LAPPEENRANNAN TEKNILLINEN YLIOPISTO
TEKNISTALOUDELLINEN TIEDEKUNTA
TIETOTEKNIIKAN KOULUTUSOHJELMA

KANDIDAATINTYÖ

KARI JYRKINEN

OLIOPOHJAINEN KONENÄKÖSOVELLUSKEHYS

Kandidaatintyön aihe on hyväksytty 6.4.2010.

Työn tarkastajana toimii professori, fil. tri Matti Heiliö.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Kari Jyrkinen

Oliopohjainen konenäkösovelluskehys

Kandidaatintyö

2010

30 sivua, 7 kuvaa ja 1 liite

Tarkastaja: Professori Matti Heiliö

Hakusanat: oliopohjainen sovelluskehys, konenäkö, kuvankäsittely, hahmontunnistus
Keywords: object-oriented software framework, machine vision, image processing, pattern recognition

Monet teollisuuden konenäkö- ja hahmontunnistusongelmat ovat hyvin samantapaisia, jolloin prototyyppisovelluksia suunniteltaessa voitaisiin hyödyntää pitkälti samoja komponentteja. Oliopohjaiset sovelluskehykset tarjoavat erinomaisen tavan nopeuttaa ohjelmistokehitystä uudelleenkäytettävyyttä parantamalla. Näin voidaan sekä mahdollistaa konenäkösovellusten laajempi käyttö että säästää kustannuksissa.

Tässä työssä esitellään konenäkösovelluskehys, joka on perusarkkitehtuuriltaan liukuhihnamainen. Ylätason rakenne koostuu sensorista, datankäsittelyoperaatioista, piirreirrottimesta sekä luokittimesta. Itse sovelluskehyksen lisäksi on toteutettu joukko kuvankäsittely- ja hahmontunnistusoperaatioita. Sovelluskehys nopeuttaa selvästi ohjelmointityötä ja helpottaa uusien kuvankäsittelyoperaatioiden lisäämistä.

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Degree Program in Information Technology

Kari Jyrkinen

Object Oriented Framework for Machine Vision

Bachelor's Thesis

2010

30 pages, 7 figures, and 1 appendix

Supervisor: Professor Matti Heiliö

Keywords: object-oriented software framework, machine vision, image processing, pattern recognition

Many industrial machine vision and pattern recognition problems are rather similar. When implementing prototype applications to solve these problems, it is possible to reuse existing components. Object-oriented software frameworks offer an excellent way to speed up the software development by improving the reusability. In this way, it is both possible to enable wider use of machine vision applications and save in the development costs.

In this thesis I present a machine vision framework, which follows the pipeline architecture. The high-level structure is composed of sensor, data processors, feature extractor, and classifier. In addition to the framework, a set of basic image processing and pattern recognition operations are implemented. The developed software framework speeds up the programming work and makes it easier to add new image processing operations.

ALKUSANAT

Suuret kiitokset työn aiheesta, mielenkiintoisesta kesätyöprojektista tietotekniikan osaston tietojenkäsittelytieteen laitoksella ja käytännön työn ohjauksesta professoreille Ville Kyrki ja Joni Kämäräinen sekä kandidaatintyön tarkastuksesta professori Matti Heiliölle. Itse ohjelmointityö on toteutettu jo aikaisemmin, suuresti "rakastamani"kirjallinen raportointi nyt, kiitos tutkintorakenneuudistuksen. No, parempi myöhään kun ei milloinkaan.

Erityiskiitokset kummipojalleni Lassi Puraselle hienosta junaradasta, josta otettua kuvaa on käytetty tässä työssä yhtenä esimerkkinä. #plop- ja #esa-kanavat sekä muutamat muut tutut ovat kiitettävästi auttaneet siinä, ettei erikoistyö/kandidaatintyö päässyt unohtumaan – saatatte mahdollisesti ehkä saadakin ne valmistujaiset.

SISÄLLYSLUETTELO

LYHENTEET

C++	olio-ominaisuudet sisältävä ohjelmointikieli
CMY	värijärjestelmä, jossa värit esitetään syaanin, magentan ja keltaisen värin intensiteettiarvojen avulla
GTK+	Gimp Toolkit, graafinen käyttöliittymäkirjasto
HSI	värijärjestelmä, jossa värit esitetään värisävyn, värikylläisyyden ja intensiteetin avulla
MPI	message passing interface, rinnakkaislaskennan viestinvälitysrajapinta
RGB	värijärjestelmä, jossa värit esitetään punaisen, vihreän ja sinisen intensiteettiarvojen avulla

1 JOHDANTO

Erilaisia konenäkö- ja hahmontunnistusongelmia ratkaistaessa työvaiheet ovat hyvin samankaltaisia. Aluksi data, esimerkiksi sarja kuvia, pitää lukea tietokoneen muistiin - joko tiedostosta, suoraan kamerasta tai digitointikortin kautta. Sen jälkeen kuvia usein esikäsitellään tai muokataan helpommin käsiteltävään muotoon käyttämällä erilaisia suodattimia ja muuntimia. Tämän jälkeen kuvista etsitään haluttuja piirteitä, joiden avulla data voidaan luokitella.

Prototyyppisovellusta kehitettäessä voidaan käyttää apuna Mathworksin Matlab-ohjelmistoa [16] ja sen tarjoamaa kuvankäsittelykirjastoa, mutta varsinaista sovellusta ohjelmoitaessa Matlabin maksullisuus ja hitaus erityisesti suuria datamääriä käsiteltäessä puoltavat alemman tason ohjelmointikieliä. Matlab ei myöskään ole yhtä monipuolinen kuin modernit oliokielet. Aiempien sovellusten ohjelmakoodia voi usein käyttää uudelleen, mutta versionhallinnan penkominen voi olla työlästä. Siksi olisi hyödyllistä, että usein tarvittavat ohjelmakomponentit löytyisivät valmiina, ja jokainen voisi lisätä kehittämänsä uuden komponentin helposti muiden käytettäväksi.

Uudelleen käytettävyys ja kapselointi ovat oliopohjaisen ohjelmoinnin perusperiaatteita. Luokkien perintä helpottaa osaltaan uusien konenäkö- ja hahmontunnistuskomponenttien ohjelmointia, kun kaikkia ominaisuuksia ei välttämättä ole tarpeen kirjoittaa uudelleen. Ylemmän tason luokat tarjoavat muutenkin mallin yhtenäiselle käyttöliittymälle. Ohjelmointiympäristö kannattaakin toteuttaa jotain olio-ohjelmointikieltä käyttäen.

1.1 Tavoitteet ja työn rajaus

Itse ohjelmointityön tavoitteena oli kehittää mahdollisimman hyvin olioparadigmaa [1] tukeva sovelluskehys konenäkö- ja hahmontunnistusongelmien tarpeisiin. Useimmat hahmontunnistusongelmatkin käsittelevät kuvista saatavaa dataa, mutta ympäristön tuli mahdollistaa myös muunlaisen datan, kuten äänen tai lämpötilakäyrien, käsittely.

Tämän lisäksi ohjelmoitiin joukko peruskomponentteja, joiden Windows-toteutus Matrox Imaging Library 6.0:aa [17] hyödyntäen kuului tehtäviini. Itse oliomallia suunniteltiin työryhmässä, ja oliokehyksen implementointi tehtiin yhdessä komponenttien Linux-versioiden kehittäjän kanssa.

