientarea sau scala caracteristicii determinate.

În mod tradițional, trăsăturile alese pentru identificare și potrivire sunt multe, colțuri sau contururi [Cheng and Huang, 1984, Ullman, 1979]. În momentul de față, sunt propuși algoritmi care realizează și identificarea unor alte structuri, precum petele luminoase sau întunecate (eng. blob) [Lowe, 2003, Bay et al., 2006].



Figura 2.1: (a): Potrivire bazată pe tipare (template matching), (b): Potrivire bazată pe trăsături (feature matching)

Avantajul acestei metode este că necesită mai puțină putere de calcul (operând asupra unui număr relativ restrâns de pixeli) și poate fi, prin urmare, aplicată cu ușurință în timp real. În plus, datorită faptului că se lucrează cu o reprezentare intermedieră a obiectului (trăsături), metodele pot fi proiectate pentru a obține un grad ridicat de invarianță la anumiți parametrii de mediu sau la transformări affine aplicate obiectului. Abordarea eșuează dacă nu se reușește o determinare repetabilă și consistentă a trăsăturilor unui obiect în imagini diferite. În acest caz, potrivirea nu are loc și obiectul nu este detectat.

Datorită flexibilității crescute și a rezultatelor foarte bune obținute în practică de către abordarea potrivirii bazate pe trăsături, lucrarea de față utilizează

această metodă pentru recunoașterea obiectelor.

2.2 Identificarea trăsăturilor

În identificarea trăsăturilor, se disting 2 metode, utilizate, cu unele adaptări, în cele mai multe dintre aplicațiile practice curente: Detectorul Harris, și SIFT (Scale Invariant Feature Transform). Aplicația propusă în lucrare utilizează o variantă îmbunătățită a algoritmului SIFT, adaptată pentru procesarea fluxurilor de imagini, în timp real. O parte a ideilor propuse inițial de Harris și Stephens pentru detectorul Harris sunt reluate în algoritmul SIFT, prin urmare considerăm utilă prezentarea ambelor metode.

2.2.1 Detectorul Harris

Prima abordare, propusă de Harris și Stephens, identifică în imagine colțurile și muchiile [Harris and Stephens, 1988]. Cei doi pornesc de la o observație anterioară a lui Moravec, care definește un colț ca fiind un pixel care nu se aseamănă cu pixelii din vecinătatea sa. Astfel, pe o suprafață uniformă, un pixel va avea valori apropiate de cele ale vecinilor săi; pe o muchie, în vecinătatea pixelului se vor identifica modificări mari relativ la valorile vecinilor perpendiculari pe direcția muchiei, dar modificări mici în direcția muchiei. Însă dacă pixelul aparține unei trăsături cu variații în toate direcțiile (un colț), atunci nici una dintre vecinătăți nu va fi similară pixelului. În [Harris and Stephens, 1988], formalizând matematic aceste observații, se definește noțiunea de autocorelație. Funcția de autocorelație măsoară modificările locale ale semnalului 2D (imagină), folosind ferestre deplasate pe distanțe mici în vecinătatea punctului considerat. Fiind dată o deplasare $(\Delta x, \Delta y)$ și un punct (x, y) , funcția de autocorelație este

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x, y)[I(x + \Delta x, y + \Delta y) - I(x, y)]^2 \quad (2.1)$$

unde $w(x, y)$ reprezintă funcția fereastră (și poate fi aleasă ca fiind o funcție nucleu rectangular sau, pentru a reduce influența zgomotului, un nucleu Gaussian) iar $I(\cdot, \cdot)$ este funcția imagine.

Imaginea din fereastra deplasată este aproximată prin dezvoltarea în serie Taylor, trunchiată la primii termeni,

$$I(x + \Delta x, y + \Delta y) = I(x, y) + \begin{bmatrix} I_x & I_y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \quad (2.2)$$

I_x și I_y fiind derivatele parțiale pe direcția x, respectiv y.

Înlocuind 2.2 în 2.1 și considerând Δx și Δy suficient de mici, obținem o ecuație de forma:

$$E(\Delta x, \Delta y) \cong \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} M \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

M fiind o matrice 2×2 calculată din derivatele locale parțiale ale imaginii,

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Matricea M descrie structura locală a imaginii în vecinătatea pixelului considerat.

Fie λ_1, λ_2 valorile proprii ale acestei matrici. Există 3 cazuri care trebuie considerate:

1. Dacă atât λ_1 cât și λ_2 au valori mici, astfel încât funcția de autocorelație este plată (schimbări mici ale lui $E(\Delta x, \Delta y)$ în orice direcție), zona din fereastra considerată este aproximativ uniformă.
2. Dacă o valoare proprie este mare iar celală este mică, astfel încât funcția de autocorelație are forma unei trepte, atunci deplasările ferestrei într-o direcție (de-a lungul treptei) produc modificări mici ale lui E, iar deplasările pe o direcție ortogonală primeia produc modificări mari. Acest lucru indică o muchie.
3. Dacă valorile proprii sunt ambele mari, deplasările în orice direcție vor produce modificări mari ale lui E, indicând un colț.

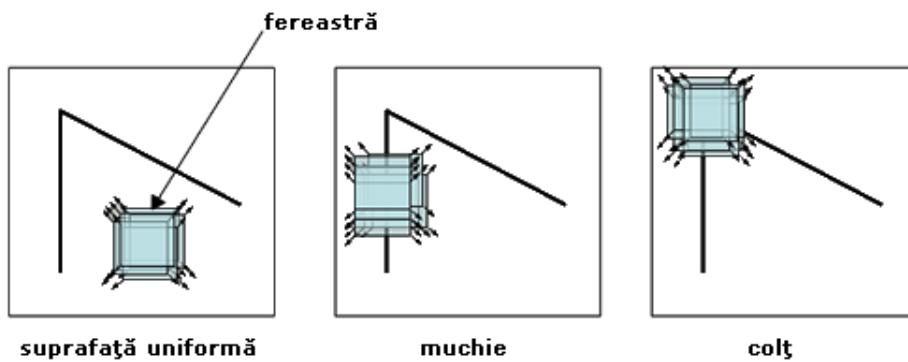


Figura 2.2: Detectorul Harris (muchii și colțuri)

Intuitiv, modul de operare al detectorului Harris este prezentat în Figura 2.2. Performanțele sale au fost analizate în detaliu [C.Schmid et al., 2000]. Concluzia

studiu este că detectorul Harris este unul robust, putând fi aplicat cu succes inclusiv pe imagini afectate de zgomot și fiind invariant la rotații sau schimbări ale luminosității ambientale. Totuși, repetabilitatea rezultatelor sale scade drastic la schimbări ale perspectivei. O altă problemă a detectorului este că nu este invariant la modificările de scală ale obiectelor considerate. Acest lucru poate fi observat cu ușurință în Figura 2.3.

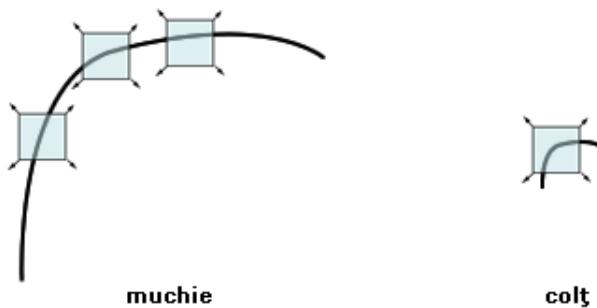


Figura 2.3: Variația rezultatelor detectorului Harris la scalări: modificarea scalei imaginii poate duce la clasificări diferite

Au fost propuse și variante care să fie invariante la scalări (Harris-Laplacian), acestea fiind similare ca abordare cu cel de-al doilea detector important, SIFT.

2.2.2 Detectorul SIFT

SIFT (Scale Invariant Feature Transform) este un algoritm propus de Lowe [Lowe, 2003], care include, ca pas intermediar, detecția unor puncte de interes asimilate unor trăsături de tip "zonă luminoasă" sau "zonă întunecată". Prin construcția algoritmului, aceste zone sunt determinate pentru a fi invariante la scalări, rotații și parțial invariante la modificări ale luminosității și la transformări afine. Metoda aplică o filtrare în cascadă, pentru a asigura calitatea punctelor de interes determinate, dar și pentru a aplica operațiile intensive computațional doar acelor zone care trec unele teste inițiale. Pe lângă determinarea locației punctelor de interes, algoritmul SIFT propune și metode de descriere a acestora în mod individual, astfel încât să poată fi identificate cu probabilitate mare în imagini noi. Practic, fiecărui punct de interes îi este asociat un descriptor (vector caracteristic), calculat pe baza informațiilor imaginii în vecinătatea punctului de interes.

Aceste caracteristici recomandă SIFT ca fiind ideal pentru aplicarea în zona recunoașterii obiectelor [Lowe, 1999]. Pentru aceasta, mai întâi se extrag trăsăturile SIFT pentru un set de imagini de referință ce reprezintă obiectele, stocând descriptorii rezultați într-o bază de date. Unei imagini noi, în care se dorește

identificarea unuia dintre obiectele existente în baza de date, i se aplică același algoritm, iar descriptorii punctelor de interes rezultate sunt comparați individual cu descriptorii din baza de date. Potrivirile între descriptori se fac pe baza distanței Euclidiene între vectori (nu se caută doar potriviri exacte). Totuși, într-o imagine aglomerată, multe trăsături din fundal nu vor avea corespondență în baza de date, dând potriviri false, pe lângă cele corecte. Potrivirile corecte pot fi însă filtrate prin identificarea unor submulțimi de puncte de interes care sunt consistente cu aceeași localizare, scală și orientare a obiectului în noua imagine. Determinarea acestor clustere poate fi realizată eficient folosind transformata Hough [Brown and Lowe, 2002].

Localizarea punctelor de interes

Primul pas în determinarea punctelor de interes SIFT îl reprezintă detectarea locațiilor din imagine care sunt invariante la scalări, prin căutarea trăsăturilor stabile, folosind o funcție de scală cunoscută sub denumirea de spațiu al scalărilor (eng. scale space). Pentru o imagine, spațiul scalărilor este definit de funcția $L(x, y, \sigma)$, obținută prin convoluția unui nucleu Gaussian $G(x, y, \sigma)$ cu imaginea, $I(x, y)$. Pentru a obține scalări diferite, se variază σ :

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y),$$

unde $*$ reprezintă operația de convoluție, iar nucleul Gaussian G este dat de formula:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Pentru a detecta punctele de interes stabile în spațiul scalărilor, Lowe propune determinarea extremelor locale ale funcției "diferență de nucleu Gauss cu scalări diferite", în convoluție cu imaginea, $D(x, y, \sigma)$. Aceasta poate fi calculată din diferența a două scalări separate de un factor k :

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \quad (2.3)$$

$$= L(x, y, k\sigma) - L(x, y, \sigma) \quad (2.4)$$

Există mai multe motive pentru care a fost aleasă această funcție în mod particular. În primul rând, imaginile pentru care se aplică filtrul Gaussian (convoluție), trebuie oricum calculate în procesul de creare al spațiului scalărilor, D calculându-se în mod eficient prin scăderea imaginilor din două scale adiacente. În al doilea rând, diferența nucleelor Gauss (Difference of Gaussian, DOG) aproximează foarte bine Laplacianul Gaussian-ului, $\sigma^2 \nabla^2 G$. S-a demonstrat că extremele acestei funcții reprezintă trăsături foarte stabile ale imaginii, în comparație cu trăsăturile determinate cu alte funcții precum gradientul, Hessian-ul sau detectorul Harris.

Pentru a detecta extremele locale ale funcției D , se realizează o eșantionare a funcției atât spațial (x, y), cât și pentru parametrul de scală (σ). Frecvența aleasă pentru eșantionare reprezintă un compromis între precizia localizării extremelor și puterea de calcul necesară pentru determinarea lor. Astfel, o eșantionare cu frecvență mare duce la costuri mari din punct de vedere computațional, iar o frecvență mică duce la o precizie scăzută a algoritmului.

Fiecare punct eșantionat este comparat cu cei 8 vecini ai săi din imaginea curentă, și cei 9 vecini din scalările adiacente celei curente (Figura 2.4). Punctul este selectat doar dacă este mai mare sau mai mic comparativ cu toți vecinii săi. Această abordare se dovedește eficientă pentru că majoritatea punctelor sunt eliminate după doar câteva comparații.

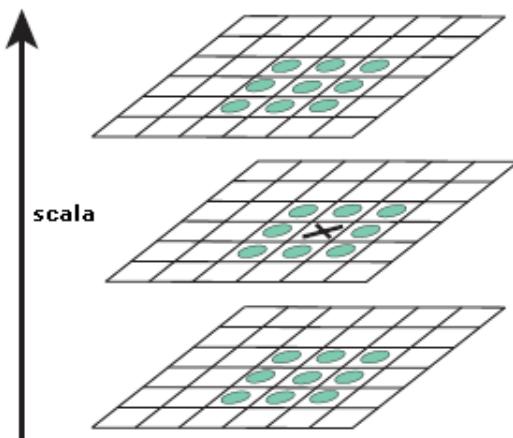


Figura 2.4: *SIFT: Detectarea minimelor și maximelor locale*; punctul central este comparat cu toți vecinii marcați ([Lowe, 2003])

O precizie crescută a localizării punctelor de interes se poate obține folosind o metodă de aproximare a poziționării maximului, prin interpolare. Astfel, se încearcă aproximarea punctelor eșantionate cu o funcție cuadratică, 3D. Practic, se realizează o dezvoltare în serie Taylor până la termenii de grad 2, a funcției $D(x, y, \sigma)$, translată astfel încât punctul eșantionat să fie în origine:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x} \quad (2.5)$$

unde D și derivatele sale sunt evaluate în punctul de eșantionare iar $\mathbf{x} = (x, y, \sigma)^T$ este deplasarea față de acest punct. Localizarea precisă a extremului, $\hat{\mathbf{x}}$ este determinată prin derivarea ecuației 2.5 în raport cu \mathbf{x} și egalarea cu zero, rezultând

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1} \frac{\partial D}{\partial \mathbf{x}} \quad (2.6)$$

Pentru a elimina punctele care sunt maxime locale dar se află într-o regiune cu un contrast slab (fiind prin urmare instabile), se vor reține doar aceleia pentru care $D(\hat{\mathbf{x}})$ este mai mare decât o valoare prag (Lowe alege valoarea de prag 0.03 pentru experimentele sale).

Totuși, pentru o stabilitate crescută, nu e suficientă îndepărțarea trăsăturilor cu un contrast slab. Funcția "diferență de nuclee Gauss" va avea un răspuns puternic de-a lungul muchiilor, chiar dacă locația respectivă este determinată imprecis, sensibilă la zgomotele din imagine. Pentru eliminarea acestor răspunsuri, se folosește o abordare bazată pe o matrice Hessiană 2×2 , calculată în poziția și pentru scara punctului de interes:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \quad (2.7)$$

Derivatele se estimează prin diferențele față de punctele eșantionate din vecinătate. Pentru eliminarea răspunsurilor de-a lungul muchiilor, se impune ca raportul valorilor proprii ale acestei matrici să fie sub o valoare prag (Lowe alege valoarea 10). Pentru că eliminarea se face în funcție de raportul valorilor proprii, nu este necesară calcularea individuală a acestora. În loc, se folosesc determinantul și urma matricii \mathbf{H} . Dacă notăm valorile proprii cu λ_1 și λ_2 , atunci:

$$\begin{aligned} Tr(\mathbf{H}) &= D_{xx} + D_{yy} = \lambda_1 + \lambda_2 \\ Det(\mathbf{H}) &= D_{xx}D_{yy} - (D_{xy})^2 = \lambda_1 \lambda_2 \end{aligned}$$

Considerăm arbitrar $\lambda_1 > \lambda_2$ și notăm raportul valorilor proprii cu r , astfel încât $\lambda_1 = r\lambda_2$. Atunci, avem:

$$\frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} = \frac{(\lambda_1 + \lambda_2)^2}{\lambda_1 \lambda_2} = \frac{(r\lambda_2 + \lambda_2)^2}{r\lambda_2^2} = \frac{(r+1)^2}{r} \quad (2.8)$$

Prin urmare, pentru a impune r ca valoare prag, trebuie verificată doar condiția:

$$\frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} < \frac{(r+1)^2}{r} \quad (2.9)$$

Descrierea punctelor de interes

După stabilirea precisă a locației unei trăsături, se dorește asocierea unui vector caracteristic (descriptor), astfel încât ea să poată fi identificată și în alte imagini.

Primul pas constă în atribuirea unei orientări fiecărui punct de interes, astfel încât descriptorul să poată fi reprezentat relativ la orientarea sa locală. Pentru operațiile care urmează, se alege imaginea filtrată cu nucleu Gaussian având scara cât mai apropiată de cea determinată prin interpolare pentru punctul de interes. Folosind această imagine, se calculează norma și orientarea gradientului într-un număr de puncte din vecinătatea punctului de interes. Valorile obținute sunt

organizate într-o histogramă a orientărilor, cu 36 de intervale. Fiecare vector gradient este adăugat în intervalul corespunzător orientării sale și ponderat cu valoarea normei. Vârfurile din histogramă corespund orientărilor dominante ale gradienților locali. Cel mai mare vârf este ales ca orientare a punctului de interes. Dacă cel de-al doilea vârf al histogramei este comparabil ca mărime, atunci în aceeași poziție din imagine se va crea un al doilea punct de interes, care să aibă orientarea acestui al doilea vârf.

Parametrii de poziție, scală și orientare determinăți până acum stabilesc un sistem de coordonate 2D, local punctului de interes, față de care se realizează descrierea acestuia.

În vecinătatea determinată de sistemul local de coordonate al punctului de interes se realizează o eșantionare, iar în punctele alese se calculează norma și orientarea gradientului, relativ la orientarea punctului de interes (Figura 2.5). Normele sunt ponderate de o funcție Gaussiană, (cercul din figură) cu σ de 1.5 ori mai mare decât dimensiunea vecinătății considerate (în experimente 16×16 pixeli). Vecinătatea este împărțită apoi într-un număr de subregiuni care nu se suprapun (16 regiuni de 4×4 pixeli). Pentru fiecare subregiune, valorile gradienților sunt acumulate într-o histogramă, similară celei folosite anterior. Dacă o histogramă discretizează unghiurile de orientare în 8 valori posibile, descriptorul punctului de interes va conține $4 \times 4 \times 8 = 128$ elemente, obținute prin concatenarea valorilor din toate histogramele. (Figura 2.5).

2.3 Identificarea trăsăturilor în timp real

Dintre cele 2 metode prezentate, SIFT se remarcă datorită invarianței la un număr mare de parametri precum și datorită stabilității punctelor de interes determinate. Totuși, este evident că aplicarea algoritmului SIFT implică un număr mult mai mare de operații în comparație cu alți detectori (Harris). Deoarece majoritatea aplicațiilor îl vor utiliza doar ca pas intermediar, se pune problema unei post-procesări a punctelor de interes (de exemplu, pentru a identifica obiecte) și se dorește ca ansamblul algoritmilor de procesare să ruleze în timp real. În forma prezentată, SIFT poate prelucra aproximativ 5 cadre (de dimensiune 650×315) pe secundă. Prin urmare, se justifică o căutare a unor îmbunătățiri care să determine o scădere a timpului de prelucrare, fără a afecta calitatea rezultatelor finale.

2.3.1 Detectorul SURF

SURF (Speeded-Up Robust Features) este una dintre soluțiile propuse în acest sens, fiind și metoda utilizată de aplicația descrisă în această lucrare. Deoarece majoritatea pașilor urmăți sunt identici cu cei ai algoritmului SIFT, vom prezenta în continuare doar elementele noi pe care le aduce în comparație cu acesta.

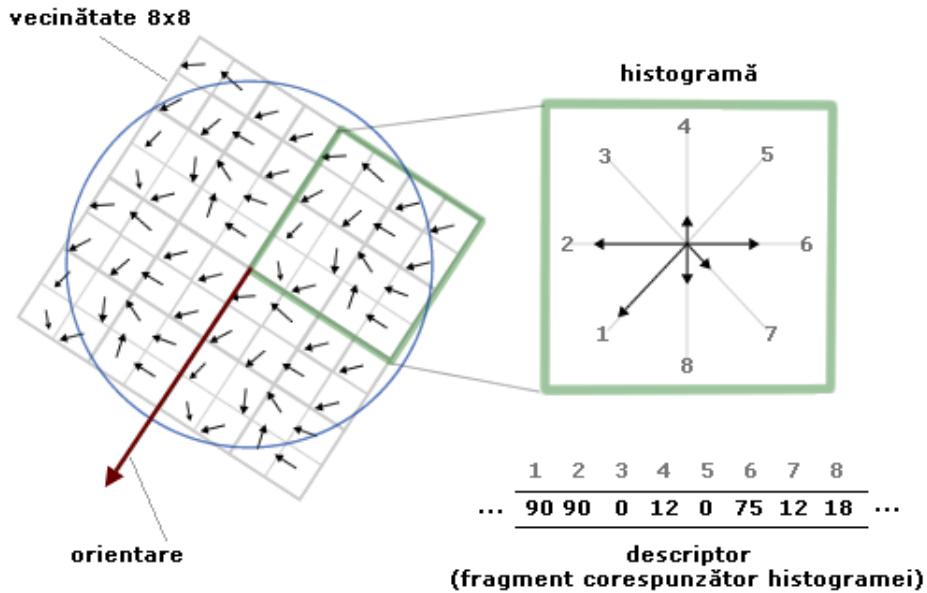


Figura 2.5: *SIFT: Procesul de determinare al descriptorului*; pentru claritatea reprezentării, a fost aleasă o vecinătate 8×8 a punctului de interes. Algoritmul folosește vecinătăți 16×16 .

Principalul pas de procesare al algoritmului SIFT în determinarea punctelor de interes îl reprezintă crearea spațiului scalărilor și convoluția imaginilor cu nuclee Gauss. Filtrarea ulterioară a extremelor se bazează pe matricea Hessiană și pe impunerea unor condiții asupra valorilor proprii ale acesteia (2.7, 2.9). Reamintim că pentru construcția matricii hessiene, Lowe [Lowe, 2003] propune folosirea diferenței de nuclee Gauss pentru a aproxima Laplacianul Gaussian-ului.

Pentru a îmbunătăți performanțele acestor etape ale algoritmului, Bay propune [Bay et al., 2006] utilizarea unor filtre compuse din filtre de mediere (eng. box filter) care să aproximeze tot Laplacianul Gaussian-ului, dar care să poată fi calculate mai eficient. O comparație între aceste filtre și cele originale propuse de Lowe poate fi observată în Figura 2.6. Calculul eficient al convoluției imaginii cu aceste filtre, este realizat prin folosirea imaginilor integrale.

Imagini Integrale

O imagine integrală reprezintă o prelucrare a unei imagini date ca intrare, pentru a permite calculul în timp constant al sumei pixelilor din orice regiune rectangulară. Fiind dată o imagine de intrare I și un punct (x, y) , imaginea integrală este

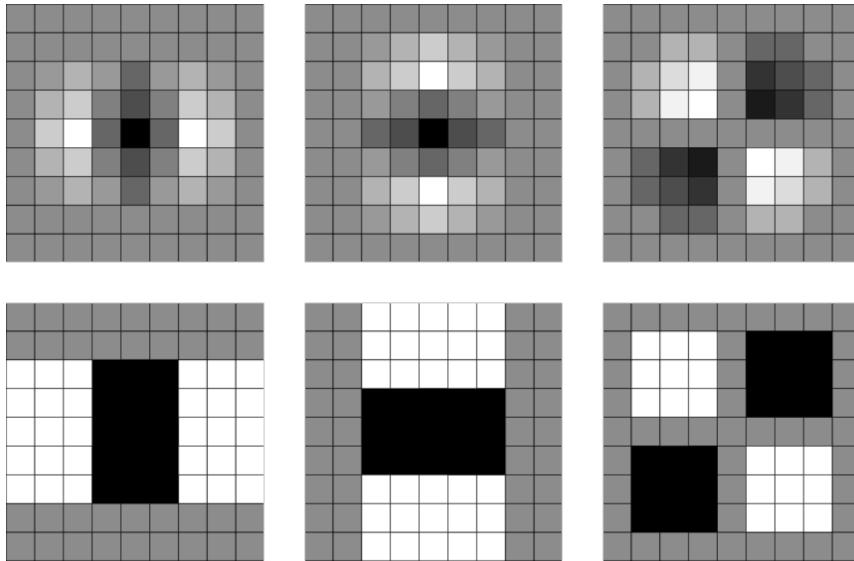


Figura 2.6: SURF: filtre de mediere în comparație cu filtrele obținute prin derivatea de ordin 2 a nucleelor Gauss. Bay notează cu D_{xx} , D_{yy} și D_{xy} aproximările cu filtre de mediere. [Evans, 2009]

calculată cu formula:

$$I_\Sigma(x, y) = \sum_{i=0}^{j \leq x} \sum_{j=0}^{j \leq y} I(x, y) \quad (2.10)$$

Având la dispoziție imaginea integrală, calculul sumei pixelilor dintr-o regiune oarecare se poate realiza prin doar 4 operații matematice. Dacă vom considera regiunea pentru care dorim să calculăm suma ca fiind limitată de colțurile A,B,C și D (precum în Figura 2.7):

$$\sum = A + C - (B + D) \quad (2.11)$$

unde am considerat prin abuz de notație A,B,C și D ca reprezentând intensitățile pixelilor din imaginea integrală în punctul respectiv.

