TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OLLI USENIUS

KUUSENTAIMEN AUTOMAATTINEN TUNNISTUS KAMERASOVELLUKSENA

Diplomityö

Tarkastajat: Prof. Ari Visa, Prof. Kari Koskinen, Prof. Jukka Vanhala
Tarkastajat ja aihe hyväksytty
tieto- ja sähkötekniikan tiedekunnan
dekaanin päätöksellä 1.4.2017

TIIVISTELMÄ

OLLI USENIUS: Kuusentaimen automaattinen tunnistus kamerasovelluksena
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua, 10 liitesivua
toukokuu 2017
sähkötekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Ari Visa, Prof. Kari Koskinen, Prof. Jukka Vanhala
Avainsanat: Koneellinen taimikonhoito, k-Lähin naapuri, konenäkö

Yksi nuorenmetsänhoidon koneellistamisen haasteista on koneiden työtehokkuus ja tuntikustannus verrattuna metsurin tekemään raivaussahatyöhön. Koneiden työtehon parantamiseksi vastuuta työskentelystä voisi tulevaisuudessa siirtää kuljettajalta koneelle lisäämällä automaatiota. Yksi vaihtoehto automaation lisäämiseksi on koneen ympäristönhavannointikyvyn kasvattaminen, jolloin kone pystyisi joko avustamaan kuljettajaa toimenpiteissä tai jopa tekemään yksinkertaisia toimenpiteitä autonomisesti.

Tämän diplomityön tavoitteena on kehittää käytännönläheinen kuusentaimentunnistusalgoritmi, jonka pystyisi liittämään osaksi metsäkoneen ohjausjärjestelmää. Työ suoritettiin Tampereen teknillisellä yliopistolla vuosina 2016-2017. Algoritmin kehittäminen aloitettiin tutustuen aihepiirin aikaisempiin julkaisuihin ja pohtien niissä käytettyjen ratkaisujen soveltuvuutta käytännössä. Tämän jälkeen kehitettiin testiohjelmisto, jota apuna käyttäen kirjoitettiin laskennallisesti yksinkertainen algoritmi käyttäen C++-ohjelmointikieltä, sekä OpenCV-konenäkökirjastoa. Työkalut valittiin niin, että algoritmi olisi mahdollista toteuttaa sulautettuna järjestelmänä.

Vaikka tämä työ rajoittui vain algoritmin kehittämiseen, pohdittiin lopuksi sulautetun järjestelmän toteutuksen vaatimuksia mekaanisen kestävyyden, kustannustason ja toimintavarmuuden osalta. Testilaitteistoa ei rakennettu, mutta algoritmia testattiin koeaineistolla onnistuneesti.

ABSTRACT

OLLI USENIUS: Automated recognition of a spruce plant as a camera application
Tampere University of Technology
Master of Science thesis, 49 pages, 1 Appendix pages
May 2017
Master's Degree Programme in Electronics
Major: Embedded Systems Examiners: Prof. Ari Visa, Prof. Kari Koskinen, Prof. Jukka Vanhala
Keywords: Young forest management, embedded systems, k-Nearest Neightbour, machine vision

One of the biggest challenges in mechanized young forest management is the working speed and hourly costs compared to manual work. To improve the working speed of such machines, responsibilities could be moved from the operator to the machine. One possibility could be to improve the machine's awareness by optical sensors in order to assist the operator in simple tasks or even by executing those tasks without the operator's decision.

The purpose of this thesis is to develop a spruce plant detection algorithm which could be implemented into a part of a real forestry machine's control system. The thesis was made during 2016-2017 at Tampere University of Technology. The research was started by reading previous publications regarding the topic and by analysing the usability of those solution in a real life. After this a development software was made to be used as a platform for testing the algorithm made in C++ and by using OpenCV computer vision library. The tools were chosen while keeping in mind the easiness to implement the system as an embedded system.

Although this thesis is limited only in the algorithm development, different requirements regarding the embedded system was also analyzed, for example what kind of physical, electrical and computational requirements are in order to implement a system usable in real life machines.

ALKUSANAT

Tämä diplomityö on tehty Usewood Forest Tec Oy:n, Tampereen teknillisen yliopiston kone- ja tuotantotekniikan laitoksen, signaalinkäsittelylaitoksen ja elektroniikan laitoksen välisenä yhteistyönä. Työ oli kolmivuotinen prosessi, jonka aikana koneellinen taimikonhoito otti suuria harppauksia eteenpäin.