1.2 Työn rakenne

Seuraavassa luvussa perehdytään oliosuunnittelun ja -ohjelmoinnin peruskäsitteisiin sekä sovelluskehyksiin. Muutamia aiemmin toteutettuja järjestelmiä esitellään lyhyesti. Yleisten oliosuunnittelumenetelmien lisäksi esitellään erilaisia formaaleja menetelmiä. Kolmas luku esittelee kuvankäsittelyn perusteita: yleisimpiä suodattimia, värimuunnoksia ja reunaviivojen etsintämenetelmiä matemaattisine perusteineen. Neljännessä luvussa esitellään toteutettu sovelluskehys komponentteineen sekä ympäristön suunnittelussa tehdyt valinnat. Lopuksi tarkastellaan toteutetun sovelluskehyksen toimivuutta sekä mahdollisia jatkokehitystoimenpiteitä.

2 OLIOPOHJAISET SOVELLUSKEHYKSET

Sovelluskehys (ohjelmistokehys, engl. software framework) tarkoittaa olio-ohjelmoinnin yhteydessä kiinteästi toisiinsa liittyvien luokkien ja rajapintojen kokoelmaa, joka toteuttaa tietyn ohjelmistoperheen perusarkkitehtuurin [15]. Sovelluskehyksien avulla ohjelmistojen koko rakenteen, ei pelkästään luokkien, uudelleen käyttöä voidaan parantaa. Ohjelmistokehys voi sisältää ohjelmakomponenttikirjaston lisäksi apuohjelmia, skriptikielen, graafisen käyttöliittymän sekä muita kehitys- ja integrointityössä tarvittavia sovelluksia.

Kehyksestä saadaan yksittäinen sovellus tai komponentti erikoistamalla. Tätä toimenpidettä varten kehyksessä on erikoistamisrajapinta, joka ei ole välttämättä ulkoisesti määritelty rajapinta, vaan pikemminkin määritelmä dokumentaatiossa. Kehyksen erikoistamiseen käytetään perintää, luokkien rajapintoja ja yleispäteviä olioita.

Oliokehys on kehitetty tietyntyyppistä käyttötarkoitusta varten – toisin kuin oliokielten luokkakirjastot, jotka tarjoavat yleiskäyttöisiä algoritmeja. Oliopohjaiset sovelluskehykset ovat lupaava tekniikka kustannusten pienentämiseksi sekä tuottavuuden ja laadun parantamiseksi [9, 8]. Näiden tavoitteiden saavuttamiseksi täytyy ongelma-alueeseen perehtyä hyvin, jotta sovelluskehyksestä tulee riittävän joustava, helppokäyttöinen ja oikeellinen.

2.1 Aiemmat tutkimukset

Bowskill, Katz ja Cattez ovat kehittäneet sovelluskehyksen elektroniikkateollisuuden tuotannon joustavuuden ja laadun parantamiseksi [4]. Konenäköjärjestelmien kehitys on kallista, jos työ aloitetaan aina alusta, ja siten taloudellista vain suuria volyymejä käsiteltäessä. Kehittämällä sovelluskehys, joka on riippumaton ohjelmiston lopullisesta toteutuksesta, mahdollistetaan uusien tekniikoiden joustava käyttöönotto ja erilaisten sovellusten integrointi, ja siten voidaan järjestelmien kustannuksia alentaa ja mahdollistaa niiden laajempi käyttö. Oliopohjainen lähestymistapa tukee hyvin näitä tavoitteita.

Amatriain, Arumi ja Garcia esittelevät artikkelissaan *A framework for efficient and rapid development of cross-platform audio applications* [2] C++-kielellä toteutetun oliopohjaisen CLAM-sovelluskehyksen, joka on tarkoitettu tutkimuskäyttöön sekä audio- ja musiikkisovellusten kehittämiseen. Ohjelmistoon on toteutettu erityyppisten audioformaattien luku-, analysointi-, muunnos-, käsittely-, tallennus- ja soitto-operaatioita. Tavoitteena on

ollut kehittää ympäristö nopeiden prototyyppien luomiseksi. Ongelmankuvaus ja ratkaisu ovat hyvin samantapaisia kuin tässä työssä, vaikka data onkin audiomuotoista ja sovelluskehys pidemmälle kehitetty.

Biolääketieteessä käytetään laajalti hahmontunnistustekniikoita. Muun muassa magneettiresonanssikuvaus tuottaa suuren määrän dataa, jonka analysointiin tarvitaan tehokkaita ja helppokäyttöisiä ohjelmia. Scopira [21] on tätä tarkoitusta varten kehitetty C++-kielinen, avoimen lähdekoodin sovelluskehys. Sen tarjoama ohjelmointirajapinta helpottaa C- tai C++-kielisten biolääketieteellisten kuvankäsittelykirjastojen hyödyntämistä. Lisäksi rinnakkaislaskentaan tarjotaan MPI-viestinvälityskirjaston [19] tuki, ja graafisten käyttöliittymien toteutukseen GTK+-pohjaisia valmiita ohjelmamoduuleja.

Cheung ja Ip ovat suunnitelleet ja toteuttaneet geneerisen oliopohjaisen sovelluskehyksen kuvien hakuun niiden sisällön perusteella [5]. Järjestelmä on riippumaton kuvien ja tietokannan tyypistä, etsittävistä piirteistä sekä käyttöjärjestelmästä. Ohjelmointikieleksi on valittu C++. Kuvien piirteet ovat tällaisessa sovelluksessa oleellisia ja siten oleellinen osa luokkien ja ohjelmistoarkkitehtuurin suunnittelua ja toteutusta.

2.2 Oliosuunnittelun perusteet

Oliosuunnittelu ja olio-ohjelmointi keskittyvät kuvaamaan itse ongelmaa proseduraalisen ohjelmoinnin toiminto-orientoituneen lähestymistavan sijaan [6]. Tämä on yksi olennainen syy siihen, miksi tietyn ongelma-alueen tarpeisiin kehitetyt sovelluskehykset ovat oliokielillä toteutettuja. Oliosuunnittelussa, ja erityisesti yleiskäyttöisiä sovelluskehyksiä toteutettaessa, ongelmaan perehtyminen ja huolellinen analysointi on tärkeää.

Oliosuunnittelussa mallinnetaan järjestelmää olioina ja niiden välisenä vuorovaikutuksena. Olio-orientoitunut ajattelu onkin hyvin lähellä sitä, miten ihmiset hahmottavat ympäröivän maailman: olioina ja olioluokkina, niiden välisenä hierarkiana ja vuorovaikutuksena [12]. Oliopohjaisten järjestelmien iteratiivinen suunnittelu ja kehitys on helpompaa, sillä olioiden sisäisen rakenteen muuttaminen vaikuttaa muihin komponentteihin vain olion rajapinnan kautta.

Järjestelmäkehitys voidaan jakaa muun muassa seuraaviin vaiheisiin: analysointi, systeemisuunnittelu, oliosuunnittelu, toteutus ja testaus [12, 18]. Olioperusteinen ajattelu tulee mukaan jo analysointivaiheessa. Järjestelmän suunnitteluun on syytä panostaa, sillä mitä

myöhemmässä vaiheessa tehdään isoja, arkkitehtuuriin vaikuttavia muutoksia, sitä enemmän se maksaa. Lähtökohtana suunnittelulle ovat järjestelmälle asetettavat vaatimukset.

2.2.1 Oliokäsitteitä

Olio on abstraktio ongelma-alueen osasta, ja se kuvastaa järjestelmän sisältämää informaatiota ja sen välitystä [6]. Se on kokonaisuus, joka kykenee tiettyihin toimintoihin sekä muistamaan niihin liittyvät tilamuutokset. Olioilla voi olla staattisia suhteita, jolloin oliot lähinnä tietävät toisistaan, sekä dynaamisia suhteita, jolloin oliot kommunikoivat keskenään [12]. Olio voi myös muodostua useammasta oliosta, jolloin sitä kutsutaan aggregaatiksi.