Construirea spațiului scalărilor

În cadrul algoritmului SIFT, imaginea era în mod repetat micșorată, fiind în același timp implicată în operații repetitive de convoluție cu nuclee Gauss având scalări diferite. Ineficiența calculelor realizate astfel provine din necesitatea de a redimensiona imaginea, cât și datorită faptului că unele calcule nu pot fi realizate independent de calculele pentru nivelurile anterioare.

SURF, fiind avantajat de lucrul cu imagini integrale (și putând aplica filtrele asupra imaginii în timp constant indiferent de mărimea filtrului), alege soluția inversă: în locul micșorării imaginii, va fi realizată o mărire progresivă a

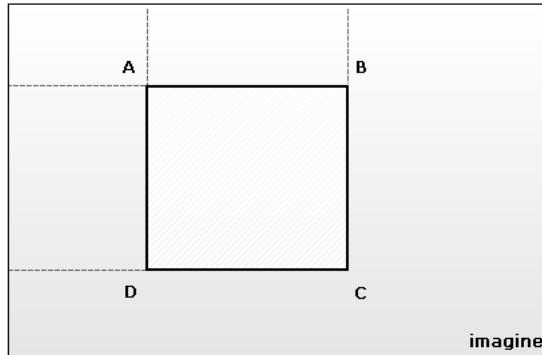


Figura 2.7: Calculul sumei intensității pixelilor folosind imagini integrale.

dimensiunii filtrelor. Această idee simplă dă și posibilitatea calculării mai multor niveluri din spațiul scalărilor în mod simultan (calculele nu mai depind de valori obținute pe niveluri inferioare). O imagine sugestivă care prezintă comparația dintre cele două metode poate fi observată în Figura 2.8. Spațiul scalărilor este

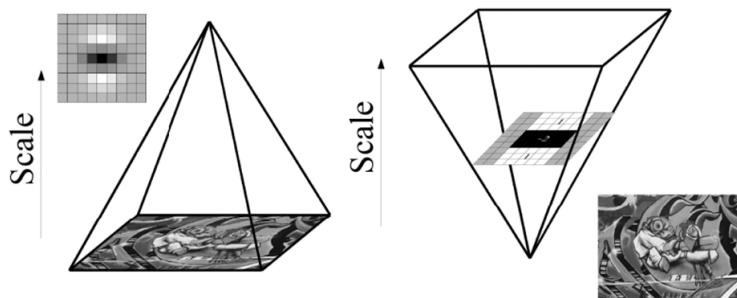


Figura 2.8: Construirea spațiului scalărilor, SURF/SIFT [Evans, 2009]

divizat într-un număr de octave, unde printr-o octavă se înțelege secvența de imagini obținute prin conoluția cu filtre de mediere care acoperă o dublare a scălei (σ). Dimensiunea primului filtru folosit este de 9×9 , iar acesta corespunde cu un filtru Gauss real cu $\sigma_b = 1.2$ [Evans, 2009]. Nivelurile succesive din spațiul scalărilor sunt obținute prin redimensionarea filtrelor. Redimensionarea se face astfel încât filtrul să-și păstreze structura generală, având un pixel central. Datorită acestei redimensionări proporționale, se poate estima scala echivalentă a unui filtru Gaussian care ar produce același efect, folosind formula:

$$\sigma \approx \text{Dim. filtrului curent} \cdot \frac{\sigma_b}{\text{Dim. filtrului corespunzătoare } \sigma_b} \quad (2.12)$$

iar după înlocuirea cu valori,

$$\sigma \approx \text{Dim. filtrului curent} \cdot \frac{1.2}{9} \quad (2.13)$$

Localizarea punctelor de interes

Asemănător ecuațiilor 2.5 și 2.6, algoritmul SURF realizează o interpolare pentru determinarea cu precizie de subpixel a locației extremelor locale. Diferența constă în faptul că aplicarea acestor ecuații se face asupra determinantului Hessian-ului, nu asupra diferenței de nuclee Gauss în convoluție cu imaginea:

$$H(\mathbf{x}) = H + \frac{\partial H^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 H}{\partial \mathbf{x}^2} \mathbf{x} \quad (2.14)$$

Este important de menționat faptul că, din analiza realizată de [Bay et al., 2006], aproximările făcute în scopul de a reduce din calculul necesar pentru determinarea punctelor de interes nu influențează în mod semnificativ rezultatele, acestea fiind comparabile cu cele ale algoritmului SIFT.

Descrierea punctelor de interes

Vectorul caracteristic determinat de SURF pentru fiecare punct de interes măsoără modul în care este distribuită intensitatea pixelilor într-o vecinătate a punctului. Pentru determinarea gradienților locali în mod eficient, sunt folosite wavelet-uri Haar. Acestea sunt filtre extrem de simple, după cum se poate observa și în Figura 2.9



Figura 2.9: Wavelet-uri Haar, dintre care prima este utilizată pentru determinarea gradienților pe direcția x iar a doua pentru cei de pe direcția y

Desigur, descrierea punctului de interes se realizează tot în 2 etape (la fel ca în cazul algoritmului SIFT). Mai întâi, punctului de interes îi este atribuită o orientare, bazată pe orientarea dominantă a gradienților din vecinătate, iar apoi, în funcție de această orientare se calculează un descriptor.

Atribuirea orientării se realizează practic prin calculul răspunsurilor unor filtre Haar de dimensiune 4σ , într-o vecinătate a punctului de interes de dimensiune 6σ . Desigur, σ reprezintă aici scara la care a fost determinat punctul de interes. Răspunsurile sunt calculate în interiorul vecinătății, folosind o eșantionare cu pas σ . Pentru a da o mai mare importanță direcțiilor gradienților din puncte mai apropiate de punctul de interes, rezultatul este ponderat de o funcție Gauss centrală pe punctul de interes. Nucleul Gaussian folosit este ales cu deviația standard egală cu 2.5σ .

O dată ponderate, răspunsurile în direcția x și cele în direcția y pentru fiecare locație eșantionată sunt reprezentate ca puncte într-un spațiu vectorial, cu

x pe abscisă și y pe ordonată. În acest spațiu se realizează determinarea orientării dominante, prin rotirea unei ferestre și însumarea vectorilor din respectiva fereastră, așa cum este prezentat în Figura 2.10.

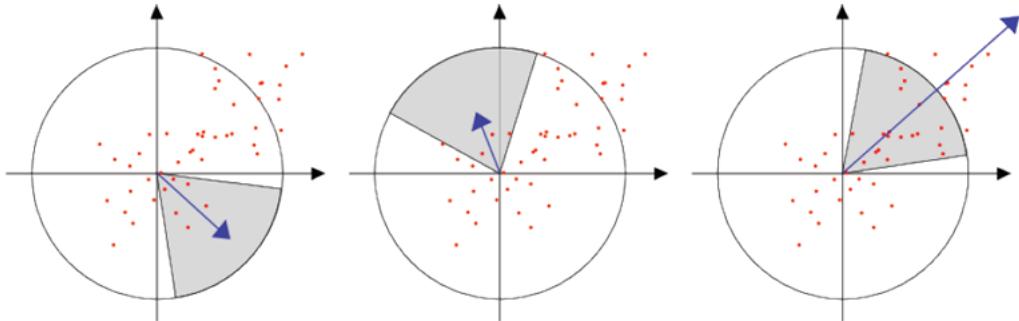


Figura 2.10: SURF: Determinarea orientării pentru un punct de interes [Evans, 2009]

Pentru determinarea vectorului caracteristic, se alege o vecinătate de dimensiune 20σ în jurul punctului de interes, orientată conform orientării punctului. (Toate calculele sunt realizate relativ la această direcție, cu scopul de a obține invarianță la rotații). Spre deosebire de SIFT, în cazul SURF această vecinătate este împărțită în mai puține subregiuni, formându-se o grilă de dimensiune 4×4 . În cadrul fiecarei subregiuni, sunt alese 25 de puncte dispuse la rândul lor pe o grilă regulată, iar în fiecare dintre aceste puncte sunt calculate răspunsurile pe direcțiile x și y ale filtrelor Haar. Pentru fiecare subregiune din cele 16, însumând rezultatele obținute în respectiva zonă, rezultă un vector descriptor de tipul:

$$v_{subregiune} = \left[\sum dx, \sum dy, \sum |dx|, \sum |dy| \right] \quad (2.15)$$

, unde în prima jumătate se rețin informații referitoare la direcția vectorilor, iar în a doua jumătate informații legate de norma lor. Pentru că fiecare dintre cele 16 subregiuni este asociată unui vector cu 4 componente, vectorul descriptor general va fi compus din $16 \times 4 = 64$ componente.

2.3.2 Alegerea parametrilor de rulare ai algoritmilor

În general, algoritmii descriși până acum folosesc o serie de parametri care determină în mod decisiv acuratețea rezultatelor și nivelul de acoperire al unui obiect dintr-o imagine cu puncte de interes. Trebuie realizat un compromis având în vedere dorința de a menține durata rulării algoritmului în limite reduse (pentru prelucrarea în timp real). Astfel, parametrii de rulare de la care se pleacă în lucrarea de față sunt următorii:

Număr de octave calculate	3
Intervale/octavă	4
Frecvență eșantionare pentru determinarea extremelor	variabilă
Prag stabilitate (cf. ecuație 2.9)	5-25

2.4 Potrivirea trăsăturilor

O dată ce a fost aleasă metoda folosită în determinarea punctelor de interes asociate unor trăsături din imagine, se pune în mod natural problema potrivirii acestor puncte între imagini distincte ale aceluiași obiect. Cu alte cuvinte, dorim să aflăm dacă în imaginea sau cadrul (eng. frame) curent există puncte de interes similare cu unele determinate anterior. Pe baza acestei potriviri, putem decide dacă un obiect este prezent sau nu în noua imagine.

În mod tradițional, potrivirea trăsăturilor este realizată prin corelare, utilizând geometria epipolară (geometria sistemului fizic determinat de axul optic al camerei/camerelor foto și obiectele din scena fotografiată) drept constrângere pentru asigurarea consistenței rezultatelor [Zhang et al., 1995, Beardsley et al., 1996].

Această metodă funcționează bine pentru modificări mici ale poziției obiectelor în scenă, însă eşuează în momentul în care diferențele de scală sau de perspectivă sunt mari. Faptul că simpla corelare pe baza geometriei nu este suficientă judecătoare folosirea descriptorilor (vectorilor caracteristici) locali, pentru fiecare punct de interes. Atât timp cât acești descriptori sunt determinați cu invarianțe la parametrii de mediu și poziționare a camerei (precum în cazul algoritmilor de tip SIFT), potrivirea se poate realiza pe baza lor.

Lucrarea de față utilizează o potrivire simplă, bazată pe suma pătratelor diferențelor între 2 vectori caracteristici (SSD, eng. Sum of Squared Differences):

$$SSD = \sum_0^n (f_i - d_i)^2 \quad (2.16)$$

, unde n este dimensiunea vectorilor caracteristici, f este vectorul determinat pentru primul punct de interes iar d este vectorul pentru cel de-al doilea punct. Dacă această diferență este sub o valoare prag, atunci cele două puncte de interes sunt considerate identice (reprezentând aceeași trăsătură a obiectului).

În literatura de specialitate au fost propuse și modele mai complexe, avantajul acestora fiind o îmbunătățire considerabilă a preciziei. În schimb, aplicarea lor presupune cerințe mai ridicate asupra resurselor de calcul.

Metoda aplicată în [Brown and Lowe, 2002] spre exemplu, presupune stabilirea unor grupuri de puncte de interes și identificarea lor în alte imagini. Clusterizarea se realizează pe baza transformatei Hough, iar trăsăturile definite de clustere sunt eficient potrivite între imagini utilizând arbori k-d (eng. k-d trees). Pentru îmbunătățirea preciziei, se aplică algoritmul RANSAC (RANdom SAM-

ple Consensus), care determină o estimare a transformării 2D prin care un set de puncte de interes (pentru obiectele cunoscute) este transformat în alt set (acela al grupărilor de puncte de interes din imaginea curentă).

Capitolul 3

Proiectarea aplicației

3.1 Descrierea aplicației

Aplicația dezvoltată permite selecția unei surse de date, (aparat foto, camera web, un fișier imagine de pe disc) și realizarea unor prelucrări asupra datelor furnizate de către aceasta. Algoritmii aplicați în cazul de față realizează recunoașterea obiectelor. Parametrii de rulare ai algoritmului, precum și formatul rezultatelor întoarse de acesta pot fi controlate dintr-o interfață grafică. Înțial, utilizatorul folosește o serie de imagini statice pentru a defini care sunt obiectele pe care dorește să le identifice; apoi, aplicația poate realiza recunoașterea acestor obiecte în datele furnizate de către o altă sursă de date, chiar dacă în noile imagini obiectele sunt ocluzionate, rotite, sau privite dintr-o altă poziție (între anumite limite).

Proiectul descris are la bază o proiectare detaliată, urmărindu-se o interacțiune corectă a modulelor aplicației, dar și o cuplare slabă între acestea. O astfel de abordare asigură o extensibilitate sporită, sau, în funcție de cerințe, posibilitatea reutilizării diverselor componente în alte aplicații.

3.2 Obiective

Necesitatea unei noi implementări pentru algoritmi precum SIFT sau SURF rezultă din faptul că arhitectura variantelor open-source existente este una destul de greu extensibilă. Chiar și abordările care folosesc biblioteci portabile precum OpenCV* (una dintre implementări este OpenSURF†) sunt greu de adaptat cerințelor unei aplicații industriale sau desktop. Majoritatea implementărilor de până acum au fost realizate în scop academic, urmărindu-se cu precădere testa-

*<http://sourceforge.net/projects/opencvlibrary/>

†<http://code.google.com/p/opensurf1/>

rea performanțelor algoritmilor sau obținerea de informații privind comportarea lor pe seturi de date de intrare diverse. De aici, optimizările au fost realizate în cadrul unor proiecte comerciale. Spre deosebire de acestea, implementarea propusă este open-source, bazată pe o arhitectură ușor extensibilă, modulară, care urmărește sporirea performanțelor și abstractizarea.

Rezultatul este o platformă de prelucrare a imaginilor și a fluxurilor de imagini achiziționate în timp real, care poate fi utilizată și adaptată cerințelor unui număr mare de probleme. Arhitectura propusă pentru această platformă este validată prin implementarea eficientă a unei metode clasice de detecție a obiectelor pe baza punctelor de interes (algoritmul, SURF, a fost prezentat din punct de vedere teoretic în Capitolul 2).

Având în vedere gama largă de aplicații în care o astfel de platformă poate fi utilizată, se dorește asigurarea unei portabilități cât mai bune și a unei flexibilități crescute, astfel încât adaptarea la cerințe sau algoritmi noi să poată fi realizată cât mai ușor. De asemenea, arhitectura aplicației trebuie să permită implementarea eficientă pe arhitecturi multiprocesor sau la nivelul procesoarelor grafice (GPU), prin paralelism de tip SIMD (Single Instruction, Multiple Data: o singură instrucțiune este rulată în paralel pe date diferite). Se urmărește atât paralelismul de granularitate mică (paraleлизarea algoritmului), cât și cel de granularitate mare (algoritmul secvențial să poată folosi în mai multe instanțe ce rulează în paralel pe date diferite).

3.3 Constrângeri

În alegerea arhitecturii generale a aplicației, au fost luate în calcul o serie de constrângeri, prezentate în cele ce urmează, grupate în funcție de natura lor:

1. Constrângeri specifice sistemelor de prelucrare în timp real:
 - Prin proiectarea aplicației, trebuie facilitată o latență mică a sistemului de prelucrare în ansamblul său. Structura generală a claselor și interacțiunile dintre ele (într-o abordare orientată obiect) nu trebuie să introducă tempi de întârziere semnificativi.
 - Având în vedere prima constrângere, se dorește o izolare a procesărilor care au loc asupra unei imagini în cadrul unui singur modul, care să poată fi optimizat din punct de vedere al implementării.
 - În funcție de performanțele sistemului final, se admite ca prezentarea rezultatelor algoritmului aplicat să nu fie realizată tot în timp real. Prin urmare, în măsura în care afișarea informațiilor obținute durează prea mult, se poate renunța la o parte din ele, sau afișarea se poate realiza cu o frecvență mai redusă.

2. Constrângerile legate de formatul datelor de intrare și al datelor de ieșire

- În general, trebuie urmărită o decuplare a algoritmului propriu-zis de structura și formatul particular al datelor de intrare. Spre exemplu, forma algoritmului nu trebuie să depindă de sursa datelor (fișier de pe disc, date preluate de la o camera web sau o cameră digitală firewire), sau de formatul în care acestea sunt codate (bmp, jpeg, tiff etc).
- Datele de ieșire trebuie furnizate într-un format care să poată fi afișat sau stocat pe disc. De asemenea, trebuie asigurată posibilitatea prelucrării suplimentare a rezultatelor, de către alți algoritmi.

3.4 Arhitectura generală a proiectului software

Având în vedere obiectivele și constrângerile prezentate, a fost aleasă o arhitectură pe 4 niveluri, fiecare dintre acestea putând fi dezvoltat sau extins în mod independent de celelalte.

Primul nivel asigură abstractizarea sursei imaginilor și a formatului acestora. În acest mod, nivelurile superioare pot avea o perspectivă uniformă asupra datelor de intrare.

Cel de-al doilea nivel este destinat procesării imaginilor. Pentru acest nivel a fost aleasă o structură de tip pipeline, astfel încât mai mulți algoritmi să poată fi înlanțuiți, ieșirea unuia fiind "conectată" cu intrarea următorului. La acest nivel este izolată partea de procesare propriu-zisă a imaginilor, fiind făcute optimizări ale implementării pentru obținerea unor performanțe crescute.

Al treilea nivel este responsabil cu administrarea resurselor de procesare, realizând legătura dintre interfața grafică (acțiunile utilizatorului) și nivelul de procesare. Aici se poate realiza implementarea paralelismului de granularitate mare (pot fi create mai multe instanțe de procesare, care să ruleze în paralel).

Ultimul nivel asigură prezentarea rezultatelor algoritmilor și interfața cu utilizatorul (GUI - Graphical User Interface).

O imagine de ansamblu asupra celor 4 niveluri poate fi consultată în Figura 3.1.

În cadrul fiecărui nivel, este urmărită o coeziune ridicată a modulelor ce îl compun, acestea fiind organizate funcțional (urmăresc îndeplinirea un scop comun) și operând asupra același tip de date de intrare. De asemenea, comunicarea între niveluri respectă stratificarea acestora. Fiecare nivel comunică predominant cu modulele adiacente lui, prin interfețe bine specificate. Aceste tehnici de proiectare sunt folosite pentru a asigura localizarea modificărilor ce trebuie făcute la apariția unor cerințe noi în aplicație, dar și pentru a obține o structură generală extensibilă și flexibilă. Utilizarea la nivel de modul a unor şabloane de proiectare (eng. design patterns) asigură aceleași proprietăți și pentru structura detaliată a claselor [Gamma et al., 1995].

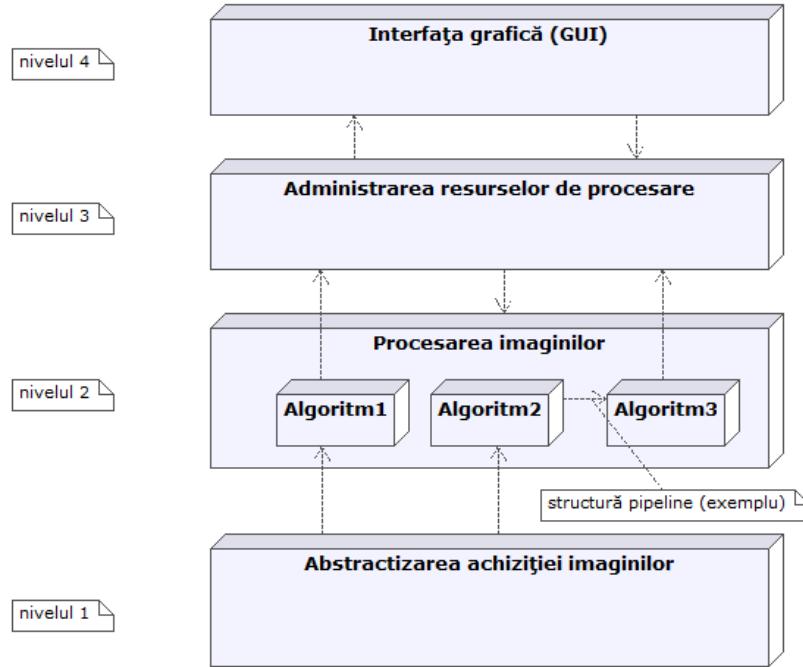


Figura 3.1: Arhitectura generală a aplicației

3.5 Descrierea detaliată a componentelor

3.5.1 Nivelul de abstractizare a achiziției imaginilor

Acest nivel are ca funcție principală preluarea datelor (a imaginilor) din surse externe și furnizarea către aplicație a unui format uniform (un sir de biți reprezentând pixelii). Datorită numărului mare de surse și formate de codare posibile, arhitectura acestui nivel definește 2 interfețe de uz general, *IImageSource* și *IImageConsumer*. Acestea formează o structură clasica producător consumator, proiectată pe baza şablonului Observator (eng. Observer, design pattern). La o sursă de date, se pot asocia un număr de consumatori (algoritmi de prelucrare, în cazul de față), fiecare dintre aceștia fiind "anunțați" atunci când o imagine nouă este disponibilă. O dată anunțați, algoritmii pot copia în bufferele proprii imaginea, urmând să realizeze prelucrările specifice.

Pentru aplicația curentă, sunt propuse două specializări pentru interfața *IImageSource*, și anume *DirectShowCameraSource* și *FileImageSource*. Prima dintre ele realizează achiziția de la dispozitivele compatibile DirectX (aparate foto, camere web, camere digitale), iar cea de-a doua preia imaginile din fișiere stocate pe disc. Prin modul de proiectare, clasa *DirectShowCameraSource* este definită ca singleton (se poate instanția un singur obiect din această clasă). Această soluție arhitecturală este folosită pentru că în aplicație se dorește achiziționarea de

imagini de la o singură sursă video la un moment dat.

Structura detaliată a nivelului de achiziție al imaginilor poate fi observată în Figura 3.2. De asemenea, pentru a vedea cum se integrează acest nivel cu restul modulelor aplicației, puteți consulta Anexa A. Nivelul a fost creat ca un pachet separat, denumit cameraHAL (Hardware Abstraction Layer).

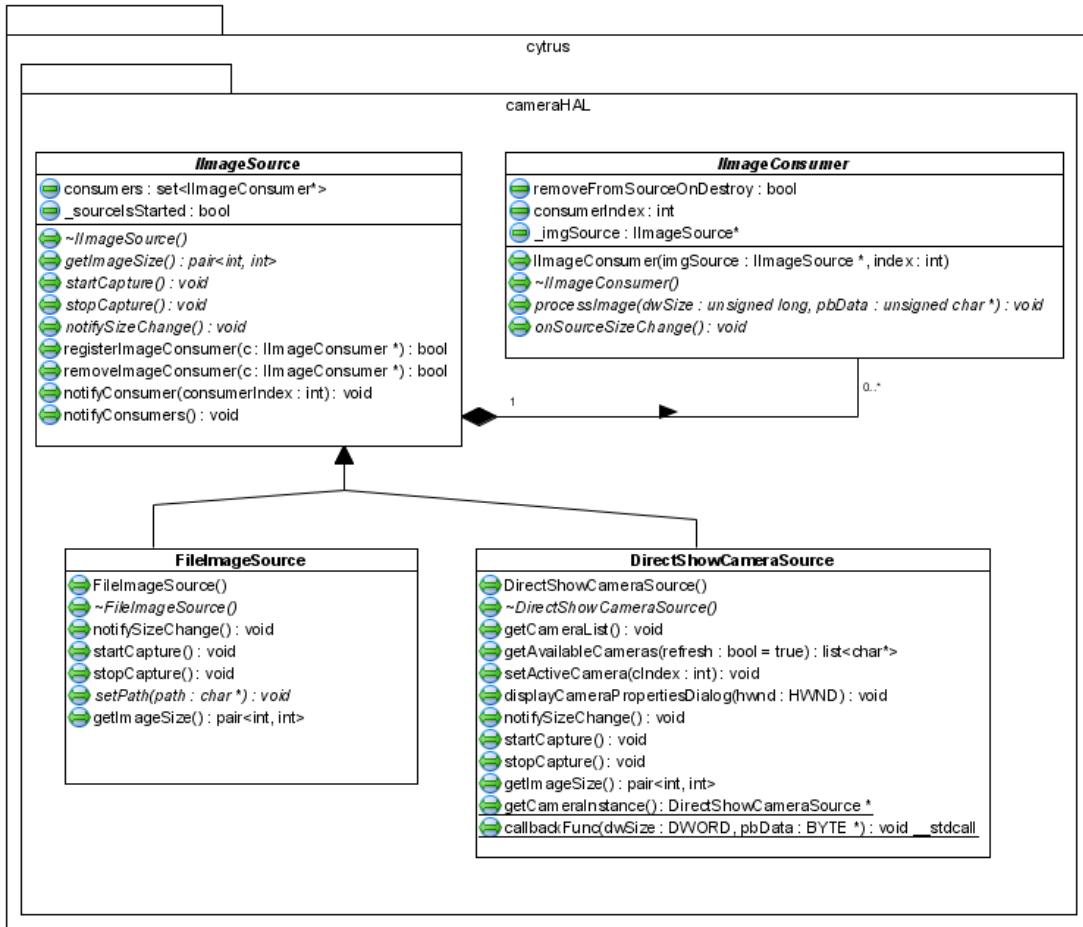


Figura 3.2: Structura nivelului de achiziție a imaginilor

Implementarea interfeței **IImageConsumer** este realizată de către obiecte din nivelul de procesare al imaginilor. Practic, datele sunt transferate între module prin intermediul funcției **IImageConsumer::processImage**, apelată în mod automat de sursă în cadrul procesului de notificare al consumatorilor. Funcția `processImage` îndeplinește astfel rolul funcției cu denumirea clasice "notify" ("anunță") din şablonul de proiectare Observer.