Lämpimät kiitokset työn ohjaajina toimineille professori Ari Visalle ja professori Kari Koskiselle asiantuntevasta ja avuliaasta työn ohjauksesta. Sain työn aikana kaiken tarvitsemani avun aina kun sitä tarvitsin. Kiitokset myös tekniikan tohtori Jussi Aaltoselle auttamisesta työn eri vaiheissa ja lääketieteen tohtori Jussi-Pekka Useniukselle kaikesta koneellisen taimikonhoidon parissa tehdystä työstä, joka mahdollisti tämän työn toteuttamisen.

Lisäksi haluaisin kiittää tulevaa vaimoani Leenaa kaikesta tuesta minkä minulle työn aikana annoit. Haluan myös pyytää jo etukäteen anteeksi sitä, ettet päässytkään työn viivästymisen takia naimisiin diplomi-insinöörin, vaan virallisesti vielä teekkarin, kanssa.

Tampere, 6.5.2017

Olli Usenius

SISÄLLYS

LYHENTEET JA MERKINNÄT

vesakko	Itsestään kylväytynyt nuori puusto
TTY	Tampereen teknillinen yliopisto
k-NN	k-nearest neighbour
RGB	Reg-Green-Blue värimalli
HSV	Hue-Saturation-Value värimalli

1. JOHDANTO

Tavaralajimenetelmään perustuva metsäteollisuus, mitä Pohjoismaissa on harjoitettu ja kehitetty viimeisten vuosikymmenten ajan, perustuu metsästä saatavan puun mahdollisimman tehokkaaseen hyötykäyttöön. Metsänhoidon tavoitteena on saada mahdollisimman suuri kasvupotentiaali metsästä, keräten puuaines siten, että mahdollisimman vähän siitä menisi hukkaan. Tavaralajimenetelmässä puu prosessoidaan metsässä suoraan määrämittaisiksi tukeiksi, jotka kuljetetaan myöhemmin sieltä pois. Tällöin puun käsittelyssä poistettu puuaines, erityisesti puun oksat, jäävät metsään ravinteiksi seuraavalle puusukupolvelle. Tavaralajimenetelmällä metsämaan taloudellinen arvo voidaan kasvattaa moninkertaiseksi hoitamattomaan metsään verrattuna, mutta sen haastavuus on korjuumenetelmien ja metsänhoidon monimutkaisuus. Tehokkaasti kasvava metsä vaatii oikeanaikaista hoitoa, etenkin kun puut ovat vielä taimia.

Viime vuosien aikana metsureiden määrä on vähentynyt ja manuaalisen työn rinnalle on alettu kehittämään koneellisia ratkaisuja, joilla nuorenmetsänhoitoa saataisiin tehostettua. Koska metsämaasto on epätasaista ja puiden taimet ovat pieniä ja ne kasvavat tiheässä, on koneellisen taimikonhoidon ongelmana sen tehokkuus. Kone työskentelee lähellä toisiaan olevien taimien välissä, tavoitteena poistaa mahdollisimman läheltä tainta risut raapaisemattakaan itse tainta. Koneen kuljettajan havannointikyky on rajallinen ja koko työpäivän jatkuvaa keskittymistä vaativa työskentely on henkisesti kuormittavaa.

Vuonna 2013 Hyyti, Kalmari ja Visala kehittivät Aalto-yliopistolla kuusentaimia ylhäältäpäin tunnistavan algoritmin, sekä tuota algoritmiä käyttävän prototyyppikoneen, joka pystyi ilman kuljettajaa tunnistamaan 80 % heinittyneen niityn istutetuista kuusentaimista [7]. Tutkimus osoittaa kuusentaimen automaattisen tunnistamisen olevan mahdollista, mikä puolestaan vähentäisi kuljettajan tarvetta keskittyä työskentelyyn jatkuvasti. Automaatio voisi olla keino lisätä koneellisen taimikonhoidon koneiden työtehokkuutta, mutta jotta automaatiota hyödyntäviä koneita tulisi

markkinoille, tulee taimentunnistussovellus saada valmistettua siten, että se toimii osana koneen ohjausjärjestelmää kuljettajan apuna. Tätä varten pöytätietokoneen asentaminen osaksi työkonetta on epäkäytännöllistä ja toiminnaltaan epävarmaa, sillä perinteiset tietokoneet eivät ole suunniteltuja toimivaan likaisessa ja tärisevässä ympäristössä. Toimiva järjestelmä vaatisikin sulautetun järjestelmän suunnittelua, joka pystyy tunnistamaan kuusentaimen reaaliaikaisesti sulautetun järjestelmän rajoitetulla laskentateholla.