Luokka on oliota yleisempi käsite, eräänlainen oliotemplaatti. Se kuvaa samantyyppisten olioiden ominaisuuksia, niille yhteisiä toimintoja ja riippuvuuksia muihin olioryhmiin [18]. Luokan avulla voidaan useampaan olioon liittää joukko samoja ominaisuuksia. Tiettyyn luokkaan kuuluvaa oliota kutsutaan instanssiksi. Luokka kuvaa instanssin toimintojen ja informaation rakenteen, mutta itse sisältö määräytyy instanssin operaatioiden kautta [12]. Oliopohjaisessa järjestelmässä kaikki oliot kuuluvat johonkin luokkaan.

Monilla luokilla on joukko yhteisiä piirteitä, vaikka luokat eroavatkin. Tällöin on mahdollista luoda yleisempi, ylemmän tason luokka, joka määrittelee yhteiset ominaisuudet, ja periyttää alaluokat tästä. Uudet luokat perivät yläluokkansa toiminnot ja tietorakenteet, ja niihin tarvitsee määrittää vain luokkien spesifiset ominaisuudet. Perintä on yksi tapa uudelleenkäyttää ohjelmakoodia [12].

Kapselointi tarkoittaa luokkien sisäisen rakenteen piilottamista muilta luokilta. Luokkien tietoon pääsee käsiksi rajatun käyttöliittymän kautta, jolloin luokan sisältämän tiedon muuttaminen on kontrolloitua. Luokan sisäistä rakennetta voi muuttaa melko vapaasti muun järjestelmän toiminnan tästä kärsimättä, niin kauan kuin käyttöliittymä ei muutu. Käyttöliittymän tarjoavat funktiot, luokan metodit, kannattaakin suunnitella mahdollisimman yleiskäyttöisiksi.

2.2.2 Oliosuunnittelumenetelmiä

Oliosuunnittelu on järjestelmän toiminnan ja informaation analysoinnin iterointia. Järjestelmän analysointivaiheessa etsitään oliot ja luokitellaan ne samankaltaisuuden mukaan, kuvataan olioiden välinen vuorovaikutus sekä suunnitellaan olioiden operaatiot ja sisäinen rakenne. Ohjelmointivaiheessa yleinen malli sovitetaan valitulle kehitysympäristölle ja ohjelmointikielelle. Tässä vaiheessa pitää ottaa kantaa myös kohdejärjestelmän asettamille vaatimuksille, kuten muistinkulutukseen, luotettavuuteen ja vasteaikaan.

Oliomallin tuottamiseen on kehitetty lukuisia erilaisia menetelmiä, kuten olioanalyysi (Object-Oriented Analysis, OOA) [6], oliosuunnittelu (Object-Oriented Desing, OOD) [3], oliopohjainen sovelluskehitys (Object-Oriented Software Engineering, OOSE) [12] sekä oliomallinnustekniikka (Object Modeling Technique, OMT) [18]. Analysoinnin perusvaiheet löytyvät kaikista eri menetelmistä, mutta asioita painotetaan hieman eri tavalla.

Olioanalyysi jakaantuu viiteen vaiheeseen. Liikkeelle lähdetään ongelma-alueen olioiden ja luokkien etsimisellä. Toisessa vaiheessa ryhmitellään luokat niin, että niistä muodostuu isompia toisiinsa liittyviä rakenteita. Seuraavaksi määrittele jokaisen ryhmän tarkoitus eli mitä osaa järjestelmästä luokat kuvastavat. Lopuksi määritellään luokkien sisältämä data ja niiden tarjoamat palvelut, metodit. Oliosuunnittelu jatkaa tästä määrittämällä muun muassa toteutuksessa tarvittavia luokkia dialogeja sekä tehtävien ja datan hallintaa varten.

Oliopohjainen sovelluskehitys perustuu viiteen malliin. Vaatimusmalli pyrkii kuvaamaan järjestelmän toiminnalliset vaatimukset käyttäjän näkökulmasta. Tässä vaiheessa kuvataan käyttötapaukset, käyttöliittymä ja kohdealue. Analyysimalli määrittää systeemin karkean oliorakenteen yleisellä tasolla. Suunnittelumalli pyrkii tarkentamaan aiemmin luodun oliomallin rakennetta toteutusympäristöä ajatellen, muun muassa vuorovaikutusdiagrammin avulla. Implementaatiomalli kuvaa järjestelmän toteuttamiskelpoisesti uudelleenmäärittämällä ja tarkentamalla oliomallia. Viimeinen malli, testimalli pyrkii verifioimaan järjestelmän toiminnan.

Oliomallinnustekniikassa on kolme vaihetta. Ensimmäisessä vaiheessa kuvataan olioiden staattinen rakenne ja niiden väliset suhteet laatimalla oliomalli. Dynaamisella mallilla kuvataan järjestelmän tapahtumia ajan suhteen tilakoneiden avulla. Lopuksi funktionaalisella mallilla kuvataan datan kulkua ja muuttumista järjestelmässä.

3 KUVANKÄSITTELYN PERUSTEET

Konenäköön kuuluu oleellisena osana itse laitteisto: kamerat, valaistus ja kuvien digitaaliseen muotoon muuttaminen [14]. Kehittämämme oliokehys ei juurikaan ota kantaa itse datan keräämiseen, joten konenäkölaitteistoon liittyviä suunnittelunäkökohtia ei tässä työssä käsitellä. Seuraavaksi esitellään lyhyesti erilaisia toimenpiteitä, joita kuville joudutaan usein tekemään, ennen kuin dataa voidaan hyödyntää kunnolla.

Kuvankäsittelyssä muunnetaan kuvia toiseen muotoon. Erilaisia menetelmiä ovat muun muassa kuvien muokkaus suodattamalla tai terävöittämällä, muunnokset eri värijärjestelmien sekä kuva- ja taajuustason välillä, segmentointi ja pakkaus. Kuvankäsittelyalgoritmeja käytetään konenäköjärjestelmän alkuvaiheessa kuvan muokkaamiseksi paremmin käsiteltävään muotoon, minkä jälkeen voidaan käyttää erilaisia piirreirrottimia oleellisen informaation löytämiseen. Muun muassa hahmontunnistusta [20] käytetään saadun numeerisen tai symbolisen datan luokitteluun ja sitä kautta kohteiden tunnistamiseen.

3.1 Kuvien suodatus

Raakakuva ei usein ole sellaisenaan käyttökelpoinen, vaan intensiteetti saattaa vaihdella satunnaisesti tai valaistus voi olla epäoptimaalinen, jolloin kontrastierot jäävät pieniksi [13]. Kuvassa voi olla myös häiriöitä. Kuvaa voidaan parantaa käyttämällä erilaisia filttereitä.