Pentru adăugarea de funcționalități noi la acest nivel (de exemplu, preluarea imaginilor dintr-o bază de date), este necesară construirea unei clase care să moștenească interfața **IImageSource** și să implementeze minimal funcțiile `startCapture`, `stopCapture` și `getImageSize`.

3.5.2 Nivelul de prelucrare a imaginilor

Acesta este nivelul care asigură procesarea efectivă a imaginilor, preluând datele de la una din sursele configurate și trimițând rezultatele prelucrării către nivelurile superioare.

Proiectarea la acest nivel se concentrează pe definirea unei structuri care să descrie în mod flexibil algoritmii de detecție a obiectelor pe baza punctelor de interes, însă, în mod similar, pot fi create clase pentru alți algoritmi de prelucrare a imaginilor. Descrierea succintă a ierarhiei de clase poate fi observată în Figura 3.3. Pentru a vedea cum se integrează acest nivel cu restul modulelor aplicației, puteți consulta Anexa A.

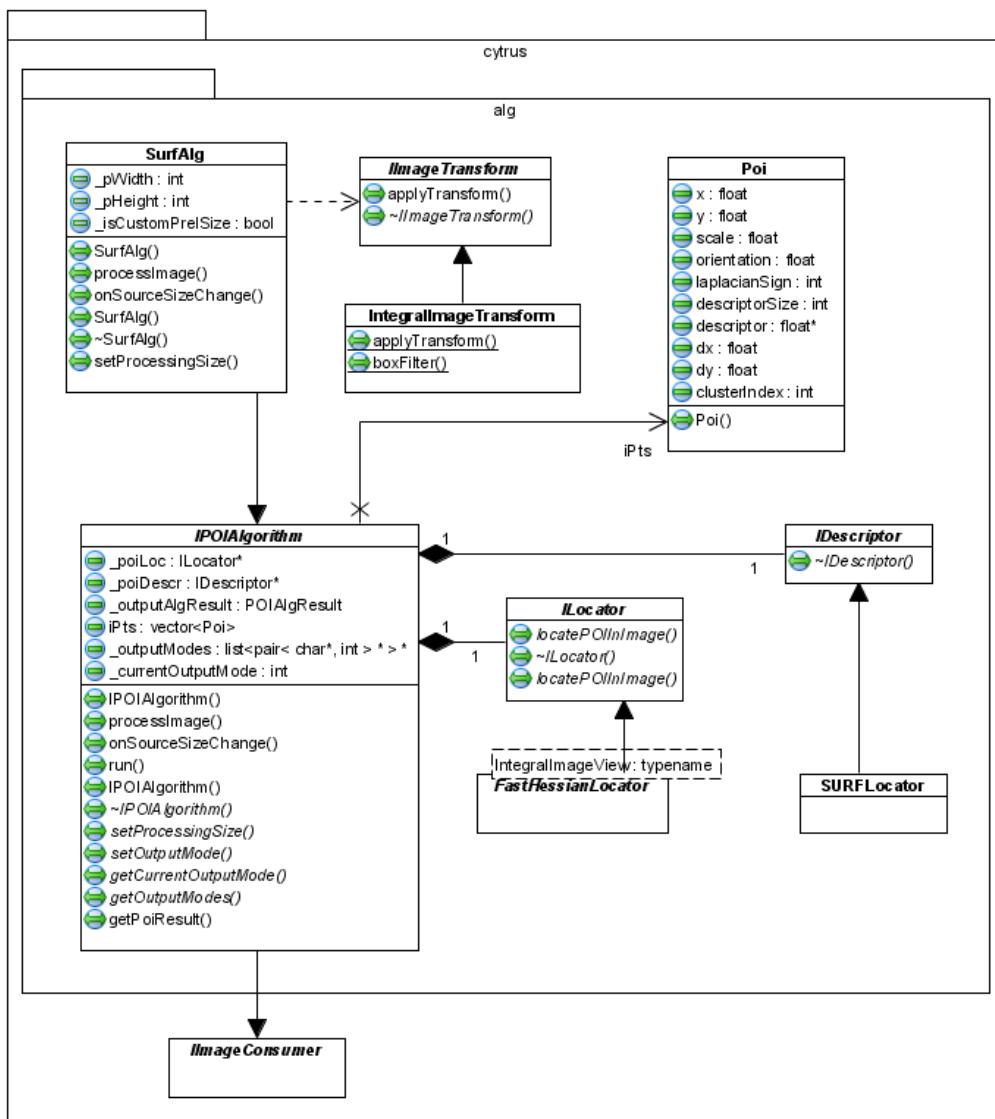


Figura 3.3: Structura nivelului de prelucrare a imaginilor

Algoritmul este definit pornind de la interfața *IPOIAAlgorithm*, care descrie elementele comune pentru toți algoritmii care folosesc puncte de interes (POI - eng. Point of Interest). Interfața moștenește *IImageConsumer* din nivelul anterior (pentru a putea interacționa cu sursele de date). În plus, ea utilizează şablonul de proiectare Strategie (eng. Strategy, design pattern). Folosind această structură, este permisă modificarea ușoară a două etape importante ale algoritmilor și anume procedura de detectie a punctelor de interes (descrișă de interfața *ILocator*) și modalitatea de descriere a acestora (descrișă de interfața *IDescriptor*). Se pot realiza astfel comparații de performanță între diverse metode (spre exemplu, între detectorul Harris și detectarea punctelor de interes cu SIFT), prin simplă schimbare a obiectelor ce aparțin de clase care implementează cele două interfețe. De asemenea, se obține o descriere abstractă a algoritmului, independentă de etapele sale.

Algoritmul utilizat în această lucrare va fi implementat în clasa *SurfAlg*, care implementează *IPOIAAlgorithm* și instanțiază pentru etapele intermediare ale algoritmului obiecte de tipul *FastHessianLocator* (detectia punctelor de interes pe baza calculului rapid al determinantului Hessian-ului) și *SurfLocator* (descrierea punctelor de interes specifică algoritmului SURF).

Pe măsură ce informațiile legate de punctele de interes sunt calculate, ele sunt stocate în instanțe ale clasei *Poi*, iar lista acestora poate fi consultată după terminarea rulării algoritmului. Formatul datelor de ieșire poate fi controlat prin și apeluri ale funcției *IPOIAAlgorithm::setOutputMode*. Inițial, fiecare algoritm definește o listă a modurilor de ieșire suportate (în mod implicit, este definit modul "Normal"). Clasa care implementează propriu-zis algoritmul asigură logica necesară de modificare a ieșirii în funcție de modul de operare activ, selectat de utilizator.

Tot la nivelul de procesare a imaginilor este creată și interfața *IImageTransform*, cu scopul de a organiza ierarhic și a descrie abstract o serie de transformări sau prelucrări ce se aplică imaginii de intrare. Algoritmii pot mai apoi folosi aceste transformări în cadrul operațiilor curente. Algoritmul SURF (de exemplu) utilizează ca pas de preprocessare (pentru reducerea calculelor necesare) transformarea imaginii capturate într-o imagine integrală (pentru care intensitatea unui pixel este egală cu suma intensităților pixelilor din regiunea dreptunghiulară definită de originea imaginii și pixelul respectiv). Această prelucrare este descrisă de clasa *IntegralImageTransform*.

3.5.3 Nivelul de administrare a resurselor de procesare

Acest nivel realizează o legătură dintre componentele descrise până acum (surse de imagini, algoritmi de procesare) și interfața cu utilizatorul.

Aici, denumirea de resursă de procesare nu face referire la o componentă hardware (procesor), ci la secvențe de cod care pot realiza procesarea imaginilor

(algoritmi de procesare). Prin urmare, clasele descrise au rolul de a parta activitățile existente în sistem și de a le distribui către alte clase, care realizează procesarea propriu zisă. Apoi, se dorește colectarea rezultatelor și trimiterea lor pentru afișare sau stocare, către nivelul superior (interfață grafică). Structura generală poate fi observată în Figura 3.4, iar interacțiunea cu restul modulelor aplicației poate fi găsită în Anexa A. Există 2 clase principale, asociate celor 2

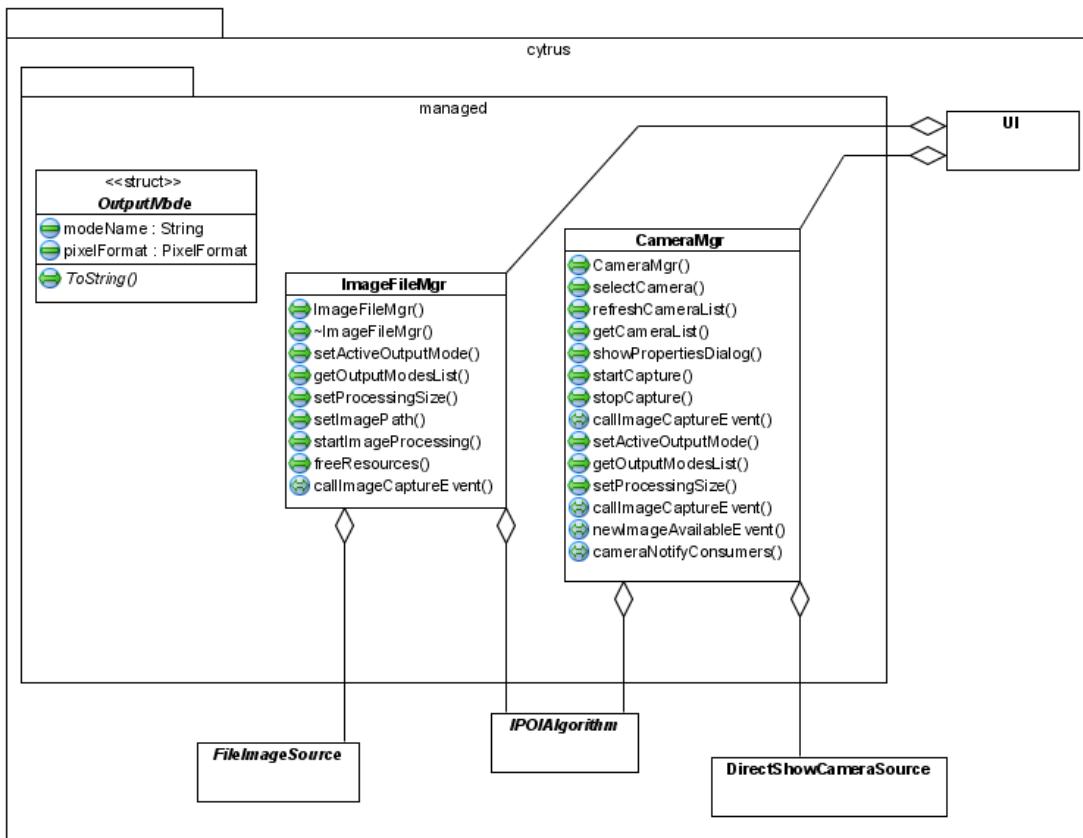


Figura 3.4: Structura nivelului de administrare a resurselor de procesare

tipuri de surse existente în aplicație: CameraMgr realizează administrarea imaginilor provenite de la DirectShowCameraSource, inițializând procesarea datelor de la această sursă și trimiterea rezultatelor către interfață grafică. În mod asemănător, ImageFileMgr administrează imaginile provenite de la FileImageSource.

Asigurarea unui paralelism de granularitate mare poate fi realizată prin instanțierea pe fire de execuție diferite a unui număr de algoritmi (identici) și direcționarea imaginilor venite de la sursa de date către prima instanță liberă (dacă ea există), sau formarea unei cozi de așteptare dacă toate resursele de procesare sunt ocupate. Pentru procesarea în timp real, există și posibilitatea de a renunța la procesarea unei imagini dacă toate resursele sunt ocupate, cu scopul de a păstra timpul de răspuns între anumite limite. Practic, pentru o sursă care furnizează

un flux de date (precum DirectShowCameraSource), ar trebui implementată o structură asemănătoare cu cea din Figura 3.5

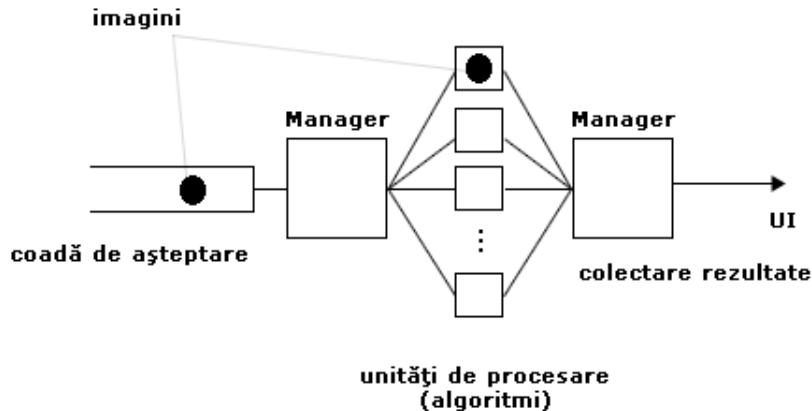


Figura 3.5: Modelul de paralelizare al activităților

3.5.4 Interfața grafică

Acest nivel realizează interacțiunea cu utilizatorul. Din punct de vedere funcțional, trebuie asigurate: controlul asupra diversilor parametrii de rulare ai algoritmului, posibilitatea de alegere a unei surse de imagini și afișarea rezultatelor. De asemenea, în contextul recunoașterii obiectelor, utilizatorului i se va cere să încarce o imagine statică a obiectului și să realizeze (interactiv) o selecție a punctelor de interes ce definesc obiectul din respectiva imagine. Apoi, o dată pornită achiziția în timp real, se vor afișa obiectele recunoscute din fiecare frame, precum și punctele de interes care nu aparțin unor obiecte cunoscute.

Comunicarea cu nivelurile inferioare se realizează prin apel de funcții, declanșate de acțiuni ale utilizatorului. Se urmărește dezvoltarea unui mediu grafic intuitiv, ușor de folosit. Pentru aceasta, vor exista 2 moduri de lucru: unul "live", în care utilizatorul va controla parametrii surselor de date care furnizează un flux de imagini și unul "off-line", în care utilizatorul va putea încărca imagini statice și defini obiecte.

3.6 Metrici de proiectare

Pentru codul managed (primele două niveluri), aceste metrici sunt obținute în mod automat de către mediul de dezvoltare (Visual Studio) pe baza implementării efective a arhitecturii prezentate. Pentru restul nivelurilor, metricile au fost

calculate cu ajutorul aplicației Source Monitor[‡]

Complexitatea ciclomatică este o măsură a numărului de căi liniar independente posibile în cod. Pentru un program structurat, complexitatea ciclomatică se calculează după formula $M = e - n + 2p$ unde e este numărul de muchii din graful asociat structurilor de control din cod, n este numărul de noduri ale acestui graf, iar p este numărul componentelor conexe. Numere mari indică o complexitate ridicată și deci posibilitatea apariției unor erori. Pentru un modul, se calculează suma metricilor pentru fiecare clasă componentă.

Cuplarea claselor se definește ca numărul total de dependențe pe care o anumită clasă le are față de alte tipuri/clase. Pentru un modul, se calculează suma acestor metrii pentru fiecare clasă care îi aparține. În general, un număr mic indică candidați pentru reutilizarea codului.

Modul	Complexitate Ciclomatică		Cuplarea claselor
	sumă	maxim	
Interfață grafică	115	34	102
Nivel de administrare a resurselor	96	8	21
Nivel de procesare a imaginilor	90	14	8
Nivel de achiziție a imaginilor	93	30	4

[‡]<http://www.campwoodsw.com/sourcemonitor.html>

Capitolul 4

Implementarea aplicației

4.1 Descriere generală

Având în vedere necesitatea prelucrării imaginilor într-un timp cât mai scurt, a fost aleasă pentru implementare o soluție hibridă: cod unmanaged (C/C++) și cod managed (.NET C# și C++/CLI). Astfel, modulele pentru care viteza de rulare este critică au fost scrise în C++, iar restul modulelor, inclusiv partea grafică, au fost implementate folosind tehnologii .NET.

O asemenea abordare ar putea ridica semne de întrebare în privința portabilității aplicației pe sisteme de operare precum Linux. Totuși, prin structurarea atentă a componentelor aplicației, se reușește ca o mare parte a codului să poată fi rulată și pe alte sisteme de operare, în afară de Windows. Alegerea bibliotecilor externe utilizate a ținut cont de aceste restricții, fiind urmările soluții care să poată fi portate pe un număr cât mai mare de platforme. Pentru elementele C# care nu apelează funcționalități specifice unui sistem de operare (DirectX), poate fi utilizată platforma Mono.

Împărțirea pe niveluri este reflectată în structura modulară a implementării, deși corespondența nu este perfectă: există niveluri (spre exemplu cel de achiziție a imaginilor) care sunt împărțite în mai multe biblioteci partajate (dll-uri); acest lucru este dictat în principal tot de portabilitate: părțile neportabile ale aplicației sunt separate de cele care pot fi recomilate pe orice sistem de operare. Pentru a obține o portare completă a aplicației pe sisteme de operare GNU/Linux, este necesară rescrierea (parțială) a nivelului de achiziție și a nivelului de administrare a resurselor (dacă se dorește utilizarea firelor de execuție native sistemului de operare), precum și a interfeței grafice. (acestea sunt nivelurile care folosesc cod managed și funcționalități specifice sistemului de operare Windows). O imagine de ansamblu asupra modulelor rezultate în urma implementării poate fi analizată în Figura 4.1

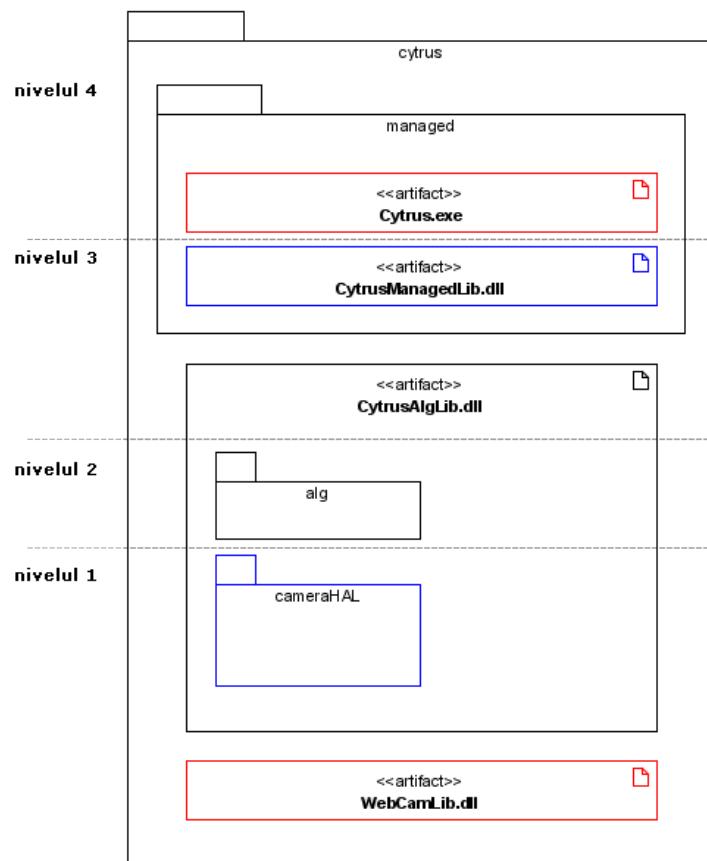


Figura 4.1: *Structura de implementare, prezentând modulele externe rezultate și nivelurile din proiectare.* Cu roșu sunt reprezentate modulele neportabile/care trebuie recrise pentru sisteme te tip GNU/Linux. Cu albastru sunt modulele care ar putea avea nevoie de adaptări pentru a rula cu performanță maximă.

În secțiunile următoare, vor fi prezentate succint bibliotecile externe utilizate în aplicație, împreună cu o motivație a alegerii lor. De asemenea, vor fi trecute în revistă fragmentele cele mai importante de cod și soluțiile tehnice găsite pentru unele dintre problemele apărute pe parcursul dezvoltării.

4.1.1 Identificarea provenienței unui fragment de cod

Toate fișierele care aparțin autorului sunt marcate explicit astfel, printr-un antet corespunzător. Dacă fișierul nu a fost dezvoltat în cadrul acestei lucrări ci de către o persoană terță, acest lucru este marcat în antet. De asemenea, fișierele aparținând unor biblioteci externe care nu au antet nu reprezintă cod scris de către autor. O serie de fragmente de cod (în special legate de interfața grafică) au fost preluate în mod legitim din cadrul unor tutoriale existente pe net. Acestea pot fi recunoscute în general prin apartenența lor la namespace-uri diferite de

ierarhia `cytrus`: . . . , precum și prin antetele ce păstrează informațiile legate de autori.

4.2 Biblioteci externe utilizate

4.2.1 Motivație

Multitudinea formatelor grafice existente la momentul actual, precum și numărul mare de echipamente de achiziție disponibile, fac greoaie dezvoltarea de la zero a codului care să realizeze interfațarea cu acestea. Prin urmare, au fost alese și utilizate două biblioteci grafice externe, care să furnizeze facilități de bază în lucrul cu imagini și camere digitale. Este important de menționat faptul că aceste biblioteci nu conțin rutine care să realizeze recunoașterea obiectelor, sau care să reprezinte subpași ai algoritmului implementat (SURF).

4.2.2 Biblioteci externe

- **WebCamLib** este o bibliotecă C++ de dimensiuni reduse care realizează preluarea de imagini de la dispozitive video compatibile DirectShow/DirectX. A fost dezvoltată în cadrul unui proiect de cercetare al Microsoft, denumit Touchless*. Avantajul acestei biblioteci (utilizabilă doar în cadrul sistemului de operare Windows) în detrimentul unor biblioteci portabile care implementează și această funcționalitate (OpenCV) este numărul mare de echipamente și interfețe suportate. În mod particular, este posibilă achiziționarea de imagini atât de la dispozitive conectate prin USB cât și de la cele conectate prin firewire (camere digitale de rezoluție înaltă). Biblioteca este disponibilă sub o licență open-source (Microsoft Public License).

Fiind disponibilă sub formă de cod sursă, biblioteca este inclusă în proiectul aplicației și compilată ca dll separat, utilizat de modulul în care este inclus nivelul de achiziție a imaginilor (CytrusAlgLib). Au fost realizate o serie de modificări față de versiunea disponibilă pe internet, legate de eliberarea corectă a memoriei.

- **gil[†]** este o bibliotecă C++ dezvoltată de către cei de la Adobe în cadrul proiectelor open-source (dar utilizată inclusiv în proiecte comerciale precum Photoshop). Ea a fost aleasă pentru includere în setul de biblioteci boost[‡]. Aceasta conține un număr semnificativ de funcționalități, fiind una dintre bibliotecile recunoscute pentru portabilitate și calitatea crescută a codului. Boost nu este inclusă în proiectul Visual Studio, fiind una dintre cerințele externe pentru compilarea codului.

*<http://touchless.codeplex.com/>

[†]<http://opensource.adobe.com/wiki/display/gil/Generic+Image+Library>

[‡]<http://www.boost.org/>

Gil este puternic abstractizată, folosind în mod avansat template-urile C++ pentru a obține o performanță crescută (o parte dintre calcule/instancieri sunt realizate la compilare). Biblioteca permite accesul la pixelii unei imagini într-un mod care nu depinde de formatul concret al acesteia. În plus, sunt asigurate structuri de acces (iteratori) care să utilizeze în mod eficient cache-ul procesorului (procesoarelor), astfel încât prelucrarea datelor să fie realizată cât mai rapid.