Mustavalkoisissa kuvissa harmaasävyjä on yleisimmin käytössä 256, mutta arvot voivat olla epätasaisesti jakaantuneita. Pienet kontrastierot eivät välttämättä haittaa automaattisessa kuvankäsittelyssä, mutta ihmissilmä hahmottaa kuvan paremmin, kun koko intensiteettialue on käytetty tehokkaasti. Tämä on mahdollista normalisoimalla histogrammit [13]. Kun intensiteettiarvot vaihtelevat välillä $[a, b]$ ja mahdollinen arvoväli on $[z_1, z_k]$, voidaan jokaisen pikselin z uusi intensiteettiarvo z' laskea kaavasta

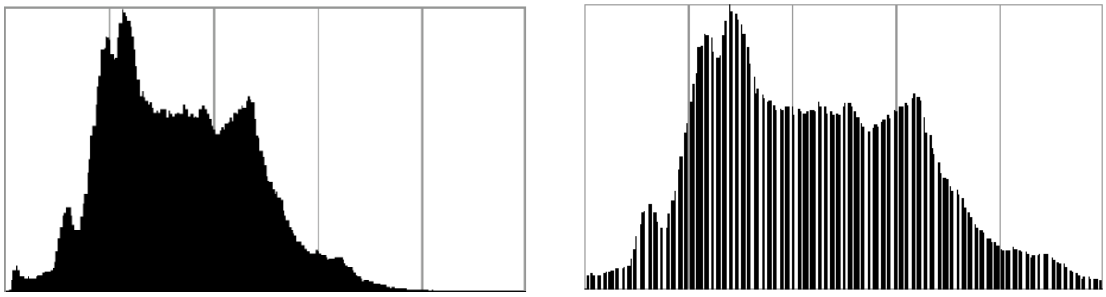
$$z' = \frac{z_k - z_1}{b - a}(z - a) + z_1. \tag{1}$$

Tällä tavalla laskettuna histogrammitasoitus jättää joitain harmaasävyjä käyttämättä eli histogrammiin jää välejä, kuten esimerkkikuvan 1 histogrammista 2(b) on havaittavissa. Jos haluttu harmaasävyjakauma tiedetään, voidaan arvot jakaa tasaisesti tähän a priori -

jakaumaan perustuen. Histogrammitasoitusta voi käyttää myös värikuvilla, esimerkiksi RGB-kuvilla tasoitus lasketaan vain jokaiselle värille erikseen.

(a) (b)

Kuva 1: Pienikontrastinen kuva (a) ennen histogrammitasoitusta ja (b) sen jälkeen.



(a) (b)

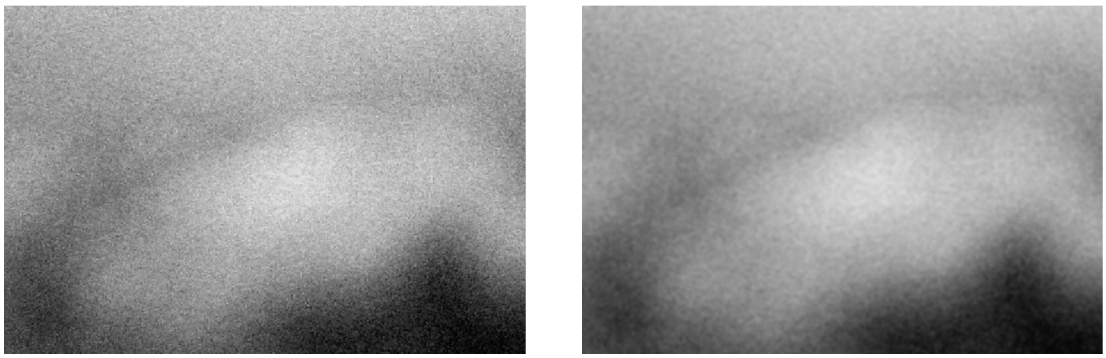
Kuva 2: Kuvia 1(a) ja 1(b) vastaavat histogrammit.

Suola ja pippuri -kohina tarkoittaa kuvissa joskus esiintyviä yksittäisiä valkeita ja mustia kuvapisteitä, jotka eivät kuulu kuvaan. Impulssikohinassa esiintyy vain vaalean intensiteetin häiriöpikseleitä, kun taasen gaussisessa kohinassa häiriöpisteiden intensiteettiarvojen jakauma noudattaa normaalijakaumaa. Kuvissa esiintyviä häiriöitä voidaan häivyttää, kun käytetään lineaarisia suodattimia, jotka hyödyntävät kuvapisteen naapuripikseleiden painotettua summaa mahdollisten virheiden korjaamiseksi [13]. Yksi yksinkertaisimmista suodattimista on keskiarvosuodin, jossa jokainen kuvapiste (i, j) lasketaan uudestaan naapuruston pikseleiden keskiarvona. 3×3-naapurustoa käyttäen kuvapisteiden uudet arvot lasketaan seuraavasti:

$$h(i,j) = \frac{1}{9} \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} f(k,l). \tag{2}$$

Käytetyn naapuruston koko vaikuttaa suodatuksen määrään. Mitä suurempaa naapurustoa käytetään, sitä suurempi on suodatuksen vaikutus. Laajempi naapurusto poistaa tehokkaammin kohinaa, mutta samalla heikentää kuvan terävyyttä ja vähentää yksityiskohtia. Tämä on helposti havaittavissa kokeilemalla. Kuvassa 3 on esitelty 5×5-suodatittimen vaikutusta. Myös painoja voidaan säätää tarvittaessa, kuitenkin niin että painomatriisi on symmetrinen sekä vaaka- että pystysuuntaan:

$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$
$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{8}$
$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$



(a) (b)

Kuva 3: (a) Osa alkuperäistä, hämärässä otettua kohinaista kuvaa sekä (b) 5×5–keskiarvosuodattimella käsitelty kuva.

Mediaanisuodatin ei hämärrä kuvan yksityiskohtia niin kuin keskiarvosuodatin [13]. Sen toiminta perustuu myös naapuruston käyttöön. Jokaisen kuvapisteen uusi arvo määräytyy naapuruston intensiteettiarvojen mediaanin perusteella. Mediaanisuodatin tehoaa parhaiten satunnaiseen kohinaan.

Gaussin suodattimen painot valitaan puolestaan Gaussin käyrän mukaan, minkä vuoksi suodin toimii parhaiten normaalijakaumaa noudattavaan kohinaan. Gaussin suodatin on tehokas alipäästösuodin niin kuva- kuin taajuustasossa, ja sitä käytetään yleisesti erilaisissa kuvankäsittelysovelluksissa [13]. Kuvatasoon tarvitaan kaksiulotteinen Gaussin funktio, joka on esitelty alla. Mitä suurempaa varianssin σ arvoa käytetään, sitä voimakkaampi suodatus.

$$g(i,j) = \frac{1}{M} \sum_{(k,l)\in N} e^{-\frac{k^2+l^2}{2\sigma^2}}. \tag{3}$$

3.2 Värijärjestelmämuunnokset

Mustavalkoisten kuvien digitaaliseen esittämiseen riittää yleensä kaksiulotteinen matriisi, mutta värikuvat tuovat kolmannen ulottuvuuden. Hyvin yleinen värijärjestelmä on ihmisnäköön [11] perustuva RGB, jossa värit kuvataan punaisen, vihreän ja sinisen intensiteettiarvoina. Valon spektriä voidaan kuvata kymmenillä tai sadoilla eri aallonpituuksille jakautuvilla komponenteilla, mutta yleisesti käytössä olevat värijärjestelmät tyytyvät värin esittämiseen kolmella arvolla. Värikomponentteihin perustuvista kuvista mustavalkoinen kuva saadaan laskemalla eri komponenttien keskiarvo.

CMY-värijärjestelmä esittää valon syaanin, magentan ja keltaisen värin, pigmentin, komponentteina [11]. Kun esimerkiksi keltaista pintaa valaistaan valkoisella valolla, ei pinta heijasta takaisin sinistä valoa, eli keltainen poistaa valosta sinisen komponentin. Värimuunnos RGB-järjestelmästä CMY-järjestelmään ja takaisin on yksinkertainen laskea kaavasta (4). Tätä värijärjestelmää käyttävät pääosin väritulostuslaitteet, eivät niinkään konenäkösovellukset.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix} \tag{4}$$

Kuvankäsittelyn kannalta kätevämpi värijärjestelmä on HSI-järjestelmä, jonka komponentit ovat värisävy, värikylläisyys eli värisävyn suhteellinen osuus muihin väreihin nähden sekä intensiteetti. Harmaasävyihin pohjautuvia algoritmeja voidaan käyttää suoraan rajoittumalla pelkästään intensiteettitasoon I, kun taasen erilaisten kuva-alueiden segmentointia väriin perustuen voidaan tehdä pelkästään värisävyn H perusteella, kunhan alhaisen värikylläisyyden S arvot jätetään huomiotta. [13]

RGB-kuvien muuntaminen HSI-järjestelmään ei ole ihan suoraviivaista, mutta usein tarpeellista. Muunnos voidaan tehdä normalisoiduista RGB-arvoista [11]. Jos värikylläisyys on 0, värisävy on määrittelemätön ja jos intensiteetti on 0, ei värikylläisyyttä voida määrittää. Värisävy H lasketaan käyttämällä kaavoja

$$H = \begin{cases} \theta & , kun\ R \leq G, \\ 360 - \theta & , kun\ R > G \end{cases} \tag{5}$$

ja

$$\theta = cos^{-1}\left\{\frac{[\,(R-G)+(R-B)\,]}{2\sqrt{(R-G)^2+(R-B)(G-B)}}\right\}. \tag{6}$$

Värikylläisyyden S ja intensiteetin I laskeminen on tämän jälkeen melko suoraviivaista:

$$S = 1 - \frac{3}{R+G+B}\,[\,min(R,G,B)\,] \tag{7}$$

$$I = \frac{R+G+B}{3}. \tag{8}$$

3.3 Reunaviivojen etsiminen

Kuvien muokkauksen jälkeen dataa pyritään vähentämään etsimällä sovelluksen kannalta oleelliset piirteet. Reunaviivojen etsintä on usein toteutettu piirreirrotusoperaatio. Reunaviiva määritetään kuvan intensiteetin huomattavana paikallisena muutoksena, usein epäjatkuvuuskohtana joko itse kuvan intensiteetin tai sen ensimmäisen derivaatan arvoissa [13]. Teräviä reunoja tai reunaviivoja kuvista harvoin löytyy, varsinkaan esikäsittelyn jälkeen, joten luotettavan reunaviivaoperaattorin löytäminen voi olla haastavaa.

Gradientti mittaa funktion muutoksen nopeutta. Kuvan diskreettien intensiteettiarvojen voidaan ajatella kuvaavan jonkun jatkuvan funktion arvoja tietyissä pisteissä, jolloin analogisesti voidaan kuvata myös gradienttia summittaisesti diskreettien arvojen avulla ja käyttää tätä muutosten havaitsemiseen kuvassa. Näin saadaan muodostettua erilaisia maskeja, joiden avulla reunaviivojen etsiminen on tietokoneelle nopea laskutoimitus. Etsimällä reunapisteitä sekä pysty- että vaakasuunnassa, löydetään myös muun suuntaiset viivat. Yksinkertaisimmillaan voidaan käyttää seuraavia maskeja:

-1	1
-1	1

1	1
-1	-1

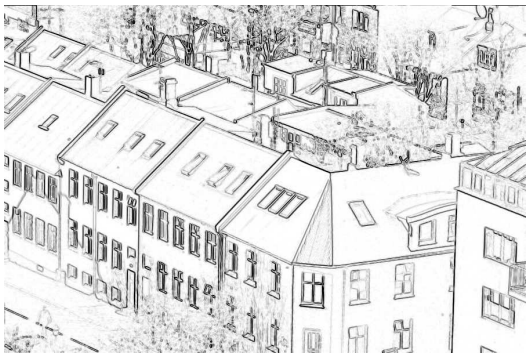
Edellä esitetyissä operaattoreissa on kuitenkin se ongelma, että ne laskevat reunapisteen kuvapisteiden väliin, vaikka olisi tärkeää löytää itse kuvapisteet, jotka ovat reunapisteitä. Siksi on parempi käyttää paritonta naapurustoa, kuten 3×3. Tällainen on muun muassa

Sobelin operaattori (alla), mutta vastaavia, erilaisiin matemaattisiin lähtökohtiin perustuvia operaattoreita esitellään kirjallisuudessa lukuisia [11, 13]. Suorien viivojen etsimiseen tehokas menetelmä on Hough-muunnos [7]. Esimerkki Sobelin operaattorin tuottamasta ääriviivakuvasta on esitetty kuvassa 4.

-1	0	1
-2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1

(a)



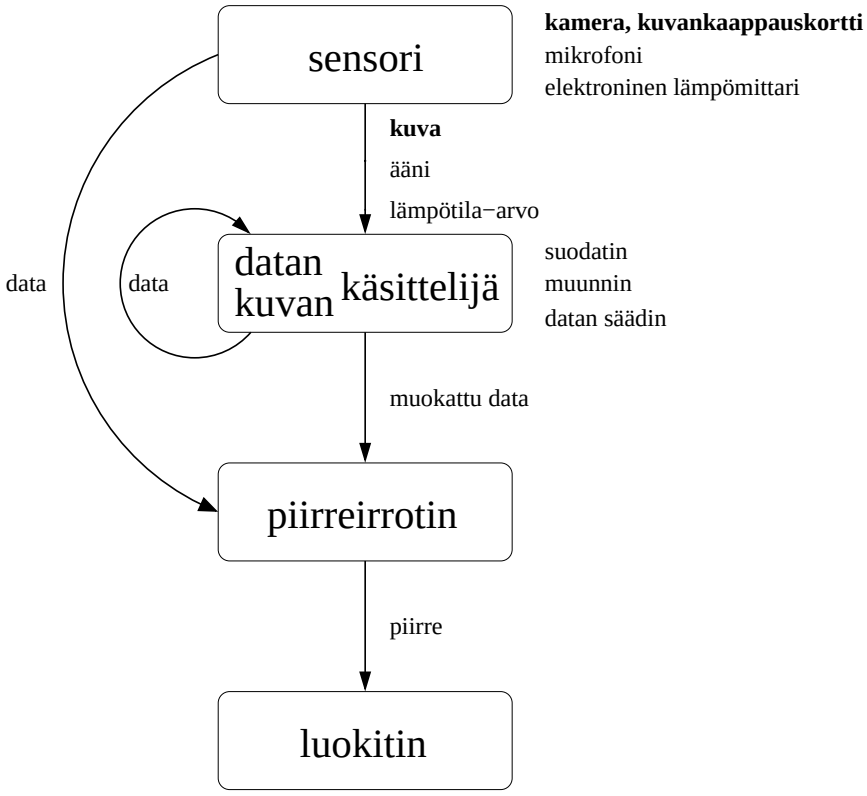
(b)

Kuva 4: (a) Alkuperäinen kuva ja (b) Sobelin maskin avulla löydetyt ääriviivat. Ääriviivakuva on histogrammitasoitettu, jotta tulos on helpommin nähtävissä.