Gil este folosită în cadrul nivelului de procesare a imaginilor, în algoritmii de prelucrare și detectare a punctelor de interes. Accesul la pixelii imaginilor furnizate de către sursele de date se face cu ajutorul structurilor definite de gil, la fel și conversia între diversele formate de pixeli (rgb32, gray).

Gil nu implementează algoritmi de nivel înalt asupra pixelilor (convoluții, trasări de figuri, recunoașteri de obiecte etc), ci doar primitive de acces la date.

În alegerea unei biblioteci care să furnizeze funcționalități similare au mai fost considerate încă două posibilități, Viga[§] și CImg[¶]. Viga este similară gil din multe puncte de vedere, însă implementează și structuri de nivel înalt, algoritmi de detectie a colțurilor sau segmentări. Multe dintre aceste funcționalități nu ar fi fost folosite în aplicație. CImg furnizează interfețe care nu sunt la fel de bine organizate sau abstractizate în comparație cu Viga sau gil.

4.3 Soluții de implementare

4.3.1 Tipuri utilizate de algoritmul SURF

Aceste tipuri de date sunt definite în aplicație și folosite de către algoritmul de prelucrare a imaginilor. Clasa Poi definește un punct de interes, informațiile fiind completate progresiv, pe măsură ce algoritmul trece prin toate etapele sale (initial, se va realiza doar localizarea punctului, iar mai apoi descrierea acestuia):

```

1  class CYTRUSALGLIB_API Poi{
2      public:
3          float x, y;
4          float scale;
5          float orientation;
6          int laplacianSign;
7          int descriptorSize;
8          float* descriptor;

```

[§]<http://hci.iwr.uni-heidelberg.de/viga/>

[¶]<http://cimg.sourceforge.net/>

```

11   float dx, dy;
12   int clusterIndex;
13
14   Poi(int descrSize=64) : orientation(0), descriptorSize(descrSize)
15   {
16     descriptor=new float[descriptorSize];
17   }
18 };

```

POIAlgResult reprezintă pointer-ul către funcția care va fi apelată după ce rezultatul algoritmului este disponibil. Prototipul funcției are ca prim parametru dimensiunea imaginii; al doilea parametru reprezintă imaginea propriu-zisă, iar ultimul parametru servește la identificarea unității de procesare care a generat rezultatul:

```

typedef void (__stdcall *POIAlgResult)(unsigned long dwSize,
                                         unsigned char* pbData, int index);

```

4.3.2 Imagini Integrale

Obținerea unei imagini integrale din imaginea originală este un exemplu sugestiv al utilizării bibliotecii gil pentru definirea abstractă a algoritmilor. Codul începe prin a defini condițiile care trebuie respectate de către parametrii furnizați pentru template-uri. Dacă în momentul instanțierii clasei IntegralImageTransform aceasta va fi particularizată cu tipuri invalide, erorile vor fi anunțate la compilare.

Pentru că în cadrul codului vom realiza mereu accesul asupra unor vecinătăți ale pixelului curent clar stabilite (vecinul din stanga și cel de sus), aceste locații vor fi stocate într-un cache (deplasările nu vor fi recalculate de fiecare dată). Se parcurg liniile imaginii, iar pixelului curent îi este atribuită suma dintre valoarea sa din imaginea sursă, suma pixelilor de pe rândul curent și cea a pixelilor de deasupra. (cu observația că ultimele două valori sunt deja calculate în locațiile vecine - stanga și sus - din imaginea destinație, datorită iterațiilor anterioare ale buclei). Pentru prima linie (pentru care nu există vecinul de sus), se folosește doar suma pixelilor de pe rând, până la pixelul curent. Practic, codul de mai jos reprezintă implementarea directă a ecuației 2.10.

```

template <typename SrcView, typename DstView>
void IntegralImageTransform::applyTransform(SrcView& src, DstView& dst){

```

```

4     gil_function_requires<ImageViewConcept<SrcView>>();
5     gil_function_requires<MutableImageViewConcept<DstView>>();
6     gil_function_requires<ColorSpacesCompatibleConcept<
7         typename color_space_type<SrcView>::type,
8         typename color_space_type<DstView>::type>>();

9     DstView::xy_locator dst_loc = dst.xy_at(0,1);
10    DstView::xy_locator::cached_location_t above = dst_loc.
11        cache_location(0,-1);

12    //se itereaza peste pixelii din imagine, considerând fiecare
13    //rand al imaginii
14    for (int y=0; y<src.height(); ++y) {
15        typename SrcView::x_iterator src_it = src.row_begin(y);
16        typename DstView::x_iterator dst_it = dst.row_begin(y);

17        unsigned int rowSum=0;
18        for (int x=0; x<src.width(); ++x) {
19            rowSum+=src_it[x];

20            if(y>0){
21                (*dst_it) = rowSum+dst_loc[above];
22                // [...] cod de debugging, omis
23                ++dst_loc.x();
24            }
25            else{
26                (*dst_it) = rowSum;
27            }
28            ++dst_it;
29        }
30        if(y>0){
31            //locatia curenta pentru imaginea destinatie este actualizata
32            // (se trece la inceputul urmatoarei linii)
33            dst_loc+=point2<std::ptrdiff_t>(-dst.width(),1);
34        }
35    }
36}

```

Deși se poate observa că folosirea primitivelor gil reduce din lizibilitatea codului, o dată ce sunt cunoscute funcțiile de bază ale acestei biblioteci, citirea algoritmului devine mult mai ușoară. Avantajul principal este descrierea extrem de generală a operațiilor. Practic, formatul imaginii nu a fost precizat explicit nicăieri în cadrul funcției, fiind furnizat prin intermediul parametrilor template SrcView și DstView (pentru formatul imaginii de intrare, respectiv de ieșire). Folosind această tehnică, algoritmul poate lucra practic cu orice format, fără a trebui realizate modificări în implementarea sa. Tipul propriu-zis al imaginilor este determinat la compilare.

4.3.3 Implementarea algoritmului SURF

Codul prezentat mai jos compune elementele descrise anterior pentru a descrie structura generală a algoritmului de detecție a punctelor de interes. La început, informațiile de culoare din imaginea inițială sunt folosite pentru a obține o imagine cu niveluri de gri. Această imagine este transformată în imaginea integrală (preprocesarea de care are nevoie algoritmul pentru eficiență). Apoi, urmează localizarea punctelor de interes.

Aici este propusă o soluție nouă relativ la parametrii pentru localizare. În articolul în care este descris algoritmul SURF [Bay et al., 2006], sunt sugerăți anumiți parametrii, determinați ca fiind optimi. Totuși, testele din articol au fost realizate pe imagini de dimensiuni relativ mici. Pentru imagini mari, a fost determinat experimental că frecvența de eșantionare spațială și în spațiul scalarilor este prea mare (se determină prea multe puncte de interes, iar viteza algoritmului este prea mică). Pentru imaginile statice prelucrate inițial, acest lucru nu influențează negativ, o bună acoperire cu puncte de interes a obiectelor memorate fiind esențială. Pentru imaginile preluate în timp real însă, se propune o frecvență de eșantionare care să varieze liniar cu dimensiunea imaginii, asigurând o matrice de 120×120 puncte de eșantionare. Această îmbunătățire duce la o creștere a vitezei algoritmului, deoarece se analizează mai puține locații pentru a determina dacă sunt poziții de extrem.

```

void SurfAlg::processImage(unsigned long dwSize, unsigned char*
    pbData){
    // [...] initializari omise
    //crearea view-ului gil din datele obtinute de la ăsurs
    3      rgb8c_view_t myView=interleaved_view(width,height,(const
        rgb8_pixel_t*)pbData,myVal);

    // [...]
    8      //convertirea imaginii la grayscale
    gray8_image_t grImg(width,height);
    gray8_view_t grView=view(grImg);
    copy_pixels(color_converted_view<gray8_pixel_t>(*prelView),
        grView);

    13     //calculul imaginii integrale
    gray32_image_t integral(width,height);
    gray32_view_t integralView = view(integral);
    IntegralImageTransform::applyTransform(grView,integralView);

    18     //localizarea punctelor de interes
    FastHessianLocator<gray32_view_t>* locator=static_cast<
        FastHessianLocator<gray32_view_t>*>(_poiLoc);

```

```

23     if(consumerIndex==−1){ //imagine statică , folosește parametrii
24         speciale
25         locator->setParameters(3,4,10,25.007f);
26     }
27     else{
28         locator->setParameters(3,4,width/120>=2?width/120:2, 5.007f);
29         locator->setSourceIntegralImg(integralView);
30         iPts.clear();
31         locator->locatePOIIInImage(iPts);

32         unsigned long nSize=width*height*3;

33         switch(_currentOutputMode){
34             case 0:
35                 _outputAlgResult(nSize,(unsigned char*)
36                     interleaved_view_get_raw_data(*prelView),
37                     consumerIndex); break;
38             case 1:
39                 //[...] restul modurilor de ieșire omise
40             default:
41                 _outputAlgResult(dwSize,pbData,consumerIndex);
42             }
43         }

```

Implementarea noțiunilor teoretice prezentate în Capitolul 2, în secțiunea ce descrie algoritmul SURF (secțiunea 2.3) este descrisă în secvențele de cod care urmează.

Determinarea Hessian-ului și a extremelor locale

Calculul hessian-ului (2.7) și determinarea locației precise a extremelor locale (ecuația 2.6) sunt implementate în cadrul clasei FastHessian, urmărindu-se pașii de mai jos:

- Calculul determinantului Hessian-ului, pentru toate punctele eșantionate ale imaginilor din spațiul scalărilor. Numărul de scalări precum și frecvența eșantionării sunt parametrii ai algoritmului deciși în momentul rulării acestuia. (funcția buildDet);
- Eliminarea punctelor care nu sunt extreme locale în spațiul scalărilor;
- Determinarea prin interpolare a poziției exacte a extremului în spațiul scalărilor, pentru punctele rămase. (funcția interpolateExtremum, ce implementează ecuația 2.6)

```

template <typename IntegralImageView>
void FastHessianLocator<IntegralImageView>::buildDet() {
    int l, w, b, border, samplingStep;
    float Dxx, Dyy, Dxy, inverse_area;
5     for(int o=0; o<_octaves; o++)
    {
        samplingStep = _sampling * (int)floor(pow(2.0f,o)+0.5f);
        border = border_cache[o];
        for(int i=0; i<_intervals; i++) {
            l = lobe_cache[o*_intervals + i];
            w = 3 * l;
            b = w / 2;
15           inverse_area = 1.0f/(w * w);

            for(int r = border; r < i_height - border; r +=
                samplingStep)
            {
                for(int c = border; c < i_width - border; c +=
                    samplingStep)
20                {
                    Dxx = IntegralImageTransform::boxFilter
                        (_img, r - 1 + 1, c - b, 2*l - 1, w)
                        - IntegralImageTransform::boxFilter
                        (_img, r - 1 + 1, c - 1 / 2, 2*l - 1, 1)*3;
                    Dyy = IntegralImageTransform::boxFilter
                        (_img, r - b, c - 1 + 1, w, 2*l - 1)
                        - IntegralImageTransform::boxFilter
                        (_img, r - 1 / 2, c - 1 + 1, 1, 2*l - 1)*3;
                    Dxy = + IntegralImageTransform::boxFilter
                        (_img, r - 1, c + 1, 1, 1)
                        + IntegralImageTransform::boxFilter
                        (_img, r + 1, c - 1, 1, 1)
                        - IntegralImageTransform::boxFilter
                        (_img, r - 1, c - 1, 1, 1)
                        - IntegralImageTransform::boxFilter
35                   (_img, r + 1, c + 1, 1, 1);

                    //Normalizarea raspunsului filtrelor in raport cu
                    //marimea lor
                    Dxx *= inverse_area;
                    Dyy *= inverse_area;
40                   Dxy *= inverse_area;

                    //Determinarea semnului pentru Laplacian
                    int lap_sign = (Dxx+Dyy >= 0 ? 1 : -1);
45                   //Calcului determinantului Hessian-ului

```

```

template <typename IntegralImageView>
void FastHessianLocator<IntegralImageView>::interpolateExtremum
(std::vector<Poi>& iPts_out, int octv, int intvl, int r, int c)
{
    double xi = 0, xr = 0, xc = 0;
    int i = 0;
    float step = _sampling * (int)floor(pow(2.0f, octv)+0.5f);

    // Preluarea distantelelor intre punctul curent si pozitia
    // determinata prin interpolare a extremului
    interpolateStep(octv, intvl, r, c, &xi, &xr, &xc);

    // Daca punctul este suficient de aproape de extremul calculat
    // prin interpolare, el este considerat punct de interes
    if( fabs(xi) < 0.5 && fabs(xr) < 0.5 && fabs(xc) < 0.5 )
    {
        // Crearea punctului de interes si adaugarea sa in lista de
        // puncte
        Poi ipt;
        ipt.x = (float)(c + step*xc);
        ipt.y = (float)(r + step*xr);
        ipt.scale = (float)((1.2f/9.0f) * (3*(pow(2.0f, octv+1) * (
            intvl+xi+1)+1)));
        ipt.laplacianSign = getLaplacianSign(octv, intvl, c, r);
        iPts_out.push_back(ipt);
    }
}

```

4.3.4 Paralelizarea procesării

Distribuția sarcinilor de lucru pe mai multe fire de execuție se realizează în cadrul codului managed, pe nivelul de administrare a resurselor de procesare. Aici, este

preluată instanța unei surse de date și configurată astfel încât în momentul în care o nouă imagine este disponibilă, să fie rulată o funcție care să realizeze trimiterea acestei imagini către o resursă de prelucrare disponibilă:

```

newImage= gcnew NewImageCallback(this , &CameraMgr::
    newImageAvailableEvent);
2      nigch = GCHandle:: Alloc(newImage);
      IntPtr ipni = Marshal::GetFunctionPointerForDelegate(newImage);
      newImageAvailable = static_cast<NewImageAvailableCallback>(ipni
        .ToPointer());
      cs=DirectShowCameraSource::getCameraInstance( newImageAvailable)
        ;

```

Funcția CameraMgr::newImageAvailableEvent consultă dacă există fire de execuție libere (dintr-un număr de fire de execuție inițializate la început). Practic, soluția .NET folosește un ThreadPool pentru administrarea procesării concurante:

```

void CameraMgr:: newImageAvailableEvent () {
    //rulare asincrona:
    int workerThreads;
    int completionPortThreads;
5      ThreadPool:: GetAvailableThreads(workerThreads ,
        completionPortThreads);
    if(workerThreads>0) //renunta la frame daca nu exista fire de
        executie disponibile , pentru a asigura un timp de raspuns
        mic (si a nu forma o coada de asteptare care sa introduca
        intarzieri
        ThreadPool:: QueueUserWorkItem(gcnew WaitCallback(this ,
            CameraMgr:: cameraNotifyConsumers));
    }

```

În continuare, atunci când va fi rulată (asincron), funcția cameraNotifyConsumers va apela pentru sursa de date curentă funcția notifyConsumer(index) (pentru că sursa are atașați ca observatori toate firele de execuție care realizează procesarea, nu trebuie anunțat de prezența unei noi imagini decât acel fir de execuție care a fost identificat ca fiind liber). Soluția tehnică de implementare menține un dicționar în care au fost trecute intrări de forma <identificator thread -> index consumator>, maparea thread/algoritm de procesare fiind de 1 la 1.

Apelul notifyConsumer(index) va declanșa mai apoi prelucrarea propriu-zisă a imaginii. (și execuția algoritmului SURF).

Această soluție a fost aleasă pentru folosirea thread-urilor native Win32 (și asigurarea performanței). Codul din zona de administrare a resurselor de procesare este unul foarte important în funcționarea corectă (multi-threading) a aplicației. În plus, la acest nivel se realizează și interoperabilitatea dintre diversele limbaje de programare utilizate (fiind scrise în cod managed, clasele trebuie să preia și să trimită informații de la/către funcții sau clase din codul unmanaged). Ca limbaj de programare .NET este folosit C++/CLI, care este un limbaj dezvoltat de către cei de la Microsoft exact în scopul de a facilita o interoperabilitate COM(C++)/C#(sau VB) cât mai bună. C++/CLI permite utilizarea tuturor tipurilor C++ (inclusiv a bibliotecii STL), dar și a tipurilor platformei .NET. Totuși, transferul de date între modulele managed și cele native se realizează cu o penalizare de performanță. De aceea, soluția găsită minimizează numărul de treceri managed/nativ necesare pentru o imagine dată (o singură trecere; procesul de transfer al datelor managed/nativ poartă numele de Marshalling).

4.4 Probleme apărute în dezvoltare. Soluții propuse

Majoritatea problemelor apărute în dezvoltarea aplicației țin de faptul că este necesară prelucrarea și afișarea rezultatelor în timp real. Astfel, unele zone din cod trebuie optimizate sau uneori trebuie alese abordări care nu sunt evidente inițial.

4.4.1 Rularea pe mai multe fire de execuție, fără a utiliza primitive de sincronizare

În general, atunci când se lucrează cu mai multe fire de execuție care rulează în paralel, este necesară folosirea unor primitive de sincronizare (semafoare, mutex-uri) pentru a asigura corectitudinea rezultatelor. Totuși, folosirea acestor primitive introduce zone de cod (zone critice), care nu pot fi rulate decât secvențial. Astfel, rezultă o scădere a performanțelor algoritmului.

Inițial, soluția aleasă pentru rularea algoritmului implica existența unei singure instanțe a algoritmului (SurfAlg), care era rulată pe mai multe fire de execuție simultan. Datele fiind partajate între thread-uri, exista nevoie folosirii de mutex-uri în momentul accesului unei funcții la buffer-ul imagine. Totuși, accesul la imagine se realizează pe tot parcursul algoritmului, în foarte multe zone de cod. Definirea de regiuni critice pentru toate aceste zone ar fi dus chiar la o scădere a performanțelor comparativ cu rularea secvențială.

Rezolvarea problemei s-a realizat printr-un compromis de memorie: se vor utiliza mai multe instanțe ale algoritmului (fiecare cu zona proprie de memorie), iar la rularea aplicației, se va atribui procesarea primei instanțe libere dintre cele inițializate. Astfel, este eliminată complet necesitatea folosirii regiunilor critice pentru algoritm (datele nu mai sunt partajate).

Desigur, regiuni critice sunt totuși folosite în program, în special în zona de afișare a rezultatelor, unde sunt primite răspunsurile de la toate instanțele de procesare. (într-o singură zonă de memorie).

4.4.2 Afișarea în interfața grafică a unui număr foarte mare de obiecte

La construirea interfeței grafice a fost folosită tehnologia Microsoft WPF (Windows Presentation Foundation). Deși această tehnologie permite folosirea posibilităților plăcii grafice în interfață (accelerare grafică pentru animații de exemplu), există unele probleme de performanță atunci când pe suprafața vizibilă a ferestrei trebuie desenate foarte multe elemente.

Acesta este și cazul aplicației prezentate în lucrare, deoarece se dorește afișarea punctelor de interes pentru fiecare imagine. Pentru o imagine de dimensiuni 320×240 sunt identificate în mod tipic în jur de 2000 de puncte de interes. Afișarea acestora în mod clasic (prin crearea de obiecte în interfață), duce la un consum mare de memorie, o viteza scăzută sau chiar blocarea aplicației (din cauza timpului mare petrecut pentru randarea rezultatelor).

Au fost luate în calcul mai multe soluții, iar cea la care s-a ajuns constă în definirea unui singur obiect (de tip grafic), pentru toate punctele de interes, urmată de desenarea în cadrul acestui obiect a unor elipse, la coordonatele determinate prin algoritm. Obiectul folosit, StreamGeometry, este optimizat în scopul randării rapide a unui număr mare de obiecte. Pentru că nu mai este creat în memorie câte un obiect separat pentru fiecare element grafic, viteza de randare crește. De asemenea, este invalidată prin cod procesarea evenimentelor pentru punctele de interes afișate (acest lucru îmbunătățește în mod semnificativ performanța):

```

protected override void OnRender(DrawingContext drawingContext)
{
    Rect adornedElementRect = new Rect(this.AdornedElement.
        DesiredSize);
    Size renderedImgSize=((Image)base.AdornedElement).
        RenderSize;
    SolidColorBrush renderBrush = new SolidColorBrush(
        Colors.Black);
    renderBrush.Opacity = 0.7;
    renderBrush.Freeze();
    Pen renderPen = new Pen(new SolidColorBrush(Colors.
        White), 1);
    double renderRadius = 2.0;
    renderPen.Freeze();

    StreamGeometry geometry = new StreamGeometry();
    geometry.FillRule = FillRule.EvenOdd;
    Size mySize=new Size(renderRadius, renderRadius);
}

```

```

17      using (StreamGeometryContext ctx = geometry.Open())
18      {
19          foreach (Poi_m p in _poiList)
20          {
21              //actualizarea pozitiei
22              double relPosX = p.X * renderedImgSize.Width /
23                  _captureSize.Width;
24              double relPosY = p.Y * renderedImgSize.Height /
25                  _captureSize.Height;
26              Point pct = new Point(relPosX - renderRadius,
27                  relPosY);
28              ctx.BeginFigure(pct, true, true);
29              ctx.ArcTo(new Point(relPosX + renderRadius,
30                  relPosY), mySize, 0.0, false, SweepDirection.
31                  Clockwise, true, true);
32              ctx.ArcTo(new Point(relPosX - renderRadius,
33                  relPosY), mySize, 0.0, false,
34                  SweepDirection.Clockwise, true, true);
35          }
36      }
37      geometry.Freeze(); // se precizeaza ca nu vor mai fi
afcute modificari asupra geometriei
38      drawingContext.DrawGeometry(renderBrush, renderPen,
39          geometry);
40  }

```

4.4.3 Deadlock-uri în codul DirectShow

Inițial, procesarea imaginii era realizată sincron, în cadrul funcției de callback definite de DirectShow (funcția este apelată automat de către DirectShow în momentul în care o nouă imagine este disponibilă). Acest lucru presupune o durată destul de mare a callback-ului, lucru care luat individual nu produce probleme. Acestea apar însă dacă în cadrul callback-ului (sau a funcțiilor apelate succesiiv din el), se folosesc instrucțiuni/funcții DirectShow. Documentația Microsoft atenționează că, mai ales dacă respectivul cod conține regiuni critice, există posibilitatea apariției unor deadlock-uri (interblocarea firelor de execuție), ducând la blocarea completă a aplicației.

Deși codul scris explicit nu apela și alte instrucțiuni DirectShow în cadrul callback-ului, tehnologia WPF folosește primitivele DirectShow în cadrul randării. Prin urmare, aplicația avea momente în care apăreau blocaje (deadlock-uri)

Pentru a rezolva această problemă (după ce a fost identificată cauza), a fost modificată funcția de callback din sursa de imagini DirectShowCameraSource. Astfel, buffer-ul DirectShow este copiat local, iar procesarea suplimentară este atribuită unui nou thread, nu tot aceluia de pe care s-a realizat apelul callback-

ului. Astfel, se permite terminarea foarte rapidă a funcției callback, și se elimină deadlock-urile ce apar colateral în codul DirectShow:

```

void __stdcall DirectShowCameraSource :: callbackFunc (DWORD dwSize ,
    BYTE* pbData) {
    if (imageDataSize!=dwSize) {
        imageDataSize=dwSize;
        imageData=(BYTE*) realloc ((void*) imageData , sizeof(BYTE)*
            dwSize);
        if (imageData==NULL) exit (1);
    }
    memcpy_s((void*) imageData , sizeof(BYTE)*dwSize , pbData , sizeof(
        BYTE)*dwSize);
    signalNewImageAvailable ();
}

```

4.4.4 Ordinea obținerii rezultatelor procesării

Având în vedere că se lucrează pe mai multe fire de execuție, se pune în mod natural întrebarea legată de ordinea în care vor fi afișate rezultatele procesării. Oordonare precisă a cadrelor ar presupune menținerea pentru fiecare cadru a unui timp al apariției în sistemul de prelucrare și afișarea rezultatelor pe baza acestui timp. Nici această soluție nu este una perfectă, deoarece cadre apărute anterior în sistem pot să fie terminate de procesat după ce un alt cadru (apărut după ele) a fost deja prelucrat și afișat. De asemenea, ar fi necesare o serie de regiuni critice, care ar scădea din performanța algoritmului.

Datorită faptului că ordinea de afișare nu este un element foarte important în rularea aplicației, a fost acceptată prin implementare posibilitatea ca frame-urile să nu fie afișate către utilizator exact în ordinea în care ele au fost capturate de către cameră. Totuși, chiar dacă acest eveniment are loc, rularea algoritmului și recunoașterea obiectelor nu au de suferit. În practică, datorită duratei aproximativ constantă de prelucrare a fiecărui frame (obținută de algoritm), nu apar inversiuni ale ordinii de afișare.

4.5 Interfața cu utilizatorul

Interfața grafică este implementată folosind tehnologia WPF și permite accesul facil la toate funcțiile aplicației. Astfel, utilizatorul poate modifica parametrii de rulare ai algoritmului, poate modifica sursa de unde sunt achiziționate imaginile în timp real și poate defini interactiv obiecte pe imagini statice, încărcate de pe disc.

Există 3 zone mai importante în interfață, ele putând fi observate în Figura 4.2.

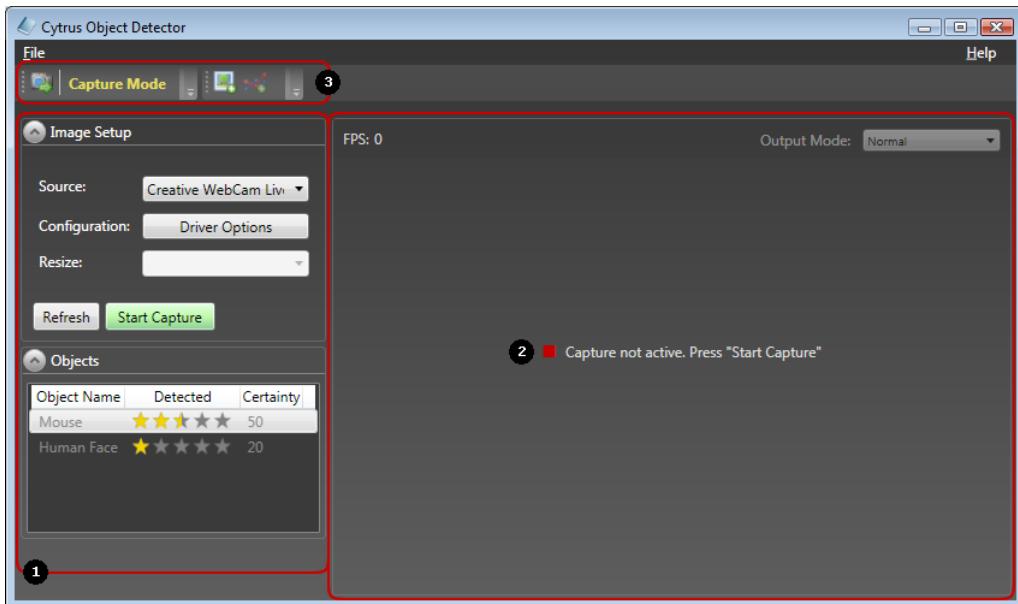


Figura 4.2: Interfața grafică, la pornirea aplicației

Zona 1 conține controale care permit utilizatorului să realizeze configurarea sursei de la care se va realiza captura imaginii. Zona de selecție "Source" conține o listă a dispozitivelor disponibile în sistem (recunoscute de către sistemul de operare). Butonul "Driver Options" afișează fereastra de configurare specifică fiecărui dispozitiv (implementată de către driver). Opțiunea "Resize" devine activă după ce începe captura imaginilor, dând posibilitatea efectuării procesărilor pe versiuni redimensionate ale imaginii capture. Folosîtă corect, această opțiune permite obținerea unor viteze mari de prelucrare. Totuși, redimensionarea unei imagini mari durează timp și presupune o parcurgere a imaginii originale pixel cu pixel. Prin urmare, dacă după reducerea dimensiunii imaginea nu este suficient de mică, costul suplimentar al redimensionării va avea efectul contrar al încetinirii algoritmului.

Zona 2 are rolul afișării imaginilor capture și a rezultatelor rulării algoritmului. O măsură a performanțelor obținute este dată de indicatorul FPS (care afișează o medie a numărului de frame-uri care au fost procesate pe secundă). "Output Mode" permite selectarea modului de operare al algoritmului. Dacă algoritmul implementează funcționalitatea afișării unor rezultate intermediare, atunci rezultatele etapelor intermediare vor putea fi selectate și vizualizate aici). În funcție de algoritmul care rulează, este posibil ca această selecție să afecteze modul de operare al algoritmului, sau doar imaginile trimise de acesta la ieșire. În cazul algoritmului SURF implementat, există 3 opțiuni:

1. Normal - Imaginea capturată este afișată nemonificată, iar peste ea se realizează suprapunerea unui strat transparent, care conține punctele de interes detectate.
2. Grayscale - Zona de afișare conține imaginea în niveluri de gri, obținută de algoritm ca pas intermediar al preprocesării. Punctele de interes sunt în continuare suprapuse pe imagine.
3. Integral - Se afișează imaginea integrală calculată, iar valorile intensităților pixelilor sunt scalate în intervalul [0 255].

Zona 3 conține o bară de instrumente, permitând trecerea între cele două moduri ale aplicației: Capture Mode, în care interacțiunea cu utilizatorul după pornirea capturii este minimală, și Object Mode, în care utilizatorul îi sunt puse la dispoziție controale pentru definirea de obiecte. Modul în care se pot defini obiectele poate fi analizat în Figura 4.3. Pentru definirea unui obiect, după

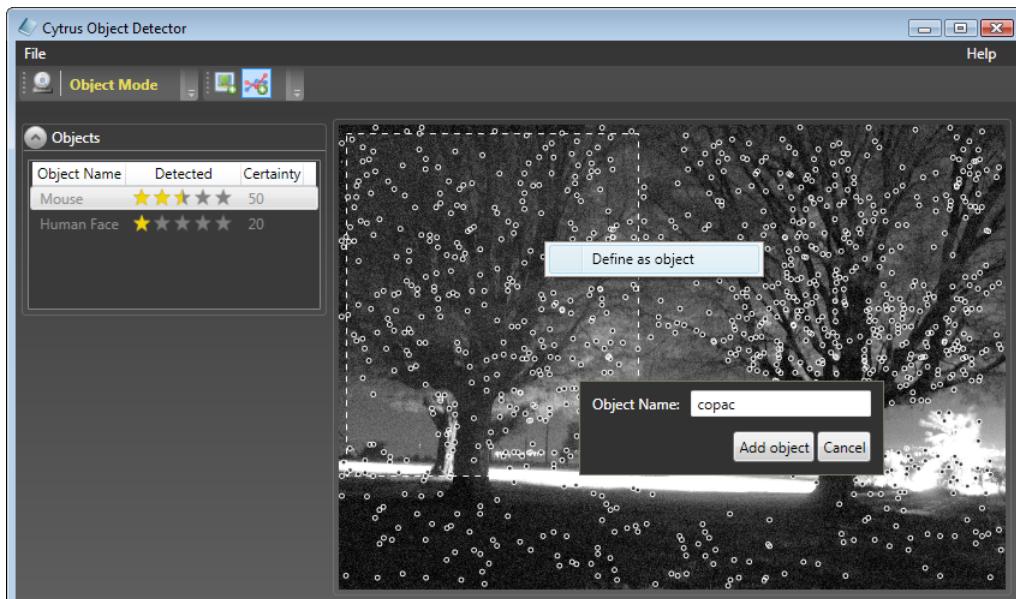


Figura 4.3: Interfața grafică, modul de definire al obiectelor

ce imaginea statică a fost încărcată, se selectează din bara de instrumente butonul "Define Objects" () și se marchează o zonă rectangulară care încadrează obiectul. Pentru a memora respectivul obiect, se dă click dreapta pe zona rectangulară, selectând apoi "Define as object". În fereastra care apare după această acțiune, se va introduce numele obiectului și se va apăsa butonul "Add object".

Capitolul 5

Compilarea și testarea aplicației

Deoarece singura dependență pentru rularea cu succes a aplicației pe sistemul de operare Microsoft Windows este legată de prezența versiunii 3.5 a platformei .NET (.NET Framework 3.5), s-a considerat că nu este necesar un program de instalare separat. Aplicația poate fi lansată în execuție folosind fișierele existente în directorul /bin. Totuși, pentru a folosi ultima versiune a programului și pentru a obține performanțe maxime, este recomandată compilarea din codul sursă.

Codul aplicației este pus la dispoziție sub licența GNU GPL v3, prin urmare modificarea și redistribuirea sa este posibilă în condițiile în care sunt păstrate toate informațiile legate de autor și copyright*.

Biblioteca WebCamLib (WebCamLib.dll) conține cod sub licența Microsoft Public License (Ms-PL), incompatibilă GPL v3. Având în vedere că această componentă este de sine stătătoare, linkeditată dinamic (dll) și înlocuibilă ca modul cu alte biblioteci care oferă funcționalități identice, ea nu va fi considerată ca făcând parte din aplicație.

Ultima versiune a aplicației poate fi descărcată utilizând un browser web de la adresa <http://github.com/luciancarata/b.sc-work/tree/master>, sau utilizând git, un sistem distribuit de control al versiunilor, de la adresa <git://github.com/luciancarata/b.sc-work.git>.

5.1 Compilarea aplicației

5.1.1 Dependențe

Există două librării externe (neincluse în ierarhia directoarelor proiectului), care trebuie să fie prezente pentru a putea compila aplicația cu succes:

*<http://www.gnu.org/licenses/gpl-3.0.txt>

- Windows Platform SDK - conține fișierele antet pentru funcțiile DirectShow utilizate de către modulul de captură a imaginilor; este disponibil pentru descărcare și instalare la adresa <http://www.microsoft.com/downloads/>
- Boost - este utilizat în special modulul GIL, pentru accesul uniform și eficient al algoritmilor de procesare la pixelii unei imagini. Calea către aceste librării trebuie modificată pentru proiectul Visual Studio denumit CytrusAlgLib. La directoarele în care compilatorul va căuta fișiere de tip include, trebuie adăugată și calea către librăriile Boost. Boost poate fi descărcat de la adresa <http://www.boost.org/>.

De asemenea, părți ale codului sursă depind de biblioteca jpeglib, care este inclusă în structura directoarelor proiectului, pentru linkeditate statică. Fișierele header utilizate în proiect se găsesc în /include/jpeglib iar librăria precompilată se găsește în /lib. Acestea sunt prezente doar pentru a limita numărul dependențelor externe. Este recomandată utilizarea ultimei versiuni a bibliotecii, disponibilă la adresa: <ftp://ftp.uu.net/graphics/jpeg/>.

Compilarea aplicației a fost testată cu succes în mediul de dezvoltare Visual Studio 2008 SP2, o soluție având majoritatea setărilor și dependențelor între proiecte setate fiind disponibilă în directorul /src.

5.2 Testarea unităților

Testarea unităților este una dintre cele mai importante metode de asigurare a calității codului unei aplicații. Pentru codul prezentat în această lucrare, au fost considerate o serie de teste de tip cutie neagră (eng. black-box), cu scopul de a identifica eventualele probleme ce pot apărea în cadrul nivelului de administrare a resurselor de procesare, precum și în cadrul procesului de achiziție și prelucrare a imaginilor în general.

Testarea de tip cutie neagră presupune o evaluare a modulelor prin prisma rezultatelor pe care le oferă la anumite tipuri de intrări. Cu alte cuvinte, un modul este considerat acceptat dacă pentru un număr de intrări (de test), se obțin date de ieșire corecte. Testele verifică o serie de funcționalități de bază, iar, prin rularea lor după fiecare compilare a programului, se poate detecta dacă modificările aduse codului sursă au dus la pierderea de funcționalități sau nu. Trecerea tuturor testelor nu implică faptul că programul nu conține erori, ci doar faptul că într-un număr de situații standard el se comportă așa cum este așteptat.

Testele sunt grupate într-un proiect separat al soluției Visual Studio, denumit CytrusModuleTesting, fiind scrise utilizând platforma de testare a mediului Visual Studio (Se folosesc clasele din namespace-ul *Microsoft.VisualStudio.TestTools.UnitTesting*). Sunt implementate două tipuri de teste: de inițializare și de verificare a funcționalităților.

Cele de inițializare urmăresc instantierea corectă a obiectelor din cadrul nivelului de administrare a resurselor, verificând dacă starea obiectelor după inițializare este cea așteptată. Există două astfel de teste, unul pentru clasa CameraMgr și unul pentru ImageFileMgr. Pentru ca testele care se referă la surse live să fie trecute cu succes, pe calculatorul pe care se rulează testele trebuie să fie conectat un dispozitiv de captură (cameră web, cameră digitală).

Testarea funcționalităților de bază este realizată cu ajutorul a 5 teste:

- *FPSTestRun* realizează procesarea cu algoritmul SURF a unui flux de imagini provenit de la o sursă live de imagini. Se verifică parametrii de performanță ai acestei procesări, realizându-se o mediere a numărului de frame-uri procesate în fiecare secundă, pe o perioadă de 10 secunde. Dacă algoritmul nu reușește procesarea a măcar 10 frame-uri pe secundă, acest test va eşua.
- *StaticImage_ReadTest_Lenna* verifică dacă citirea unei imagini de test (lenna.jpg, prezentă în directorul proiectului de testare) se realizează în mod corect de către sursa statică implementată în aplicație prin *FileImageSource*.
- *SURFRun_StaticImage_Lenna* rulează algoritmul SURF pe o imagine de test, de dimensiuni mici. Se verifică comportamentul adecvat al algoritmului și detecția unui număr suficient de mare de puncte de interes (peste 50)
- *SURFRun_StaticImage_Large* rulează algoritmul SURF pe o imagine de dimensiune foarte mare, stocată pe disc. Se dorește testarea parcurgerilor de vectori din cadrul algoritmului. Dacă există zone în care algoritmul accesează zone de memorie nealocate (greșeli de index în vector), este foarte probabil ca acest test să eșueze.
- *StaticImage_SurfRun_MemoryAlloc_Dealloc_Stress50* realizează de 50 de ori alocarea memoriei pentru citirea unei imagini mari de pe disc, rularea algoritmului SURF și dealocarea memoriei. Se urmărește dacă în procesul de dealocare al memoriei sunt eliberate toate obiectele alocate, verificându-se astfel existența unor erori în dealocarea memoriei. (eng. leaks).

Se recomandă rularea acestor teste după compilarea soluției Visual Studio, pentru a verifica dacă toate modulele au fost construite și linkeditate corect. Se verifică astfel și îndeplinirea tuturor dependențelor externe necesare pentru rularea aplicației.

Capitolul 6

Rezultate și Concluzii

Experimentele realizate pentru a verifica corectitudinea implementării algoritmului SURF, precum și analiza comportamentului acestuia în condiții variate, au fost realizate într-un mediu necontrolat (nu au fost preluate date anterioare privind poziționarea obiectelor în scenă, luminozitatea ambientală sau poziția camerelor). Obiectele prezente în mediu și alese pentru a fi detectate conțin zone cu variații de culoare și textură și nu au aplicate marcaje speciale care să ajute în identificarea punctelor de interes. Detectarea unora dintre ele ar fi dificilă folosind puncte de interes determinate prin alte metode (cu detectorul Harris, de exemplu), deoarece nu au muchii sau colțuri bine definite.

Au fost folosite 3 dispozitive pentru capturarea imaginilor, necalibrate anterior:

- Cameră Web Creative Live!, conexiune USB, focus fix, rezoluție utilizată 320×240 , max. 15 cadre pe secundă (variabil în funcție de condițiile de luminozitate).
- Cameră video digitală Sony DCR-TRV730E, conexiune firewire (IEEE 1394), focus automat, rezoluție 720×520 , max. 25 cadre pe secundă
- Aparat foto digital Canon PowerShot A590 8Mp, focus automat, rezoluție 3264×2448 .

Rularea propriu-zisă a algoritmului, achiziția în timp real a datelor și detecția obiectelor au fost realizate pe un Laptop cu procesor Intel Core2 Duo 2.2GHz, cu 4GB RAM și placă grafică NVIDIA Quadro NVS 135M, cu 128 MB RAM. Toate rezultatele descrise mai jos sunt obținute experimental folosind această configurație. Pentru capturile în timp real de dimensiune 720×520 a fost obținută o viteza medie de 13 cadre pe secundă. Pentru dimensiuni mai mici ale imaginii, viteza de prelucrare este limitată de posibilitățile dispozitivelor de captură.

După executarea pașilor algoritmului, se estimează pentru fiecare cadru și precizia detecției obiectului, luându-se în calcul media distanțelor între vectorii descriptori ai fiecărui punct de interes detectat în cadrul curent și vectorii descriptori ai obiectului definit ca referință. Ecuația utilizată este:

$$Acc = \frac{\sum_{i=0}^p Dist_{r_i v_i}}{p} \quad (6.1)$$

, unde p reprezintă numărul total al punctelor identificate ca aparținând obiectului, iar $Dist_{r_i v_i}$ este distanța dintre vectorul referință r_i și vectorul detectat v_i , obținută pe baza relației 2.16. Valorile obținute nu trebuie interpretate ca fiind o probabilitate a prezenței sau absenței obiectului din scenă, deși o valoare 0 indică faptul că obiectul nu a fost detectat.

6.1 Rezultate experimentale

Mai jos sunt prezentate un număr de rezultate considerate relevante, prin prisma rezultatelor tipice obținute de către alți autori în analiza algoritmului SIFT sau SURF [Lowe, 1999, Brown and Lowe, 2002, C.Schmid et al., 2000].

Invarianța la rotații

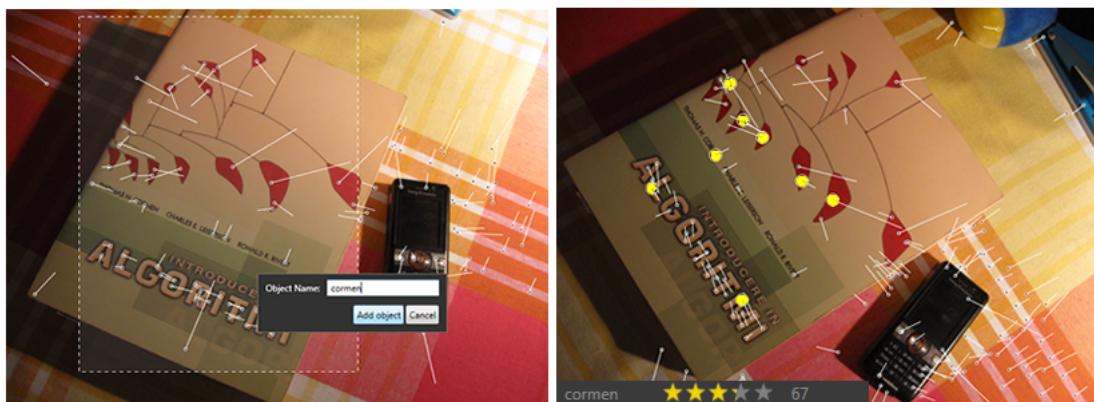


Figura 6.1: Detecția obiectului rotit relativ la imaginea de referință

În cazul de față (figura 6.1), ambele imagini sunt preluate cu aparatul foto. Imaginea din partea stângă reprezintă referință, în care au fost selectate punctele de interes corespunzătoare cărții. În partea dreaptă este prezentat rezultatul detecției obiectului, algoritmul fiind rulat pe o imagine statică. În comparație cu imaginea de referință, carteza este rotită la un unghi de aproximativ 30°. Se

poate observa detecția cu succes. Punctele reprezentând protriviri suficient de bune sunt împrăștiate pe toată suprafața obiectului, precizia detecției lor fiind în jurul valorii de 67% ($Acc = 0.67$).

Invarianța la scalări



Figura 6.2: Detectia obiectului in prezența unor modificări de scală

Ambele imagini sunt achiziționate folosind aparatul foto (figura 6.2). În partea stângă este imaginea de referință, reprezentând toată scena de test. În partea dreaptă, poate fi observat rezultatul detecției folosind o imagine de detaliu (cartea este prezentă în această imagine la o scală mai mare). Între cele două imagini există și o diferență de culoare, datorată setărilor automate ale aparatului foto. Punctele reprezentând protriviri suficient de bune sunt împrăștiate pe toată suprafața obiectului, precizia detecției lor fiind în jurul valorii de 65% ($Acc = 0.65$).

Modificări de luminositate ambientală

În figura 6.3, ambele imagini sunt preluate de la camera web, în condiții de luminositate diferite. Ele reprezintă rezultatul detecției unui obiect definit anterior. Se poate observa că deși punctele de interes detectate nu sunt aceleași în ambele situații, se reușește totuși identificarea corectă a obiectului. De asemenea, este de menționat calitatea slabă a imaginilor, regiuni relativ mari fiind subexpuse sau supraexpuse, acesta fiind unul dintre factorii care reduc calitatea detecției în cazul algoritmilor de tip SIFT.

Transformări multiple

Figura 6.4 prezintă rezultatul detecției în contextul unor transformări multiple în comparație cu imaginea de referință. Ambele imagini sunt achiziționate folosind



Figura 6.3: Detecția obiectului în prezență unor modificări de luminozitate

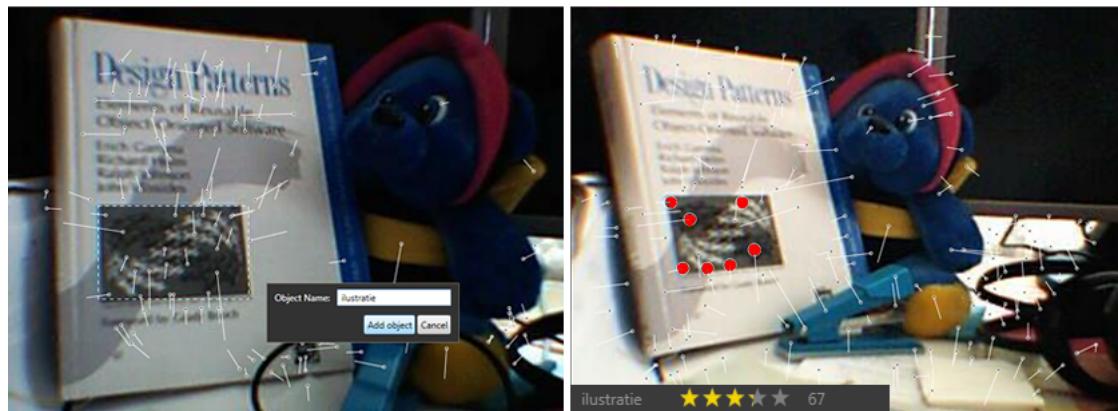


Figura 6.4: Detecția obiectului în prezență unor transformări multiple

camera web, între ele existând diferențe de scală, claritate, perspectivă și luminozitate. Recunoașterea este realizată totuși cu succes, pentru regiunea definită în imaginea referință. În plus, această regiune are o textură complicată.

Modificarea rezoluției imaginii

Figura 6.5 prezintă rezultatul detecției la modificarea dispozitivului de achiziție a imaginii. În cazul de față, imaginea de referință este preluată folosind camera video, iar imaginea pe care se realizează detecția este achiziționată de la camera web. Schimbarea rezoluției și a dispozitivului de captură duce în general la rezultate mai slabe de detecție. Cele două surse de imagini au lentile și senzori diferiți, și prin urmare imaginile rezultate vor avea caracteristici diferite. Recunoașterea este realizată cu succes, deși numărul punctelor detectate este mic și precizia nu

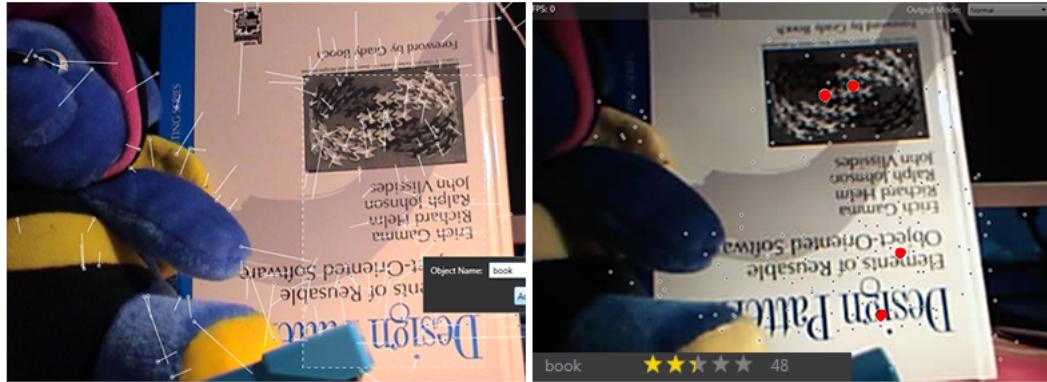


Figura 6.5: Detecția obiectului la schimbarea rezoluției

este foarte mare (aproximativ 48% ($Acc = 0.48$)).