4 MVFW-KONENÄKÖSOVELLUSKEHYS

MVFW-sovelluskehys suunniteltiin ja toteutettiin kesätyöprojektina silloisen Lappeenrannan teknillisen korkeakoulun tietotekniikan osastolla. Tarkoituksena oli nopeuttaa erilaisten konenäkö- ja hahmontunnistusongelmien prototyyppi- ja pilottisovellusten kehitystä. Matlab-ohjelmisto ja sen kuvankäsittelykirjasto ovat käteviä ongelman ratkaisua etsittäessä, mutta teollisuusprojekteissa maksullisen Matlabin käyttö ei ole usein järkevää, koska sovellukset voi kirjoittaa itsekin kohtuullisella vaivalla.

Toteutettaviksi ominaisuuksiksi valittiin kuvien tai yleisemmin datan luku, kuvankäsittelyoperaatiot, piirreirrotus ja hahmontunnistus sekä erityisesti näitä ylemmällä tasolla tukeva oliokehys. Sovelluksen rakenteeksi muotoutui jo alkuvaiheessa liukuhihnamainen arkkitehtuuri, sillä valitut operaatiot ovat pitkälti peräkkäisiä. Kuvassa 5 esitetty arkkitehtuuri on varsin yleisesti käytetty konenäkösovelluksissa.

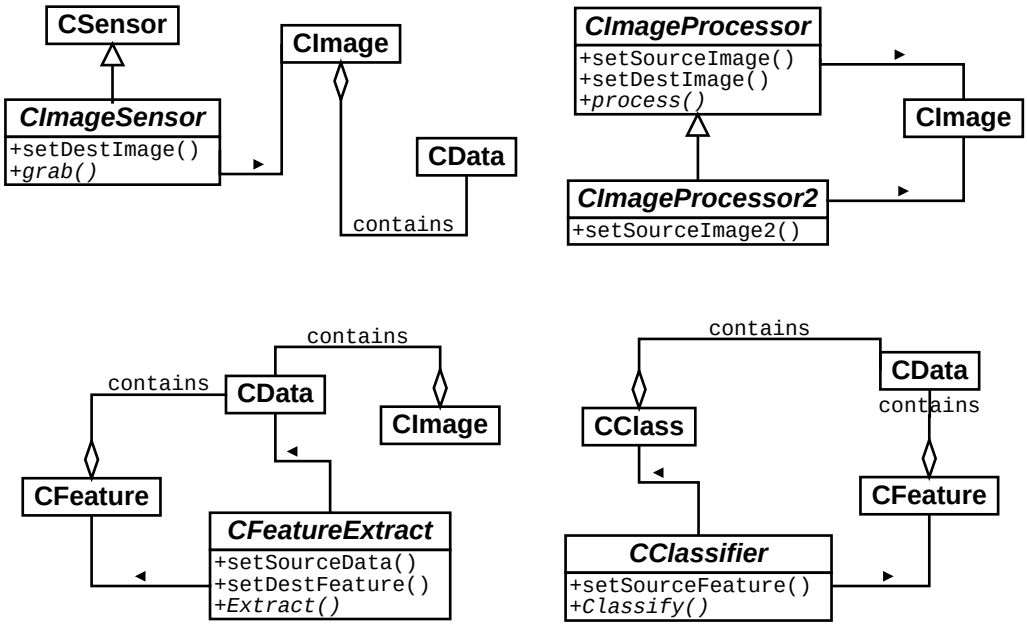


Kuva 5: MVFW-oliokehyksen arkkitehtuuri.

4.1 Sovelluskehyksen suunnittelu

Suunnittelua ja oliomallinnusta tehtiin yhdessä ohjaajien, Ville Kyrjen ja Joni Kämäräisen sekä toisen toteuttajan, Jarmo Ilosen kanssa. Sovellusalue on varsin rajattu ja rajapinnat yksinkertaisia, joten prototyypitys ja iteratiivinen kehitystapa toimi hyvin. Vaikka sovelluskehyksen yksityiskohtia muutettiin useaan kertaan projektin aikana, uudelleenohjelmointitarve pysyi kohtuullisen pienenä.

Olioanalyysissä olioiksi tunnistettiin sensori, data, kuva, suodatin, muunnin tai muunlainen datan käsittelijä sekä piirre, piirreirrotin, luokka ja luokitin. Näistä kuva ja data, sekä myös piirre ja luokka kuvaavat numeerista informaatiota ja voitaisiin toteuttaa yhdellä luokalla tai tästä periyttämällä. Päädyimme kuitenkin toteuttamaan yleisen dataluokan ja sisällyttämään sen aggregaattina kuva-, piirre-, ja luokkaluokkaan. Erilaisia kuvan ja datan käsittelijöitä voidaan kuvata niin ikään yhdellä yläluokalla. Ylätason luokkarakenne on esitetty oheisessa UML-kaaviossa (kuva 6).



Kuva 6: Sovelluskehyksen oliorakenne UML-kaaviona.

Luokkien välinen vuorovaikutus on varsin suoraviivaista arkkitehtuurin mukaisesti. Kamera tuottaa kuvadataa, jota kuvankäsittelyoperaatiot käyttävät tuottaen muokatun kuvan. Piirreirrotinkin käsittelee vielä kuvia, mutta tuottaa kompaktimpaa piirreinformaatiota, jota taas luokitin käyttää määrittäen sen, mihin luokkaan data piirteen perusteella kuuluu. Datan välitys sensori-, datankäsittely-, piirreirrotin- ja luokitinluokissa on toteutettu niin, että luokan instanssille kerrotaan, mikä oliot sisältää sen lähde- ja mikä kohdedatan.

Näin vältytään kopioimasta samaa dataa useamman kerran, kuten funktion parametreja ja paluuarvoja käyttävä toteutus tekisi. Samalla liukuhihna-ajatus toimii sujuvasti, sillä kameralta tuleva uusi kuva välittyy automaattisesti eteenpäin kuvankäsittelijälle ja niin edelleen.

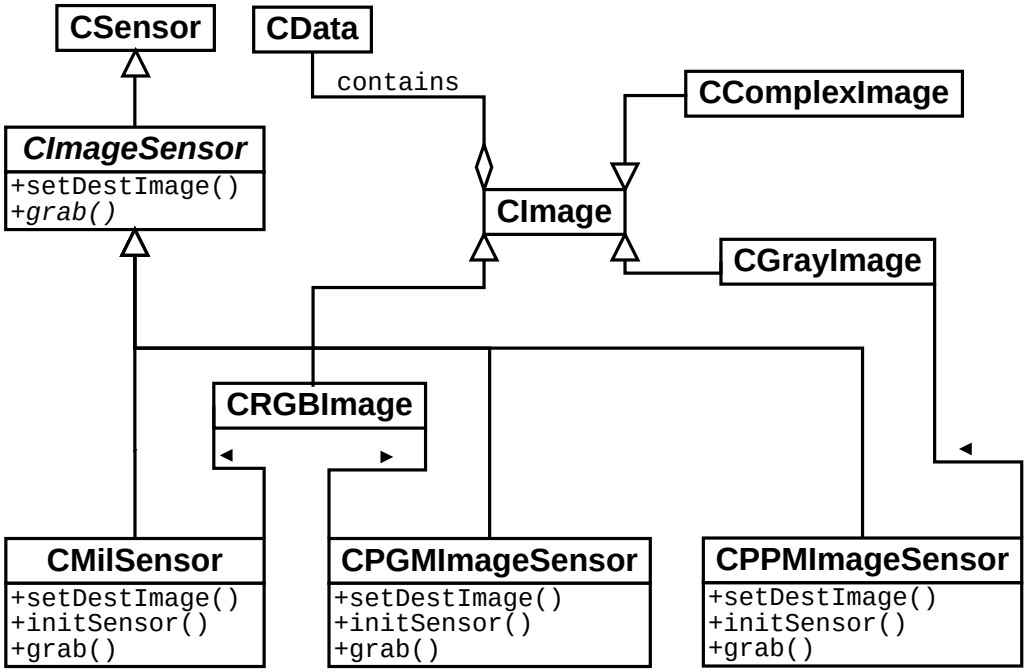
Lähde- ja kohdedatan asetuksen lisäksi edellä luetellut toimintoluokat tarvitsevat toiminnon, joka saa aikaa lähdekuvan lukemisen ja kohdekuvan tuottamisen, eli tee kuvankaappaus, suodata kuva, irrota piirteet ja luokita. Dataluokissa tarvittavat toiminnot ovat taas datan asetus ja lukeminen sekä mahdollisten lisäominaisuuksien asettaminen ja lukeminen. Esimerkiksi kuvaluokassa muita datajäseniä ovat kuvan tyyppi ja koko, jotka pitää pystyä sekä asettamaan ja lukemaan. Kapselointiperiaatteen mukaan noita arvoja ei tulisi voida muuttaa suoraan luokan ulkopuolelta, vaan tähän pitää olla erilliset rajapintametodit.