Detectia de obiecte multiple

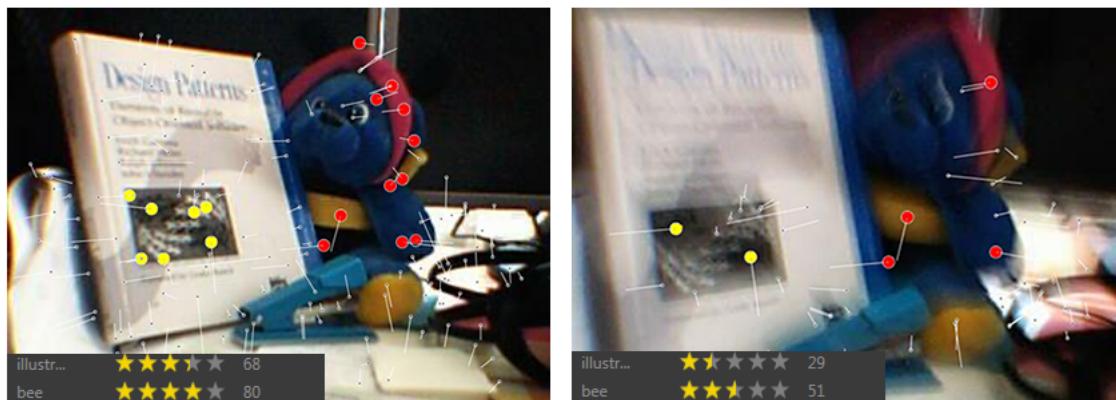


Figura 6.6: Detectia de obiecte multiple

Folosind același algoritm, se poate realiza și identificarea mai multor obiecte în aceeași scenă, chiar și în prezența unor ocluziuni parțiale a unora dintre obiecte. Un astfel de rezultat este prezentat și în figura 6.6, unde sunt detectate cu succes 2 obiecte. Imaginea din dreapta este de foarte slabă calitate (blur de mișcare), însă detectia este realizată cu succes. Folosind aceeași tehnică (a definirii mai multor obiecte), se poate îmbunătății și detectia unui singur obiect: se recomandă selectarea acestuia în mai multe imagini diferite (obiectul fiind pozat din unghiuri diferite, la scalări diferite sau în condiții diferite de luminozitate). Apoi, aplicând

algoritmul de recunoaștere, se va putea obține o invarianță crescută la condițiile de mediu, pentru detecția obiectului respectiv.

Erori în detecție

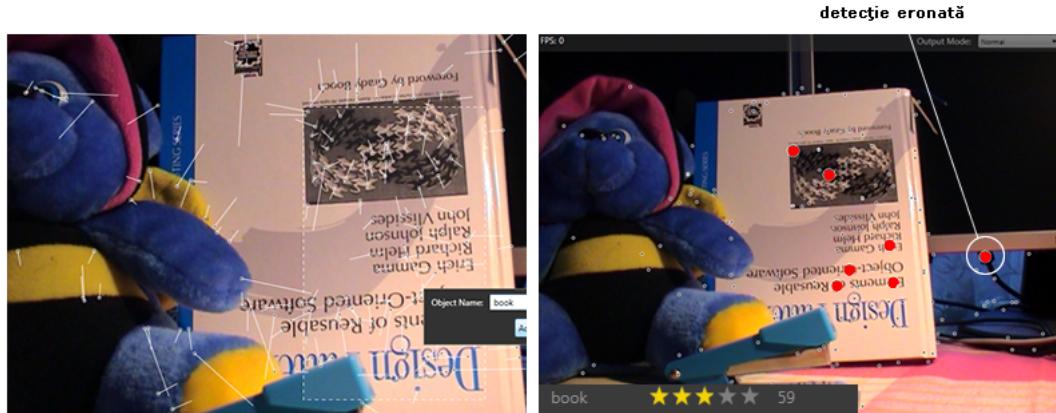


Figura 6.7: Detecția eronată a unui punct de interes

Uneori, datorită modelului simplu ales în aplicație pentru potrivirea punctelor de interes, sunt detectate și unele care nu aparțin obiectului căutat. Aceste detecții false (eng. false-positives) apar ca urmare a asemănării unor alte regiuni din imagine cu regiuni de pe obiectul căutat. Eliminarea situațiilor de acest gen poate fi realizată folosind o clusterizare a punctelor de interes și detecția unor grupări de puncte poziționate similar, în locul potrivirii punct cu punct. Precizia în acest caz poate crește semnificativ.

6.2 Concluzii

Soluția propusă pentru identificarea obiectelor cu ajutorul algoritmului SURF este una robustă, ce poate fi folosită cu succes într-un număr mare de aplicații. În plus, platforma dezvoltată pentru prelucrarea fluxurilor de imagini permite o adaptare și o extindere ușoară a funcționalităților.

Prin comparație, alte implementări open-source ale algoritmului SURF (Open-Surf, GPUSurf [Cornelis and Gool, 2008]) sunt dezvoltate utilizând biblioteci care pot fi adaptate mai greu unui număr mare de surse și formate ale imaginilor de intrare. În această implementare, s-a urmărit cu precădere obținerea unei prelucrări și detecții în timp real, pentru imagini de dimensiuni cât mai mari.

Pentru a îndeplini acest obiectiv, aplicația (cytrus) utilizează paralelismul la nivel de procesor, permățându-se astfel o îmbunătățire a performanțelor algoritmului față de soluții care utilizează un singur fir de execuție (cazul OpenSurf),

păstrând aceeași acuratețe a rezultatelor. Totuși, în comparație cu soluția identificării obiectelor propusă în [Brown and Lowe, 2002], lucrarea de față utilizează o metodă mai simplă și mai puțin precisă pentru realizarea potrivirii între seturile de puncte de interes. Eliminarea unora dintre erorile de detecție ar putea fi realizată aplicând tehnici similare celor din [Brown and Lowe, 2002], realizând compromisul unei reduceri a vitezei de procesare datorată calculelor suplimentare.

Prelucrarea imaginilor de dimensiuni mari în timp real este posibilă și datorită propunerii de a folosi o frecvență de eșantionare variabilă în spațiul scalarilor. Astfel, se realizează o limitare a creșterii numărului de puncte de interes o dată cu mărirea dimensiunilor imaginii. Practic, având în vedere că performanțele de viteza ale algoritmului depind în mod direct de numărul de puncte detectate, folosirea unei frecvențe care să scadă liniar cu creșterea dimensiunilor imaginii asigură implicit o limitare a creșterii timpului de procesare.

6.2.1 Direcții de cercetare

Reconstrucția unor obiecte ca modele 3D

Având în vedere faptul că algoritmul SURF realizează o identificare consistentă a punctelor de interes între imagini, invariant la schimbări ale condițiilor de mediu și ale perspectivei, există posibilitatea de a aplica acest algoritm în **reconstrucția unor obiecte** și crearea de modele 3D rare (bazate pe puncte) sau dense (suprafete). Dacă este necesar, modificarea parametrilor de rulare utilizării poate asigura o acoperire mai bună cu puncte de interes a obiectelor vizate (comparativ cu aplicația de identificare a obiectelor).

În momentul construirii unor astfel de modele, se pune în mod natural problema identificării obiectelor nerigide, cu luarea în calcul a deformărilor care pot apărea în timp. Se dorește analiza oportunității de a folosi algoritmi de tip SURF în studiul deformărilor care apar în cadrul unor studii medicale: un exemplu ar fi studiul comparativ al protezelor femurale (Figura 6.8), sau analiza post-operatorie a poziționării implanturilor folosind radiografii din unghiuri multiple.

Realitate virtuală

Soluții bazate pe algoritmi de tip SIFT pot fi folosite în aplicații de augmentare a realității cu obiecte virtuale. Astfel de soluții pot fi aplicate în industria cinematografică sau a jocurilor, iar biblioteci care să implementeze o parte a funcționalităților necesare sunt deja disponibile (de exemplu, ARToolkit - <http://www.hitl.washington.edu/artoolkit/>). Aici, SIFT/SURF pot fi utilizati pentru a determina parametrii geometrici ai scenei reprezentate (po-

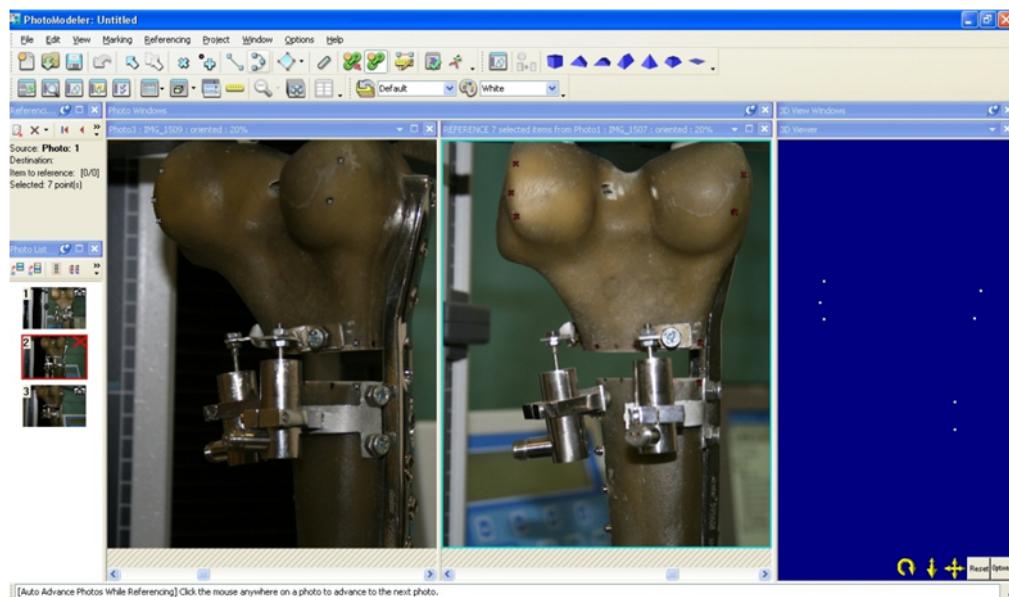


Figura 6.8: Analiza clasică a deformărilor în cazul studiului comparativ al protezelor femurale, cu PhotoModeller. Înregistrarea punctelor de interes se realizează manual.

ziție a camerelor, perspectivă), furnizând astfel informații despre cum trebuie randate obiectele virtuale pentru a se integra în imaginea de ansamblu.

Motoare de căutare

Îmbunătățiri semnificative ale vitezei de potrivire a punctelor de interes dintr-o imagine dată cu o bază de date de dimensiuni foarte mari, conținând puncte de inters ale multor obiecte sau imagini de referință, ar putea duce la aplicarea algoritmului SURF în următoarea generație de **motoare de căutare**. De exemplu, numele unui actor ar putea fi căutat furnizând ca dată de intrare o poză a acestuia. Pentru a folosi SURF în aceste aplicații, ar putea fi necesară o modificare a modului de stocare al descriptorilor, care să permită potrivirea rapidă.

Tehnologii similare au fost deja dezvoltate, la scară redusă, pentru dispozitivele mobile. Microsoft Tag este una dintre ele (<http://www.microsoft.com/tag/>). Totuși, aceste propuneri folosesc soluții similare codurilor de bare, fiind necesară o pre-marcare a obiectului pentru a putea realiza recunoașterea sa.

Bibliografie

- [Ballard and Brown, 1982] Ballard, D. and Brown, C. (1982). *Computer Vision*. Prentice-Hall, Englewood Cliffs. [cited at p. 5]
- [Bay et al., 2006] Bay, H., Tuytelaars, T., and Gool, L. V. (2006). Surf: Speeded-up robust features. In *ECCV*, pages 404–417. [cited at p. 6, 14, 17, 37]
- [Beardsley et al., 1996] Beardsley, P., Beardsley, P., Torr, P., Torr, P., Zisserman, A., and Zisserman, A. (1996). 3d model acquisition from extended image sequences. pages 683–695. Springer-Verlag. [cited at p. 19]
- [Brown and Lowe, 2002] Brown, M. and Lowe, D. (2002). Invariant features from interest point groups. In *In British Machine Vision Conference*, pages 656–665. [cited at p. 10, 19, 54, 59]
- [Cheng and Huang, 1984] Cheng, J. and Huang, T. (1984). Image registration by matching relational structures. *Pattern Recognition*, 17(1):149–159. [cited at p. 6]
- [Cole et al., 2004] Cole, L., Austin, D., and Cole, L. (2004). Visual object recognition using template matching. In *Proceedings of Australian Conference on Robotics and Automation*. [cited at p. 5]
- [Cornelis and Gool, 2008] Cornelis, N. and Gool, L. V. (2008). Fast scale invariant feature detection and matching on programmable graphics hardware. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8. [cited at p. 58]
- [C.Schmid et al., 2000] C.Schmid, Mohr, R., and C.Bauckhage (2000). Evaluation of interest point detectors. *International Journal of Computer Vision*, 37(2):151–172. [cited at p. 8, 54]
- [Evans, 2009] Evans, C. (2009). Notes on the opensurf library. Technical Report CSTR-09-001, University of Bristol. [cited at p. 15, 16, 18, 100]
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. [cited at p. 23]

- [Goshtasby et al., 1984] Goshtasby, A., Gage, S., and Bartholic, J. (1984). A two-stage cross correlation approach to template matching. *IEEE Transactions, Pattern Analysis & Machine Intelligence*, 6(3):374–378. [cited at p. 5]
- [Harris and Stephens, 1988] Harris, C. and Stephens, M. (1988). A combined corner and edge detection. In *4th Alvey Vision Conference*, pages 147–151. [cited at p. 7]
- [Lowe, 1999] Lowe, D. (1999). Object recognition from local scale-invariant features. pages 1150–1157. [cited at p. 9, 54]
- [Lowe, 2003] Lowe, D. G. (2003). Distinctive image features from scale-invariant keypoints. [cited at p. 6, 9, 11, 14]
- [Ullman, 1979] Ullman, S. (1979). *The Interpretation of Visual Motion*. MIT Press, Cambridge, MA. [cited at p. 6]
- [Zhang et al., 1995] Zhang, Z., Deriche, R., Faugeras, O. D., and Luong, Q. T. (1995). A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry. *Artificial Intelligence*, 78(1-2):87–119. [cited at p. 19]

Anexe

Anexa A

Diagrame UML

În această anexă apar diagramele UML detaliate ale proiectului cytrus. Organizarea este în funcție de namespace-urile claselor, după cum urmează:

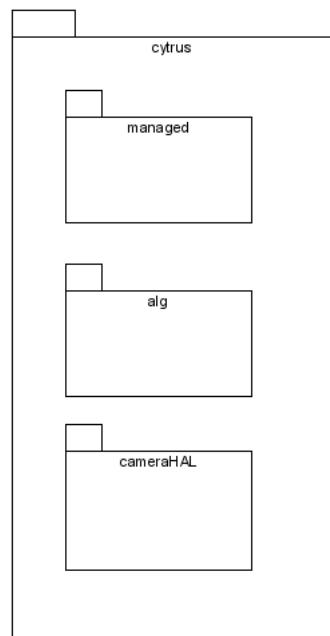
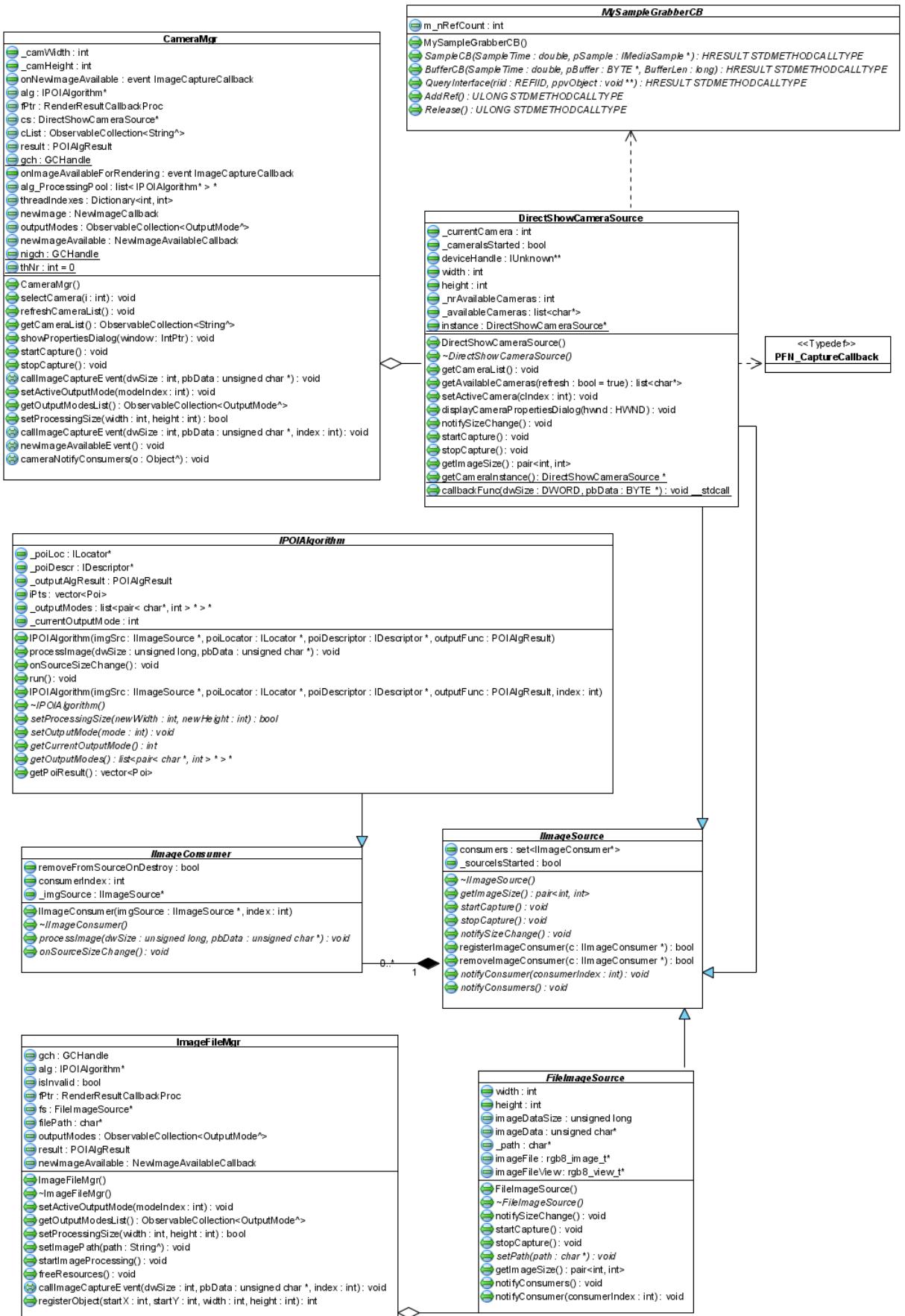
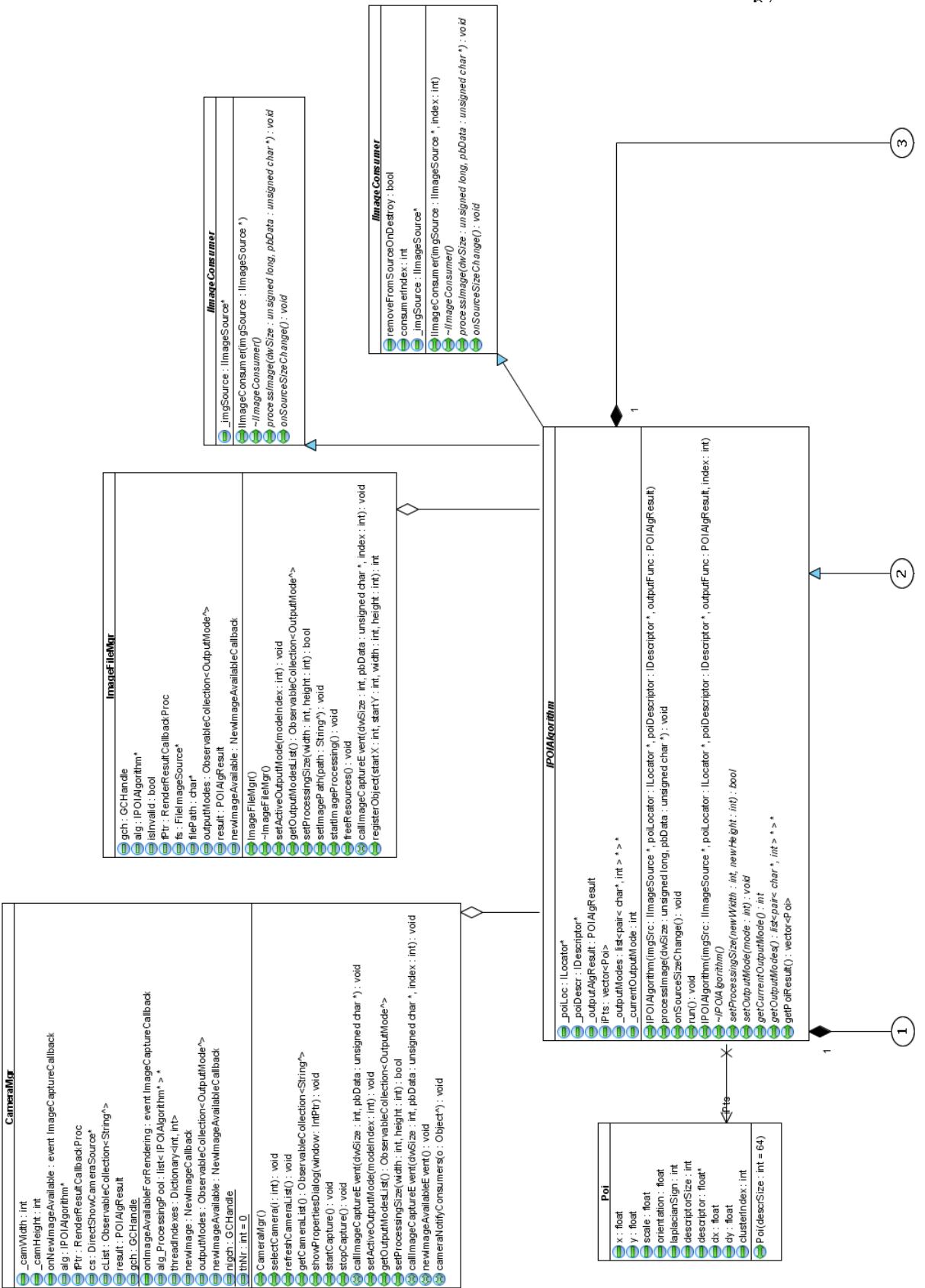