4.2 Toteutus

Ohjelmointikieleksi valittiin C++, koska Windows-toteutuksessa käytetty Matroxin kuvankäsittelykirjasto tuki vain C- ja C++-kieliä. Lisäksi muita C++-kielisiä ohjelmointikirjastoja on paljon tarjolla. Java olisi voinut olla toinen varteenotettava vaihtoehto graafisten kirjastofunktioidensa ja siirrettävyytensä takia.

Uudelleenkäytettävyyttä parannettiin templaateilla ylätason luokissa, kuten Cheung ja Ip omassa CBIRFrame-sovelluksessaan [5]. Näin luokan käsittelemä tietorakenne on helppo korvata. Esimerkiksi data voi olla kokonaislukuja, kuten usein kuvissa, tai liukulukuja, jos mitataan vaikkapa lämpötiloja. Templaattiluokkien käyttö on joustavaa, mutta ne jättävät tietorakenteiden valinnan käyttäjän vastuulle. Toteuttamissamme alemman tason luokissa tietorakenteet on valittu valmiiksi käyttötarkoituksen mukaisesti, joten sovelluskehyksen käyttäjän ei tarvitse välttämättä pohtia tarkoituksenmukaisia tietorakenteita.

Ylätason luokat eivät vielä tee mitään, ne tarjoavat vasta yhtenäisen pohjan ja muutamat perusoperaatiot samantyyppisille muille luokille. Oleellinen osa työtä tämän ohjelmistokehyksen lisäksi oli ohjelmoida joukko perusoperaatioita. Sensorinluokasta periytettiin kuvasensori ja tästä operaatiot kahden erityyppisen kuvan lukemiseksi tiedostosta sekä kuvankaappaus digitointikortilta. Näiden alaluokkien rakenne on esitetty UML-kaaviossa kuvassa 7. Kuvia käsitellään kaksiulotteisina matriiseina, joissa yksi arvo viittaa tiettyyn kuvapisteeseen. Värit esitetään tarvittaessa käyttämällä kolmatta ulottuvuutta.



Kuva 7: Sensoriluokasta periytetyt aliluokat datajäsenineen.

Konversiot eri värijärjestelmien välillä sekä harmaasävykuvaksi muuntaminen ovat perusmuunnoksia, jotka toteutettiin. Vaativampia operaattoreita varten, lähinnä Gabor-muunninta silmällä pitäen, toteutettiin myös Fourier-muunnin. Lisäksi toteutettiin Display-luokka, jotta kuvia voi katsoa tarvittaessa näyttöpäätteellä eri operaatioiden jälkeen. Piirreirrottimista toteutettiin histogrammin laskeminen ja luokittimista yleiset Bayesin ja 1-NN-luokittimet.

5 JOHTOPÄÄTÖKSET

Toteuttamamme sovelluskehys osoittautui varsin toimivaksi, mutta lisää erilaisia valmiita kuvankäsittely-, piirreirrotus- ja hahmontunnistuskomponentteja tarvitaan. Optimaalisen rajapinnat määrittäminen osoittautui haastavaksi, sillä toteutusta muutettiin useaan kertaan. Järjestelmää ei juurikaan ehditty dokumentoida, joten operaatioita lisättäessä käyttöohjeisiin on myös syytä panostaa. Mitä monimutkaisempi sovelluskehys on, sitä tärkeämpää on dokumentoida kehyksen tarkoitus, käyttö ja rakenne [10].

Jopa C++-kielellä toteutettu sovellus voi olla liian hidas, jos käsiteltävää dataa on todella paljon. Valitun kielen hyvänä puolena on kuitenkin se, että pullonkauloiksi osoittautuvia operaatioita on melko helppo koodata C-kielellä [4], sillä C on C++-kielen alijoukko.

LÄHDELUETTELO

[1] S. S. Alhir. The object-oriented paradigm. [Verkkodokumentti], lokakuu 1998. [Viitattu: 20.4.2010]. Saatavissa: http://www.cs.utep.edu/sroach/S10-5380/TheObjectOrientedParadigm.PDF.

[2] X. Amatriain, P. Arumi ja D. Garcia. A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, 14(1):15–32, kesäkuu 2008.

[3] G. Booch. *Object-Oriented Design with applications*. Benjamin Cummings, 1991.

[4] J. Bowskill, T. Katz ja D. Cattez. An object oriented approach to a machine vision framework. *IEE Colloquium on Design, Implementation and Use of Object-Oriented Systems*, sivut 6/1 – 6/3, Lontoo, Yhdistyneet kuningaskunnat, tammikuu 1994.

[5] K. K. Cheung ja H. H. Ip. Developing an object-oriented framework for content-based image retrieval. *Software-Practice and Experience*, 33(6):523–565, toukokuu 2003.

[6] P. Coad ja E. Yourdon. *Object-oriented analysis*. Prentice-Hall, 1991.

[7] R. Duda ja P. Hart. Use of the Hough transform to detect lines and curves in pictures. *Communication of the Association for Computing Machinery*, 15(1):11–15, tammikuu 1972.

[8] M. E. Fayad. Introduction to the computing surveys' electronic symposium on object-oriented application frameworks. *ACM Computing Surveys*, 32(1):1–9, maaliskuu 2000.

[9] M. E. Fayad ja D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, lokakuu 1997.

[10] G. Froehlich, H. J. Hoover, L. Liu ja P. Sorenson. Hooking into object-oriented application frameworks. *Proceedings of 19th International Conference on Software Engineering*, sivut 495–505, Boston, Yhdysvallat, toukokuu 1997.

[11] R. C. Gonzalez ja R. E. Woods. *Digital Image Processing*. Prentice-Hall, 2. painos, 2002.

[12] I. Jacobson, M. Christerson, P. Jonsson ja G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[13] R. C. Jain, R. Kasturi ja B. G. Schunk. *Machine Vision*. McGraw-Hill, 1995.

[14] K. Jyrkinen. Optinen kolmiulotteinen mittaus ohutlevytuotteiden laadunvalvonnassa. Diplomityö, Lappeenrannan teknillinen yliopisto, 2007.

[15] K. Koskimies. *Oliokirja*. Suomen ATK-kustannus, 2000.

[16] Mathworks. Matlab - the language of technical computing. [Verkkodokumentti]. [Viitattu: 21.4.2010]. Saatavissa: http://www.mathworks.com/products/matlab/.

[17] Matrox Imaging. Machine vision and imaging software - matrox imaging library. [Verkkodokumentti]. [Viitattu: 24.4.2010]. Saatavissa: http://www.matrox.com/imaging/en/products/software/mil/.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy ja W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991.

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker ja J. Dongarra. *MPI: The complete reference.*, osa 1. MIT Press, Cambridge, MA, 2000.

[20] S. Theodoridis ja K. Koutroumbas. *Pattern recognition*. Elsevier Academic Press, 2. painos, 2003.

[21] R. Vivanco, A. B. Demko, M. Jarmasz, R. L. Somorjai ja N. J. Pizzi. A pattern recognition application framework for biomedical datasets. *IEEE Engineering in Medicine and Biology Magazine*, 26(2):82–85, maaliskuu 2007.

LIITE 1: Esimerkki ja näyte lähdekoodeista

Matroxin kuvankäsittelykirjastoa käyttävä esimerkkiohjelma, joka testaa osaa toteutetun sovelluskehyksen piirteistä.

```
#define WINDOWS
#include <iostream.h>
#include "GrayImage.hxx"
#include "MilSystem.hxx"
#include "MilImageSensor.hxx"
#include "RGB2Gray.hxx"
#include "MilDisplay.hxx"
#include "ComplexImage.hxx"
#include "GaborTransform.hxx"

using namespace std;

int main ()
{
  CMilImage image;
  CGrayImage <CMilImage::Imagetype> oneBandImage;
  CMilSystem systembolaget;
  CMilDisplay display(systembolaget), fftdisplay(systembolaget);
  CFeature feature;
  CClass result;

  CMilImageSensor sensori(systembolaget);
  sensori.setDestImage(image);
  sensori.initSensor(512,512);
  sensori.grab();
  display.setSourceImage(image);
  display.Display();

  CRGB2Gray<CMilImage::Imagetype> olio;
  olio.initProcessor(olio.AVERAGE);
  olio.setSourceImage(image);
  olio.setDestImage(oneBandImage);
  olio.process();
```

(jatkuu)

```
  CHistogramFeatureExtractor featureExt;
  featureExt.setSourceImage(image);
  featureExt.setDestFeature(feature);

  C1NNClassifier classifier;
  classifier.setSourceFeature(feature);
  classifier.setDestClass(result);

  while(1)
  {
    sensori.grab();
    featureExt.Extract();
    classifier.Classify();
    cout<<result;
  }
  return 0;
}
```

Esimerkkinä itse luokista seuraavassa yleisen sensoriluokan ja siitä periytetyn Matroxin kuvankäsittelykirjastoa käyttävän luokan toteutus.

```
#include "Image.hxx"
#include "Sensor.hxx"

class CImageSensor: public CSensor
{
public:
  template <class T> CImage<T> getImage()=0;
  template <class T> void setImage(CImage<T>&)=0;
  template <class T> void setDestImage(CImage<T>&);
  virtual void grab()=0;
};
```

```
#include <mil.h>
#include <iostream.h>
#include "RGBImage.hxx"
#include "ImageSensor.hxx"
```

(jatkuu)

```
typedef CRGBImage<unsigned char> CMilImage; // for clarity

class CMilImageSensor: public CImageSensor
{
public:
  CMilImageSensor(CMilSystem&);
  ~CMilImageSensor();
  virtual void setDestImage(CMilImage& destImage_);
  virtual void initSensor(const unsigned long width_=0,
                          const unsigned long height_=0);
  virtual void grab();
private:
  MIL_ID m_application, m_system, m_digitizer, m_imageRGB;
  unsigned long m_imageWidth, m_imageHeight, m_maxWidth, m_maxHeight;
  CMilImage *m_image;
  CMilSystem *m_MilSystem;
  void MilAllocationCheck();
};

//*************************************************************
// Constructor to allocate system resources for MIL - allocation
// errors are checked. Mil system and application have to be
// reserved using instance of CMilSystem class

CMilImageSensor::CMilImageSensor(CMilSystem& MilSystem_)
{
  // Store MilSystem for possible future use
  m_MilSystem = &MilSystem_;

  // Initializing system variables to zero, which should mean
  // that that part of system is not allocated. Allocated MIL_ID
  // is assumed to be !0.
  m_digitizer = 0;
  m_imageRGB = 0;
  m_image = 0; // Not allocated yet, setting to 0

  // Getting application and system IDs from CMilSystem object
  // so that instances of other classes can use Mil, too
  m_application = MilSystem_.getApplicationId();
  m_system = MilSystem_.getSystemId();
```

(jatkuu)

```
  MdigAlloc(m_system, M_DEV0, "DXC9100P.dcf", M_DEFAULT, &m_digitizer);
  MilAllocationCheck();

  // Max image width of the camera
  m_maxWidth = MdigInquire(m_digitizer, M_SIZE_X, M_NULL);
  m_imageWidth = m_maxWidth;
  // Max image height of the camera
  m_maxHeight = MdigInquire(m_digitizer, M_SIZE_Y, M_NULL);
  m_imageHeight = m_maxHeight;

  // Allocating image buffer with maximum dimensions
  m_imageRGB = MbufAllocColor(m_system, 3, m_imageWidth,
m_imageHeight, 8+M_UNSIGNED, M_IMAGE+M_GRAB+M_PROC, M_NULL);
  MilAllocationCheck();
}

// Destuctor frees the resources
CMilImageSensor::~CMilImageSensor()
{
  MbufFree(m_imageRGB);
  MdigFree(m_digitizer);
}

// Setting destination image
void CMilImageSensor::setDestImage(CMilImage& destImage_)
{
  m_image=&destImage_;
}

// Changes image size, if values are valid
void CMilImageSensor::initSensor(const unsigned long width_,
                                 const unsigned long height_)
{
  if(width_>0 && height_>0 && width_ <= m_maxWidth
     && height_ <= m_maxHeight
     && ( width_ != m_imageWidth || height_ != m_imageHeight ))
  {
    m_imageWidth = width_;
    m_imageHeight = height_;
```

(jatkuu)

```
    MbufFree(m_imageRGB);
    m_imageRGB = MbufAllocColor(m_system,3,m_imageWidth,
m_imageHeight,8+M_UNSIGNED,M_IMAGE+M_GRAB+M_PROC,M_NULL);
    MilAllocationCheck();
  }
}

// A method which grabs an image into the image object
void CMilImageSensor::grab()
{
  if(m_image == 0) // Testing, if destination image is set
  {
    cerr << "Destination image not set using setDestImage!" << endl;
    MbufFree(m_imageRGB);
    MdigFree(m_digitizer);
    MsysFree(m_system);
    MappFree(m_application);
    throw "Allocation error";
  }

  MdigGrab(m_digitizer, m_imageRGB); // grab and check if worked
  MilAllocationCheck();
  // Setting image size - space is allocated at the same time
  m_image->setSize(m_imageWidth, m_imageHeight);
  // Copying RGB color levels to the image
  MbufGetColor(m_imageRGB,M_RGB24+M_PACKED,M_ALL_BAND,
               m_image->getData().getDataPtr());
}

void CMilImageSensor::MilAllocationCheck()
{
  char error[M_ERROR_MESSAGE_SIZE];
  if(MappGetError(M_CURRENT+M_MESSAGE, error)!=M_NULL_ERROR)
  {
    cerr << error << endl;
    if(m_imageRGB) MbufFree(m_imageRGB);
    if(m_digitizer) MdigFree(m_digitizer);
    throw "MIL allocation error";
  }
}
```