Figura A.1: cytrus

Figura A.2: `cytrus::cameraHAL`



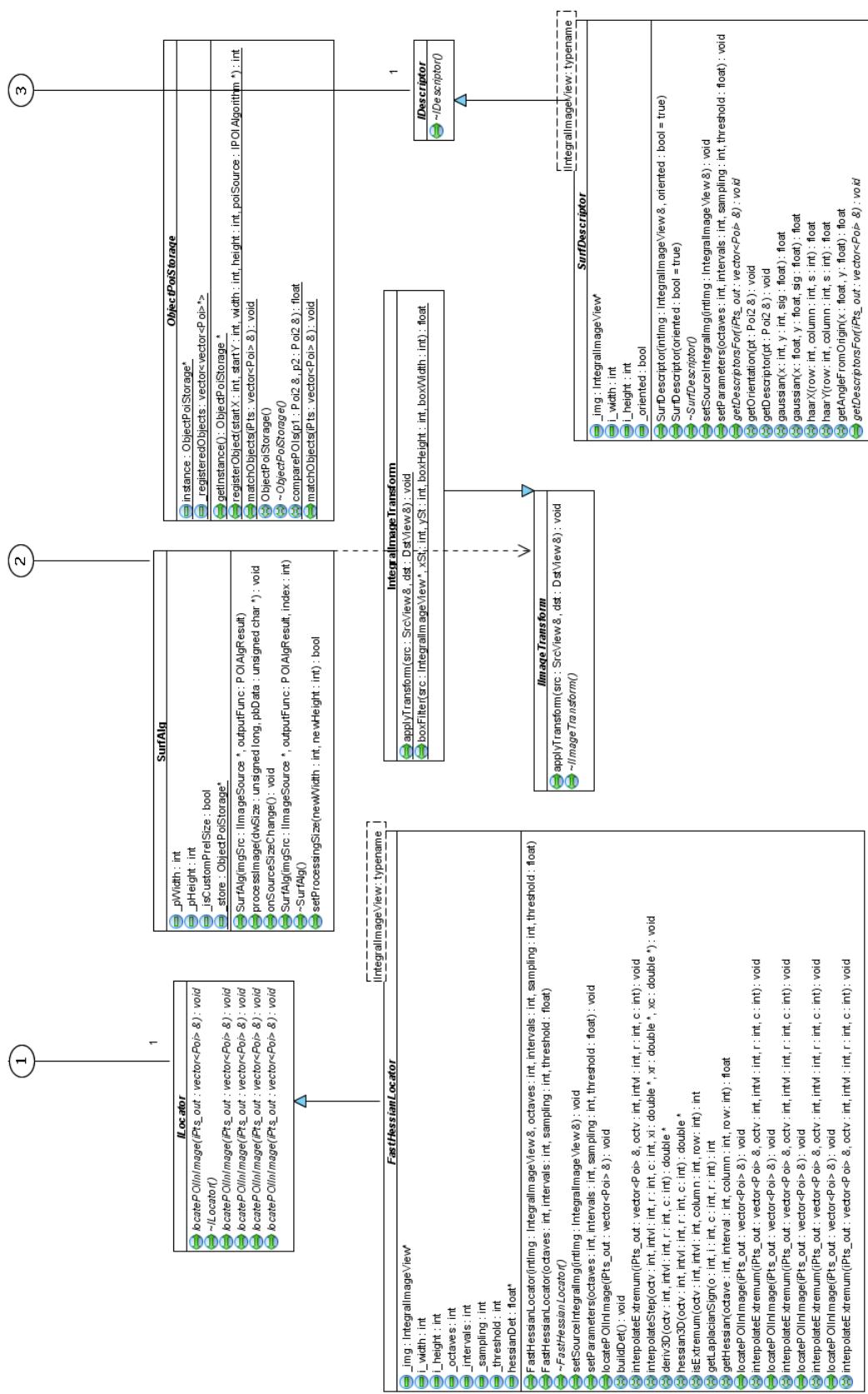
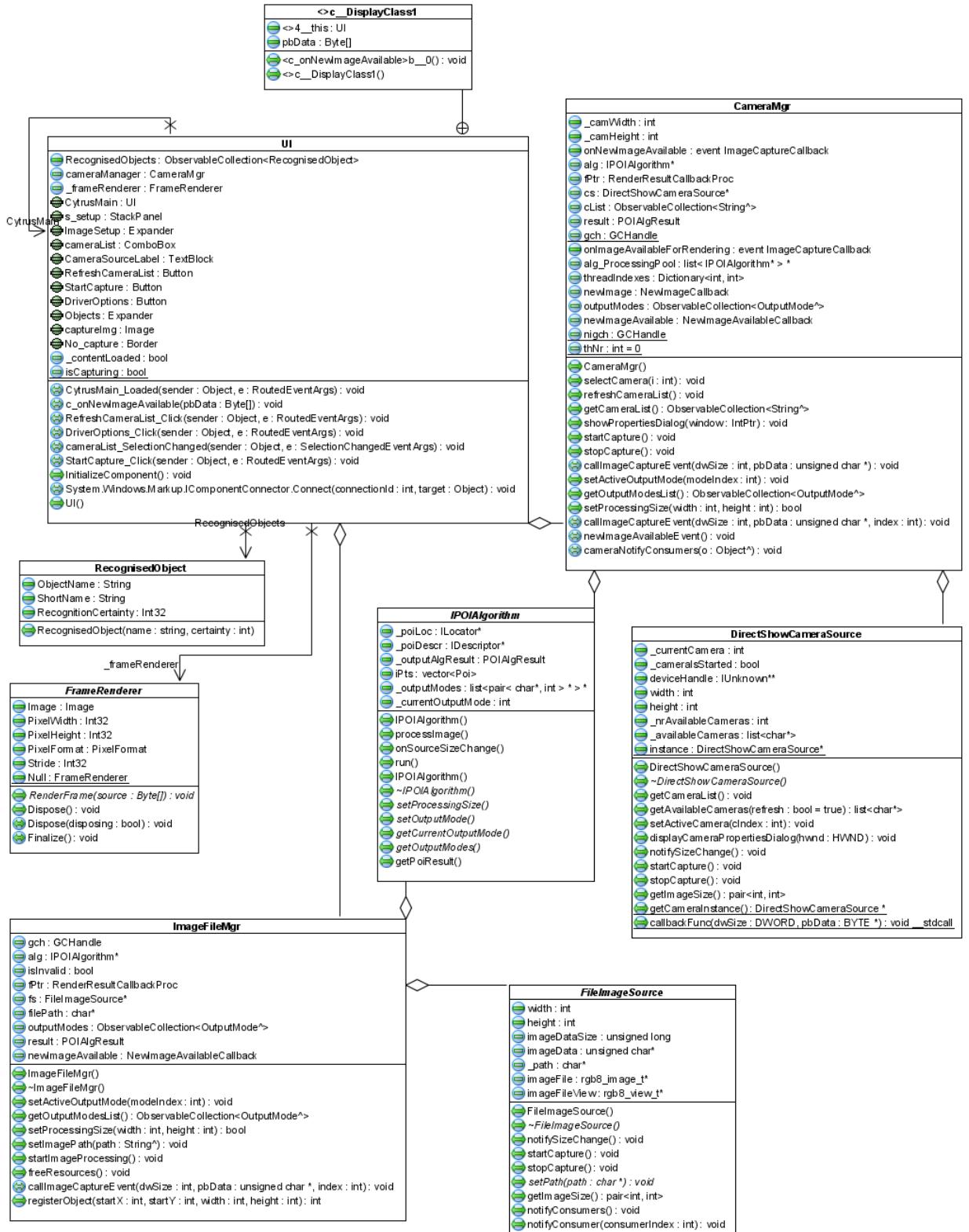


Figura A.3: cytrus::alg

Figura A.4: `cytrus::managed`

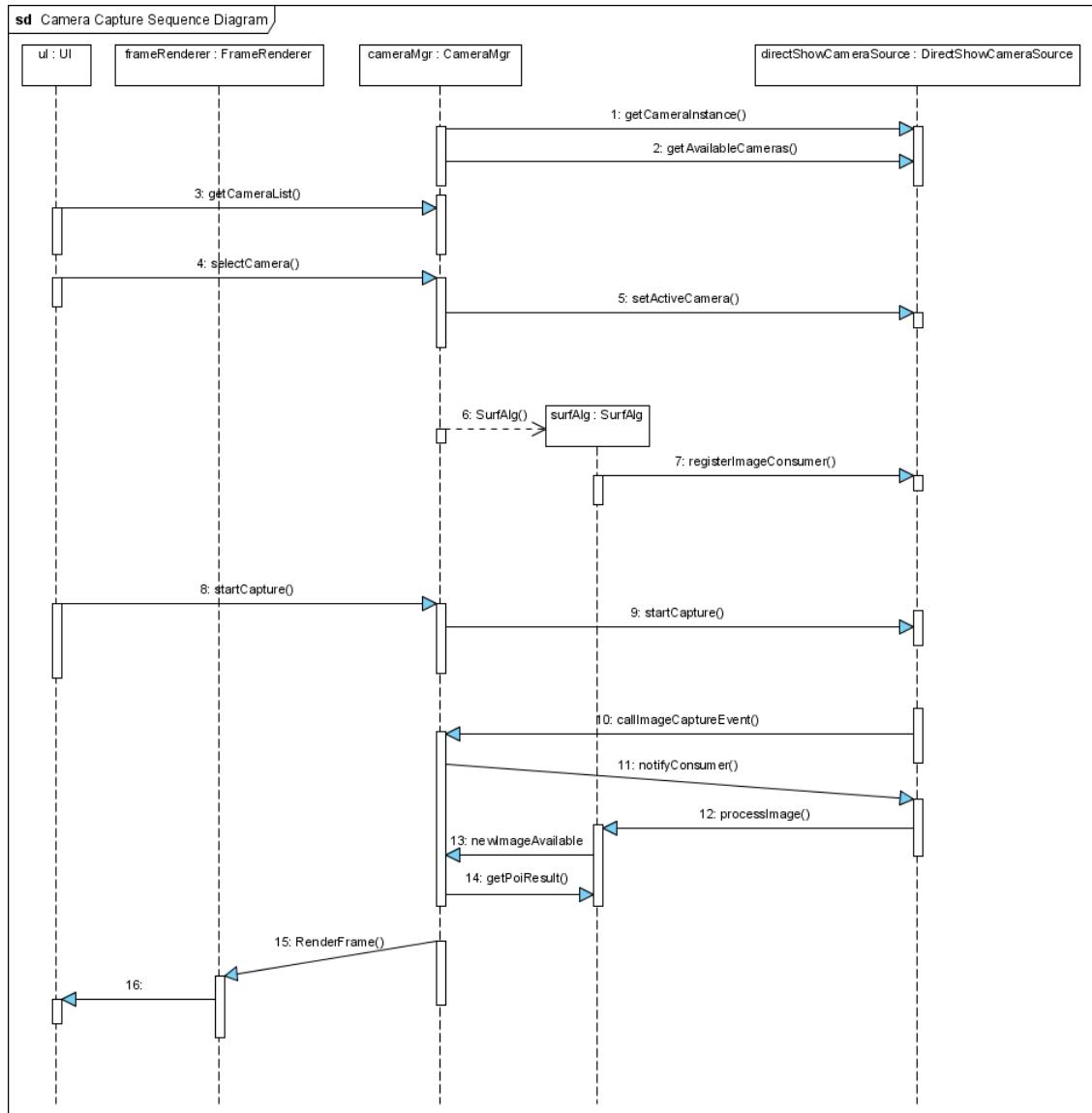


Figura A.5: Diagramă de secvență pentru procedura de achiziție a imaginilor (simplificat)

Anexa B

Filtre utilizate în algoritmul SURF

În această anexă sunt prezentate pe scurt varianțele continuii ale filtrelor folosite în cadrul algoritmului SURF, și implementate în varianta lor discretă, folosind aproximări (Figura 2.6). Totuși, o justificare a folosirii acestor filtre este importantă, pentru că alegerea utilizării lor nu a fost una întâmplătoare. Este de menționat faptul că deși în cadrul acestei anexe majoritatea filtrelor sunt reprezentate și în 3 dimensiuni, pentru prelucrarea imaginilor se folosesc varianțele 2D. Imaginile 3D au fost atașate pentru a da o viziune mai clară asupra formei generale a filtrelor.

Nucleul Gauss

Ca funcție matematică de două variabile și parametru σ , nucleul gaussian poate fi exprimat sub forma:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Forma grafică a unei astfel de funcții poate fi observată în Figura B.1. În general, nucleul Gauss este folosit în estomparea imaginilor (blur), iar în cadrul algoritmilor de tip SIFT el este utilizat împreună cu Laplacianul, variind σ , pentru a crea spațiul scalărilor. Faptul că nucleul Gauss este simetric permite obținerea invarianței la rotații. Nucleele Gauss mai sunt utilizate și atunci când se dorește ponderarea vectorilor gradienților din imagine, pentru a acorda o mai mare importanță celor mai aproape de centrul unei vecinătăți alese (a se vedea secțiunea 2.3.1).

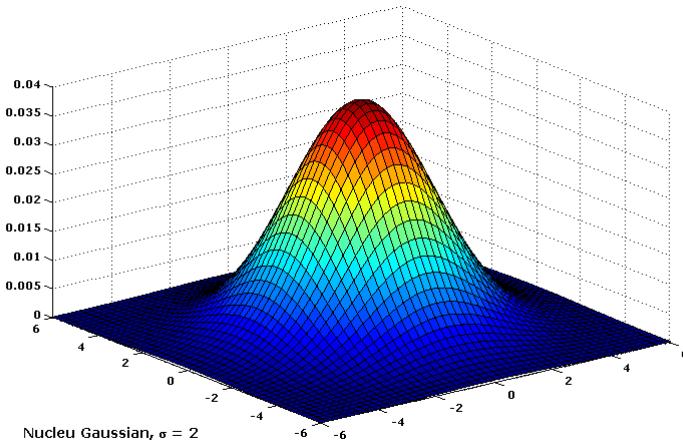


Figura B.1: Nucleul Gauss

Laplacianul Gaussian-ului, aproximare cu diferență de nuclee Gauss

Determinarea maximelor după convoluția imaginii cu o funcție ce reprezintă diferența a două nuclee Gauss, descrisă de ecuația 2.4 pornește de la premisa că această funcție aproximează destul de bine Laplacianul Gausianului, $\sigma^2 \nabla^2 G$. Aici, Laplacian-ul este folosit pornind de la ideea că pentru o singură dimensiune, derivata a două permite detecția unor structuri de tip muchie în imagine, dacă vom considera trecerile acesteia prin 0, precum în Figura B.2. Aplicarea suplimentară a nucleului Gauss în convoluție este necesară deoarece derivata a două este foarte sensibilă la zgomotul existent în semnalul inițial (în imagine pentru cazul 2D). Astfel, se dorește o netezire preliminară, pentru a mai estompa din zgomote. Datorită proprietăților filtrelor și ale convoluției, se poate aplica mai convoluția între nuclee (Laplacian,Gauss), urmând ca nucleul rezultat să fie aplicat (tot prin convoluție, imaginii). În Figura B.3 este prezentată asemănarea dintre nucleul obținut prin diferența de nuclee Gauss și nucleul funcției Laplacianul Gausianului.

Derivatele parțiale de ordinul doi ale nucleului Gauss

Pentru calcularea determinantului Hessian-ului utilizat în SURF, este necesară determinarea anterioară a derivelor parțiale ale funcției $L(x,y)$, reprezentând imaginea în convoluție cu nucleul Gaussian (2.7). Forma 2D a nucleelor este aproximată în cadul algoritmului SURF prin compunerea unor filtre de mediere (a se vedea Figura 2.6 pentru o comparație).

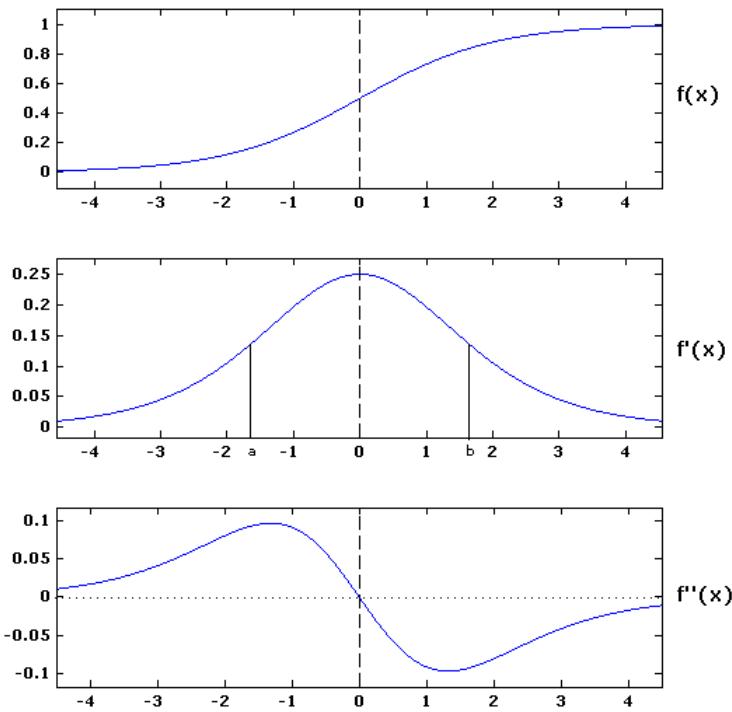


Figura B.2: Detectia unor muchii folosind trecerile prin 0 ale derivatei a doua

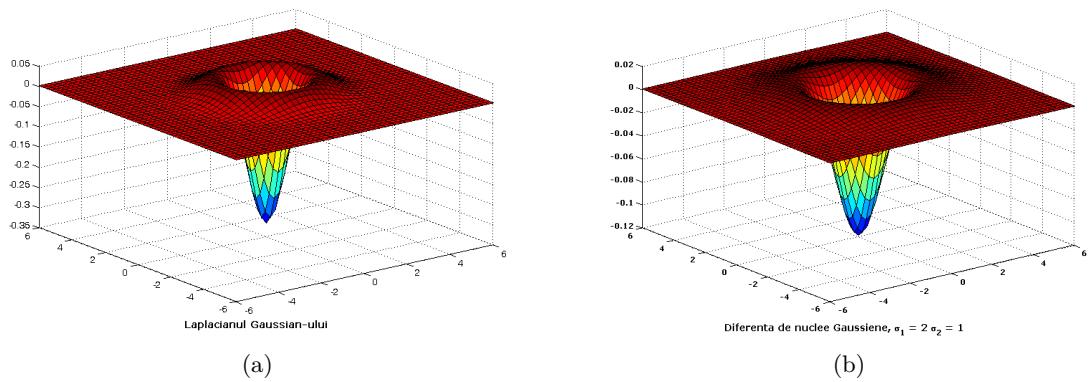
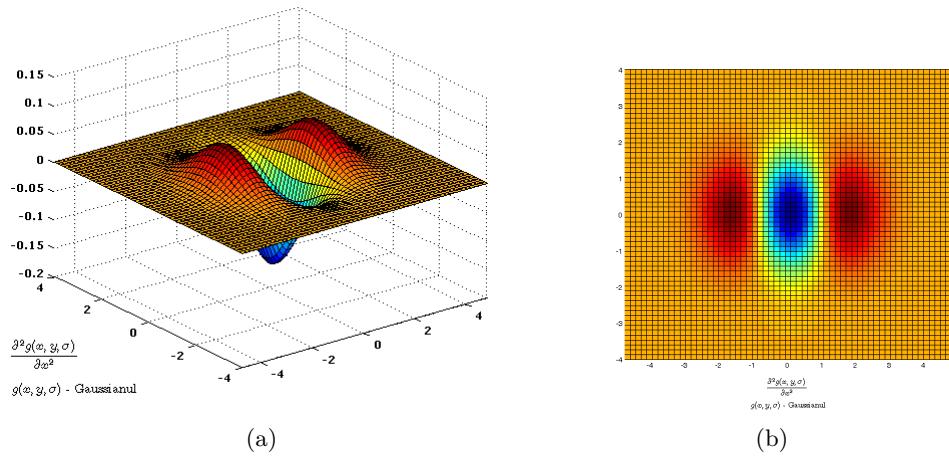
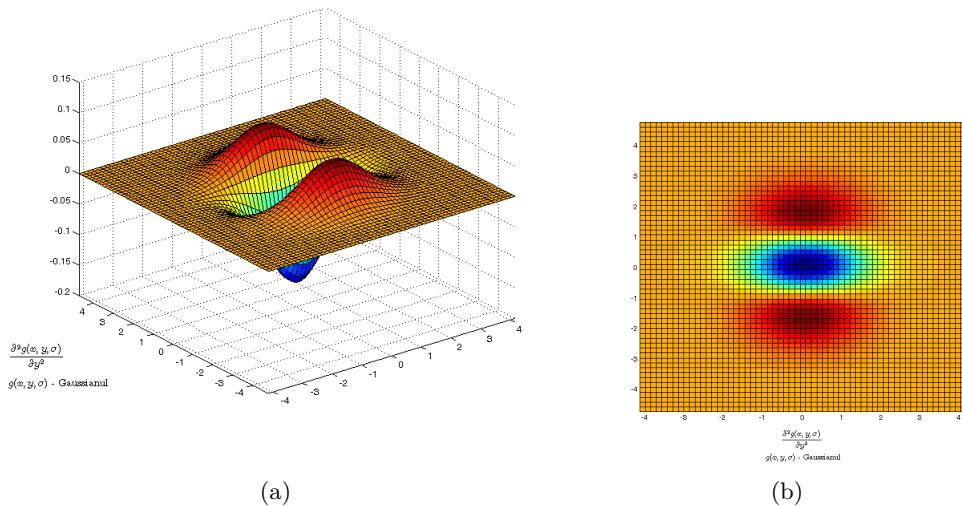
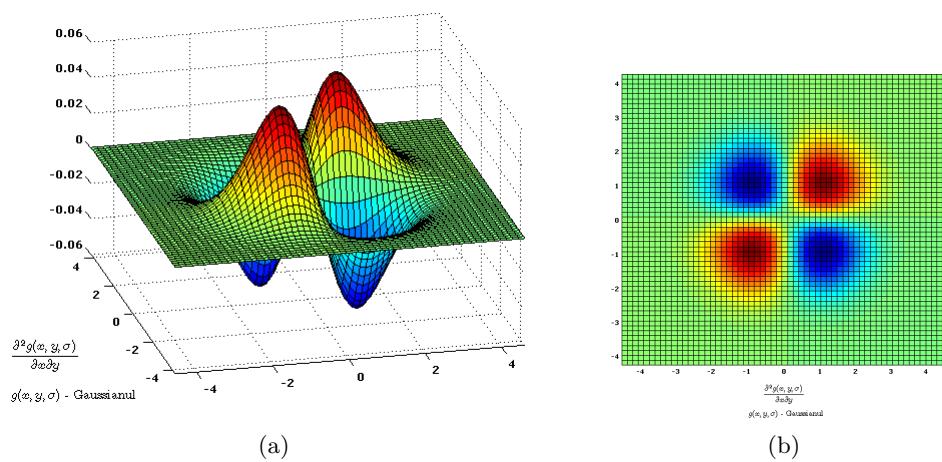


Figura B.3: Comparatia LoG/DoG (eng. Laplacian of Gaussian/Difference of Gaussian)

Figura B.4: Nucleu Gaussian, derivata L_{xx} Figura B.5: Nucleu Gaussian, derivata L_{yy} Figura B.6: Nucleu Gaussian, derivata L_{xy}

Anexa C

Codul aplicației (partial)

Carata Lucian, calucian[at]yahoo.co.uk

Copyright (C) 2009, Carata Lucian

Acest program este liber; îl puteți redistribui și/sau modifica în conformitate cu termenii Licenței Publice Generale GNU, aşa cum este publicată de către Free Software Foundation; fie versiunea 3 a Licenței, fie (la latitudinea dumneavoastră) orice versiune ulterioară.

Acest program este distribuit cu speranța că va fi util, dar FĂRĂ NICI O GARANȚIE; fără macar garantia implicită de vandabilitate sau CONFORMITATE UNUI ANUMIT SCOP. A se vedea Licența Publică Generală GNU pentru detalii.

Ar trebui să fi primit o copie a Licenței Publice Generale GNU împreună cu acest program.

În caz contrar, consultați <http://www.gnu.org/licenses/>.

```
IImageTransform.h:           General      interface     for      Image
transformations

1 #ifndef _IIMAGETRANSFORM_H_
#define _IIMAGETRANSFORM_H_

6 #ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif
11
```

```

16   namespace cytrus{
17     namespace alg{
18       class CYTRUSALGLIB_API IImageTransform{
19         public:
20           template <typename SrcView , typename DstView>
21             void applyTransform(SrcView& src , DstView& dst);
22
23           virtual ~IImageTransform(){}
24     };
25   }
26
27 #endif

```

IDescriptor.h : An interface that defines the contract for describing the points of interest in an image.

```

1 #ifndef _IDESCRIPTOR_H_
2 #define _IDESCRIPTOR_H_
3
4 #ifndef CYTRUSALGLIB_API
5   #ifdef CYTRUSALGLIB_EXPORTS
6     #define CYTRUSALGLIB_API __declspec(dllexport)
7   #else
8     #define CYTRUSALGLIB_API __declspec(dllimport)
9   #endif
10 #endif
11
12 namespace cytrus{
13   namespace alg{
14     class CYTRUSALGLIB_API IDescriptor{
15       public:
16         virtual ~IDescriptor(){}
17     };
18   }
19
20 #endif

```

ILocator.h : An interface that defines the contract for locating the points of interest in an image.

```

#ifndef _ILOCATOR_H_
#define _ILOCATOR_H_

```

```

4 #ifndef CYTRUSALGLIB_API
5     #ifdef CYTRUSALGLIB_EXPORTS
6         #define CYTRUSALGLIB_API __declspec(dllexport)
7     #else
8         #define CYTRUSALGLIB_API __declspec(dllimport)
9     #endif
#ENDIF

11 #include <vector>
12 #include "poi.h"
13
14 namespace cytrus{
15     namespace alg{
16         class CYTRUSALGLIB_API ILocator{
17             public:
18                 virtual void locatePOIInImage(std::vector<Poi>& iPts_out)
19                         =0;
20                 virtual ~ILocator(){}
21             };
22         }
23     }
#ENDIF

```

IntegralImageTransform.h: Transforms a given image into its integral representation

```

#ifndef _INTEGRALIMAGETRANSFORM_H_
#define _INTEGRALIMAGETRANSFORM_H_

5 #include "IImageTransform.h"
6 #include <boost/gil/gil_all.hpp>

7 using namespace boost::gil;

10 namespace cytrus{
11     namespace alg{
12         class IntegralImageTransform : public IImageTransform{
13             public:
14
15                 // SrcView and DstView must both be GIL Image Views. The
16                 // following conditions must be met:
17                 // ImageViewConcept<SrcView> >();
18                 // MutableImageViewConcept<DstView> >();
19                 // ColorSpacesCompatibleConcept<color_space_type<SrcView>::
20                 // type, color_space_type<DstView>::type> >();
21             template <typename SrcView, typename DstView>

```

```

20    static void applyTransform(SrcView& src , DstView& dst);

    // IntegralImageView must be a GIL Image View. The following
    // conditions must be met:
    // ImageViewConcept<SrcView> >();
    // The current implementation assumes the view has only one
    // channel
25    template <typename IntegralImageView>
        static float boxFilter(IntegralImageView* src , int xSt , int
            ySt , int boxHeight , int boxWidth);
    };

30 }

#include "IntegralImageTransform.hpp"

#endif

```

```

IImageSource.h : Image Source Interface
1 #ifndef __IMAGESOURCE_H__
#define __IMAGESOURCE_H__

# ifndef CYTRUSALGLIB_API
# define CYTRUSALGLIB_API
6 #endif

# include <set>
# include "IImageConsumer.h"

11 namespace cytrus{
    namespace cameraHAL{

        // Abstract base class for all image sources.
        // Defines the functions and members for registering and
        // removing image consumers (IImageConsumer)
16        // Register your image consumer by calling
        // registerImageConsumer, and the processImage function
        // will be called by the image source when an image is
        // available.
        // Current available implementations:
        // cytrus::cameraHal::DirectShowCameraSource - win32/directx
        // only
        // cytrus::cameraHal::ImageFromFileSource
21        // Info for subclassing:
        // If your source provides external consumer notification (like
        // function callbacks provided by a

```

```

// SO-specific image acquisition system), you should implement
// those in separate (static?) functions.
// In derived classes, implement the four pure virtual
// functions,
// notifyConsumers() - for manual consumer notification
// getImageSize() - for providing source-dependent image size
// information
26 class CYTRUSALGLIB_API IImageSource{
protected:
    std::set<IImageConsumer*> consumers;
    bool _sourceIsStarted;
31 public:
    virtual ~IImageSource() {
        // ...
    }
36    // This function manually notifies all the registered
    // consumers.
    // The implementation usually calls IImageConsumer.
    // processImage, passing the source's image
    // as a parameter.
    // ...
41    virtual void notifyConsumers()=0;
    // ...
    // This function manually notifies a specific registered
    // consumer.
    // The implementation usually calls IImageConsumer.
    // processImage, passing the source's image
    // as a parameter.
    // ...
46    virtual void notifyConsumer(int consumerIndex)=0;
    // ...
    // Returns the size <width,height> of the images that this
    // source currently provides.
    // ...
51    virtual std::pair<int,int> getImageSize()=0;
    // ...
    // Starts the capture on the specific image source. This
    // function will be called
    // after configuring the image source.
    // Implementation examples:
    // For a static single-image source, calling the
    // startCapture function should read the
    // image from disk and then notify all the consumers.
    // For a stream-type image source, calling the
    // startCapture should determine periodic
61

```

```

// notifications to the consumers, as the frames come from
// the capture device.
//
//
virtual void startCapture()=0;
66
//
// Stops the capture process on the specific image source.
//
virtual void stopCapture()=0;
71
//
// Default implementation does nothing. Subclasses might
// overwrite this to account for the cases when
// the size of the image provided by the source changes.
// Maybe it's a camera source and the user
// connected a second camera that provides another
// resolution (for example)
76
virtual void notifySizeChange() {};
//
// Registers an ImageConsumer with this source.
// The registered ImageConsumers will be notified when new
// images are produced by the source,
// and the image will be sent to them for processing.
//
// returns true if the consumer could be registered
// correctly
//           false if the consumer already exists in the list
81
;
//           another possibility is that the source is
// already started
//           for maintaining consistency, adding
// consumers "on the fly" is not supported
//
bool registerImageConsumer(IImageConsumer *c);

91
//
// Remove Image Consumer from the consumers that are
// waiting for this image source.
// returns true if the consumer could be removed correctly
//           false if the consumer did not previously exist
//           in the list;
//           another possibility is that the source is
// already started
//           for maintaining consistency, removing
// consumers "on the fly" is not supported
//
96
;

```

```

        bool removeImageConsumer(IImageConsumer *c);
    };

101 }
}

#endif

```

```

ImageConsumer.h : Image Consumer Interface
1 #ifndef _IMAGECONSUMER_H_
#define _IMAGECONSUMER_H_

# ifndef CYTRUSALGLIB_API
# define CYTRUSALGLIB_API
6 #endif

namespace cytrus{
    namespace cameraHAL{

11 class IImageSource;

    /**
     * Abstract base class for ImageConsumers
     * All the classes that make use of image sources should
     * implement this, so that they can be notified
16     * with images when those are available at the source.
     */
    class CYTRUSALGLIB_API IImageConsumer{
        protected:
            IImageSource* _imgSource;
21        public:
            bool removeFromSourceOnDestroy;
            int consumerIndex; // for identifying multiple consumers
            IImageConsumer(IImageSource* imgSource, int index);
            virtual ~IImageConsumer();
26        /**
         * This function is called by the image source when the
         * image data is ready
         * (has been loaded from disk, has been received from a
         * camera etc.).
         */
31        virtual void processImage(unsigned long dwSize, unsigned
            char* pbData)=0;
        /**
         * This function is called by the image source when the
         * size of the image

```

```

    // changes.
    //
36     virtual void onSourceSizeChange()=0;
    };

    }
}

41 #endif

```

```

FileImageSource.h : File Image Source. Implements
IImageSource

#ifndef _FILEIMAGESOURCE_H_
#define _FILEIMAGESOURCE_H_

3 #ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif

13 #include "IImageSource.h"
#include <boost/gil/gil_all.hpp>

namespace cytrus{
    namespace cameraHAL{

18     //
// FileImageSource is a source that reads images from files.
// <Limitation> Currently, the implementation only accepts .jpg
// files
//
23     class CYTRUSALGLIB_API FileImageSource: public IImageSource{
        private:
            unsigned long imageDataSize;
            unsigned char* imageData;

28             const char* _path;
            boost::gil::rgb8_image_t* imageFile;
            boost::gil::rgb8_view_t* imageFileView;

33             public:
                int width, height;

```

```

    FileImageSource();
    virtual ~FileImageSource();

38     virtual void notifySizeChange() {};
     virtual void notifyConsumers();
     virtual void notifyConsumer(int consumerIndex);

43     virtual void startCapture();
     virtual void stopCapture();
     virtual void setPath(const char* path);
     virtual std::pair<int,int> getImageSize();

48 }
}

53 #endif

```

```

DirectShowCameraSource.h : Camera Image Source. Implements
IImageSource

#ifndef _CAMERASOURCE_H_
#define _CAMERASOURCE_H_

5 #ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif

10 #endif

#include <list>
#include "WebCamLib.h"
#include "IImageSource.h"

15 #define NO_CAMERA -256

namespace cytrus{
    namespace cameraHAL{

20     #ifdef WIN32
        //
```

```

// DirectShowCameraSource is an image source aquiring images
// from camera devices
// (connected by usb, firewire etc) through DirectShow/DirectX
// As such, this is a Windows-specific implementation.
// Not threadsafe (yet)
//
25   typedef void ( __stdcall *NewImageAvailableCallback )(void) ;

30   class CYTRUSALGLIB_API DirectShowCameraSource: public
      IImageSource{
      // TODO: add thread safetyness to singleton & other members.
      private:
          static DirectShowCameraSource* instance; // singleton
          instance
          static NewImageAvailableCallback signalNewImageAvailable;
          IUnknown** deviceHandle;

          static DWORD imageDataSize;
          static BYTE* imageData;

40          // current capture device spec:
          int _currentCamera;

          //

45          DirectShowCameraSource();
          virtual ~DirectShowCameraSource();
          void getCameraList();

      public:

50          int width, height;
          int _nrAvailableCameras;
          std::list<char*> _availableCameras;

55          static DirectShowCameraSource* getCameraInstance(
              NewImageAvailableCallback callback);
          std::list<char*> getAvailableCameras(bool refresh=true);
          void setActiveCamera(int cIndex);
          void displayCameraPropertiesDialog(HWND hwnd);

60          virtual void notifySizeChange() {};
          virtual void notifyConsumers();
          virtual void notifyConsumer(int consumerIndex);

65          virtual void startCapture();
          virtual void stopCapture();
          virtual std::pair<int,int> getImageSize();

```

```

    static void __stdcall callbackFunc(DWORD dwSize, BYTE*
        pbData);
70    };
    #endif

}
75
#endif

```

IPOIAlgorthm.h : An interface that inherits IImageConsumer, and enforces a contract for all the algorithms that work on Points of Interest (POI), for object detection (Locator + Descriptor)

```

#ifndef _IPOIALGORITHM_H_
#define _IPOIALGORITHM_H_
3
#ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif
8
#include "IImageConsumer.h"
13 #include "ILocator.h"
#include "IDescriptor.h"
#include <list>

using namespace cyrus::cameraHAL;
18 using namespace cyrus::alg;

namespace cyrus{
    namespace alg{
        typedef void (__stdcall *POIAlgResult)(unsigned long dwSize,
            unsigned char* pbData, int index);
23     class CYTRUSALGLIB_API IPOIAlgorthm : public IImageConsumer{
        protected:
            ILocator *poiLoc;
            IDescriptor *poiDescr;
            POIAlgResult _outputAlgResult;
            std::vector<Poi> iPts;
28
            std::list<std::pair<char*,int>>* _outputModes;

```

```

33     volatile int _currentOutputMode;
public:
34     IPOIAlgAlgorithm(IIImageSource* imgSrc, ILocator* poiLocator,
35                     IDescriptor* poiDescriptor, POIAlgResult outputFunc,
36                     int index);
37     virtual ~IPOIAlgAlgorithm() ;

38     //This function should be overriden to process the image
39     //data from the source.
40     //It will be automatically called by the image source, when
41     //the algorithm runs.
42     //If one has to display the results of the algorithm to an
43     //outside source,
44     //the implementation of this function can call
45     //    _outputAlgResult with appropiate data.
46     virtual void processImage(unsigned long dwSize, unsigned
47                               char* pbData)=0;

48     //Allows for processing the image at a smaller size than
49     //captured
50     //returns true if the processing size could be set (is
51     //valid, processing size<image size)
52     virtual bool setProcessingSize(int newWidth, int newHeight)
53     =0;

54     //Actions to take when the image from the source changes
55     //size
56     virtual void onSourceSizeChange()=0;
57

58     //Manage algorithm Output modes
59     virtual void setOutputMode(int mode);
60     virtual int getCurrentOutputMode();
61     virtual std::list<std::pair<char*,int>>* getOutputModes();

62     //Runs the POI Algorithm
63     void run();

64     std::vector<Poi> getPoiResult();
65 };
66 }
67

#endif

```

FastHessianLocator.h : Locating POI's using the determinant
of the Hessian matrix. Some parts are adapted from Bay's
SURF implementation/notes:C. Evans, Research Into Robust Visual

```

Features, MSc University of Bristol, 2008.

#ifndef _FASTHESSIANLOCATOR_H_
#define _FASTHESSIANLOCATOR_H_

#ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif

#include <vector>
#include <iostream>
#include "poi.h"
#include "ILocator.h"

namespace cytrus{
    namespace alg{

        static const int OCTAVES = 3;
        static const int INTERVALS = 4;
        static const float THRES = 0.002f;
        static const int INT_SAMPLE = 2;

        template <typename IntegralImageView>
        class FastHessianLocator:public ILocator{

private:

    void buildDet(); // this fills the hessianDet with
                    // values calculated
                    // for pixels in the image (sampled)

        //! Interpolation functions - adapted from Lowe's
        // SIFT implementation
        // (Adapted from Bay, Evans - SURF, the above
        // comment is from their
        // implementation)
        void interpolateExtremum(std::vector<Poi>& iPts_out
                                , int octv, int intvl, int r, int c);
        void interpolateStep(int octv, int intvl, int r,
                            int c, double* xi, double* xr, double* xc );
        double* deriv3D( int octv, int intvl, int r, int c
                        );
        double* hessian3D(int octv, int intvl, int r, int c
                        );
        // Non-maximum suppression
        int isExtremum(int octv, int intvl, int column, int
    };
}
}

```

```

        row) ;

42
    //utility
inline int getLaplacianSign(int o, int i, int c,
                           int r);
inline float getHessian(int octave, int interval,
                        int column, int row);

protected:

47
IntegralImageView* _img;
int i_width, i_height;

52
int _octaves;
int _intervals;
int _sampling;
int _threshold;

57
float* hessianDet; // array that contains

public:
    FastHessianLocator(IntegralImageView& intImg,
                       const int octaves = OCTAVES,
                       const int intervals = INTERVALS,
62
                       const int sampling = INT_SAMPLE,
                       const float threshold = THRES);

    FastHessianLocator(const int octaves = OCTAVES,
                       const int intervals = INTERVALS,
67
                       const int sampling = INT_SAMPLE,
                       const float threshold = THRES);

virtual ~FastHessianLocator();

72
void setSourceIntegralImg(IntegralImageView& intImg
);
void setParameters(const int octaves,
                   const int intervals,
                   const int sampling,
                   const float threshold);

77
// Outputs in the given vector the interest points
// detected in the
// image, having only the location and scale
// information
// computed for them.
virtual void locatePOIInImage(std::vector<Poi>&
                             iPts_out);
82
};
}

```

```

    }

87 #include "FastHessianLocator.hpp"
    #endif

```

```

ObjectPoiStorage.h : Stores the interest points for objects
to be detected

#ifndef _OBJECTPOISTORAGE_H_
2 #define _OBJECTPOISTORAGE_H_

#ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
7    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif
#define FLT_MAX      3.402823466e+38F      /* max value */

12 #include "IPOIA�rithm.h"

namespace cyrus{
    namespace alg{

17     class CYTRUSALGLIB_API ObjectPoiStorage{
        // TODO: add thread safetyness to singleton & other members.
        private:
            static ObjectPoiStorage* instance; // singleton instance
22            static std::vector<std::vector<Poi>> _registeredObjects;

            ObjectPoiStorage();
            virtual ~ObjectPoiStorage();

27            static float comparePOIs(Poi& p1, Poi& p2);

        public:

32            static ObjectPoiStorage* getInstance();
            static int registerObject(int startX, int startY, int width
                , int height, IPOIA�rithm* poiSource);
            static void matchObjects(std::vector<Poi>& iPts);
        };

37    }
}

```

```
#endif
```

```

poi.h: Defines data structures for the points of interest used
in algorithms like SURF or SIFT

#ifndef _POI_H_
#define _POI_H_

#ifndef CYTRUSALGLIB_API
5   #ifdef CYTRUSALGLIB_EXPORTS
      #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
      #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
10 #endif

class CYTRUSALGLIB_API Poi{
public:
15   float x, y;
   float scale;
   float orientation;
   int laplacianSign;
   int descriptorSize;
20   float* descriptor;
   float dx, dy;
   //int clusterIndex; - not used yet
   int matchesObjectNr;
   float matchedDistance;
25
Poi(int descrSize=64) : orientation(0), descriptorSize(descrSize)
{
30   descriptor=new float[descriptorSize];
   matchesObjectNr=-1;
   matchedDistance=-1;
}
};

#endif

```

SurfDescriptor.h : Describe POI's matrix. Some parts are adapted from Bay's SURF implementation/notes:C. Evans, Research Into Robust Visual Features, MSc University of Bristol, 2008.

```

1 #ifndef _SURFDESCRIPTOR_H_
#define _SURFDESCRIPTOR_H_

6 #ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif
11 #define PI 3.1415926535897932384626433832795

16 #include <vector>
#include <iostream>
# include "math.h"
# include "poi.h"
# include "IDescriptor.h"

21 namespace cytrus{
    namespace alg{
        const float pi = float(PI);

        template <typename IntegralImageView>
        class SurfDescriptor : public IDescriptor{
26
            private:
                void getOrientation(Poi& pt);
                void getDescriptor(Poi& pt);

31
                inline float gaussian(int x, int y, float sig);
                inline float gaussian(float x, float y, float sig);
                inline float haarX(int row, int column, int s);
                inline float haarY(int row, int column, int s);
36
                float getAngleFromOrigin(float x, float y);

39
            protected:
                IntegralImageView* _img;
41
                int i_width, i_height;
                bool _oriented;

44
            public:
                SurfDescriptor(IntegralImageView& intImg, bool
                    oriented=true);
46
                // Delayed image init constructor. Call
                // setSourceIntegralImg to set image.

```

```

SurfDescriptor(bool oriented=true);

51   virtual ~SurfDescriptor() {}

      void setSourceIntegralImg(IntegralImageView& intImg
          );
      void setParameters(const int octaves,
                          const int intervals,
                          const int sampling,
                          const float threshold);

56   virtual void getDescriptorsFor(std::vector<Poi>&
                                  iPts_out);
};

61 }

#include "SurfDescriptor.hpp"

#endif

```

```

SurfAlg.h : The implementation for the SurfAlgorithm

#ifndef _SURFALG_H_
#define _SURFALG_H_

5 #ifndef CYTRUSALGLIB_API
    #ifdef CYTRUSALGLIB_EXPORTS
        #define CYTRUSALGLIB_API __declspec(dllexport)
    #else
        #define CYTRUSALGLIB_API __declspec(dllimport)
    #endif
#endif

10 #endif

15 #include <vector>
#include "IImageConsumer.h"
#include "IImageSource.h"
#include "IPOIAlgorthm.h"
#include "ObjectPoiStorage.h"
#include "poi.h"

20 using namespace cytrus::cameraHAL;
using namespace cytrus::alg;

25 namespace cytrus{
    namespace alg{
        class CYTRUSALGLIB_API SurfAlg : public IPOIAlgorthm{
            private:

```

```

    volatile int _pWidth, _pHeight;
    volatile bool _isCustomPrelSize;
    static ObjectPoiStorage* _store;

30   public:
        SurfAlg(IImageSource* imgSrc, POIAlgResult
                  outputFunc, int index);
        ~SurfAlg();

        virtual void processImage(unsigned long dwSize,
                                  unsigned char* pbData);

35   virtual bool setProcessingSize(int newWidth, int
                                  newHeight);

        //Actions to take when the image from the source
        //changes size
        virtual void onSourceSizeChange();

40   };
};

#endif

```

```

CytrusManagedLib.h Assures .NET managed camera wrappers for
the unmanaged cytrus code

#pragma once
#include "CytrusAlgLib.h"
#include <list>
#include <iostream>

5   using namespace System;
using namespace System::Collections::Generic;
using namespace System::Collections::.ObjectModel;
using namespace System::Runtime::InteropServices;
using namespace System::Windows::Media;

10  using namespace cytrus::cameraHAL;
using namespace cytrus::alg;

15  namespace cytrus {
    namespace managed{

        delegate void RenderResultCallbackProc(int dwSize, unsigned
                                              char* pbData, int index);
        delegate void NewImageCallback();
    }
}

```

```

public delegate void ImageCaptureCallback(array<byte>^ pbData,
                                         List<Poi_m>^ poiData);

public ref struct OutputMode{
    String^ modeName;
    PixelFormat^ pixelFormat;

    virtual String^ ToString() override
    {
        return modeName;
    }
};

//public delegate void OutputModeCallback(OutputMode^ newMode);
// outputMode change events (not used)

// Managed class that is responsible for camera and capture
// management. It uses the unmanaged classes from cytrus::alg
// and cytrus::hal
// for running the SURF algorithm in real time on the captured
// video.
//
// This class also implements coarse-grained parallelism,
// dispatching work on multiple threads (there are multiple
// separate instances of the algorithm working in the same time
// )
public ref class CameraMgr
{
private:
    static GCHandle gch, nigch;
    static int thNr=0;
    //static int lastdwSize; // outputMode change events (not
    // used)
    std::list<IPOIAlgAlgorithm*>* alg_ProcessingPool;
    Dictionary<int, int>^ threadIndexes;

    RenderResultCallbackProc^ fPtr;
    NewImageCallback^ newImage;
    DirectShowCameraSource* cs;
    ObservableCollection<String>^ cList;

    ObservableCollection<OutputMode>^ outputModes;
    POIAlgResult result;
    NewImageAvailableCallback newImageAvailable;

    void callImageCaptureEvent(int dwSize, unsigned char* pbData,
                               int index);
    void newImageAvailableEvent();
    void cameraNotifyConsumers(Object^ o);
}

```

```

public:
    int _camWidth, _camHeight;

65   // Treat this event to receive the images from the selected
        // camera
    // as an array of bytes.
    event ImageCaptureCallback^ onImageAvailableForRendering;
    //event OutputModeCallback^ onOutputModeChange; // outputMode
        // change events (not used)

70   CameraMgr();
    !CameraMgr();

    void selectCamera(int i);
    void refreshCameraList();
75   ObservableCollection<String^>^ getCameraList();
    void showPropertiesDialog(IntPtr window);

    void setActiveOutputMode(int modeIndex);
    ObservableCollection<OutputMode^>^ getOutputModesList();
80   bool setProcessingSize(int width, int height);

    void startCapture();
    void stopCapture();

85   };
}
}

```

ImageFileManaged.h Assures .NET managed wrappers for the image file unmanaged cytrus code

```

#pragma once
2 #include "CytrusAlgLib.h"
#include "CytrusManagedLib.h"

using namespace System::Runtime::InteropServices;
using namespace cytrus::cameraHAL;
7 using namespace cytrus::alg;

namespace cytrus {
    namespace managed{
12
        public ref class ImageFileManager
        {
            private:

```

```

17     GCHandle gch;
18     IPOIAlgAlgorithm* alg;
19     bool isInvalid;
20
21     RenderResultCallbackProc^ fPtr;
22     FileImageSource* fs;
23     char* filePath;
24
25     ObservableCollection<OutputMode^>^ outputModes;
26     POIAlgResult result;
27     NewImageAvailableCallback newImageAvailable;
28
29     void callImageCaptureEvent(int dwSize, unsigned char* pbData,
30                               int index);
31
32     public:
33
34     property int _imgWidth
35     {
36         int get()
37         {
38             if (fs!=NULL)
39                 return fs->width;
40             return 0;
41         }
42     }
43
44     property int _imgHeight
45     {
46         int get()
47         {
48             if (fs!=NULL)
49                 return fs->height;
50             return 0;
51         }
52     }
53
54     // Treat this event to receive the images from the selected
55     // camera
56     // as an array of bytes.
57     event ImageCaptureCallback^ onImageAvailableForRendering;
58
59     ImageFileMgr();
60     !ImageFileMgr();
61     ~ImageFileMgr();
62
63     void setActiveOutputMode(int modeIndex);
64     ObservableCollection<OutputMode^>^ getOutputModesList();
65     bool setProcessingSize(int width, int height);

```

```
void setImagePath(String^ path);

void startImageProcessing();
void freeResources();

67
//returns the object index
int registerObject(int startX, int startY, int width, int
height);

};

72
}
```

Lista de simboluri și prescurtări

Prescurtare	Descriere	Definiție
SIFT	Scale Invariant Feature Transform	page 9
DOG	Difference of Gaussian	page 10
SURF	Speeded-Up Robust Features	page 13

Listă de figuri

1.1	Localizarea punctelor de interes	3
1.2	Descrierea punctelor de interes	3
2.1	Potrivire bazată pe tipare / Potrivire bazată pe trăsături	6
2.2	Detectorul Harris (muchii și colțuri)	8
2.3	Variată rezultatelor detectorului Harris la scalări	9
2.4	SIFT: Detectarea minimelor și maximelor locale	11
2.5	SIFT: Procesul de determinare al descriptorului	14
2.6	SURF: filtre de mediere în comparație cu filtrele obținute prin deriva- rea de ordin 2 a nucleelor Gauss	15
2.7	Calculul sumei intensității pixelilor folosind imagini integrale.	16
2.8	Construirea spațiului scalărilor, SURF/SIFT [Evans, 2009]	16
2.9	Wavelet-uri Haar	17
2.10	SURF: Determinarea orientării pentru un punct de interes [Evans, 2009]	18
3.1	Arhitectura generală a aplicației	24
3.2	Structura nivelului de achiziție a imaginilor	25
3.3	Structura nivelului de prelucrare a imaginilor	26
3.4	Structura nivelului de administrare a resurselor de procesare	28
3.5	Modelul de paraleлизare al activităților	29
4.1	Structura de implementare, prezentând modulele externe rezultate .	32
4.2	Interfața grafică	46
4.3	Interfața grafică, definirea de obiecte	47
6.1	Detectia obiectului rotit relativ la imaginea de referință	54
6.2	Detectia obiectului în prezența unor modificări de scală	55
6.3	Detectia obiectului în prezența unor modificări de luminozitate	56
6.4	Detectia obiectului în prezența unor transformări multiple	56
6.5	Detectia obiectului la schimbarea rezoluției	57
6.6	Detectia de obiecte multiple	57

6.7	Detectia eronata a unui punct de interes	58
6.8	Analiza clasică a deformărilor în cazul studiului comparativ al protezelor femurale	60
A.1	cytrus	65
A.2	cytrus::cameraHAL	66
A.3	cytrus::alg	68
A.4	cytrus::managed	69
A.5	Diagramă de secvență pentru procedura de achiziție a imaginilor (simplificat)	70
B.1	Nucleul Gauss	72
B.2	Detectia unor muchii folosind trecerile prin 0 ale derivatei a doua . . .	73
B.3	Comparatia LoG/DoG (eng. Laplacian of Gaussian/Difference of Gaussian)	73
B.4	Nucleu Gaussian, derivata L_{xx}	74
B.5	Nucleu Gaussian, derivata L_{yy}	74
B.6	Nucleu Gaussian, derivata L_{xy}	74

Glosar

blob, 6

design patterns, 23
detector Harris, 7, 10
DOG, 10

feature matching, 6

nucleu gaussian, 10

scale space, 10
SIFT, 9, 13, 21
spațiu scalarilor, 10
SURF, 13, 21

template matching, 5

șablon de proiectare, 23