2. KONEELLINEN TAIMIKONHOITO

Koneellinen taimikonhoito tarkoittaa nuoren metsän kunnostusta, raivausta tai perkausta koneellisesti. Perinteinen tapa hoitaa taimikoita on jo pitkään ollut manuaalinen raivaus joko raivaussahalla tai moottorisahalla. Manuaalinen raivaus on kuitenkin fyysisesti rasittavaa, eikä metsureita ole enää riittävästi hoitamaan kaikkia pohjoismaisia metsiä. Koneellisessa taimikonhoidossa pyritään vähentämään miestyötä ja pienentämään taimikonhoidosta koituvia kuluja korvaten manuaalinen raivaus pienmetsäkoneella tai metsäkoneella tehdyllä työllä.

2.1 Taimikonhoidon tarkoitus

Nuorta metsää hoidetaan ja hoitoa kehitetään pohjoismaissa, erityisesti Suomessa ja Ruotsissa. Pohjoismainen metsänhoito perustuu metsästä saatavan puuaineksen mahdollisimman tehokkaaseen käyttöön ja metsän kasvun optimoimiseen. Metsän kasvun kannalta taimikonhoito on tärkeässä roolissa, sillä suurin osa metsän kasvatukseen käytettävästä työstä tehdään nimenomaan nuoressa metsässä. Taimikonhoidon tarkoitus on edistää haluttujen puuntaimien kasvua ehkäisemällä ei-haluttujen taimien liiallinen kasvu. Pohjoismaissa etenkin hieskoivu, haapa ja leppä ovat luontaisesti aukoille kasvavia puulajeja, kun taas puuteollisuuden kannalta arvokaampia lajeja ovat yleisimmät havupuumme kuusi ja mänty, sekä rauduskoivu.

Erityisesti kuusimetsän kasvattaminen vaatii ihmisen apua, sillä kuusi on kasvunopeudeltaan hitaampi kuin muut puulajit. Luontaisesti kuuset kasvavat muiden puiden alla varjossa ja kuusimetsä syntyy luontaisesti vasta sitten, kun muut puulajit poistuvat kuusien päältä. Vaikka kuusi runsashavuisena puuna selviää hengissä varjossa, on sen kasvu nopeampaa jos se saa kasvaa valtapuuna. Tämän takia kuusitamikko on raivattava silloin kun muut puulajit ovat ylittämässä kuusentaimien keskipituuden, jonka jälkeen ne alkaisivat hidastamaan kuusentaimien kasvua.

Oikein ajoitetulla taimikonhoidolla voidaan lisätä metsästä saatavaa tuloa ja pie-

nentää kasvun kiertoaikaa huomattavasti. Metsän kiertoajalla tarkoitetaan sitä aikaa, mikä menee puiden istutuksesta päätehakkuuseen. Hoidettu taimikko kasvaa keskimäärin 20-35 % nopeammin kuin hoitamaton taimikko. Myös puuaines on jatkojalostukseen sopivampaa, sillä esimerkiksi hirvivahingot vähenevät. [16]

2.2 Koneellisen taimikonhoidon menetelmät

Nuoren metsän koneellinen hoito on vielä vakiintumatonta ja sen menetelmät kehittyvät jatkuvasti. Arviolta 1-2 % Suomen taimikoista hoidetaan koneellisesti ja loput joko manuaalisesti, tai ei ollenkaan[15]. Tällä hetkellä yleisimmät menetelmät koneelliseen taimikonhoitoon ovat kitkentä ja raivaus. Ne eivät ole toisiaan pois sulkevia menetelmiä, vaan molemmilla on niihin parhaiten soveltuvat kohteet, joissa menetelmät ovat soveltuvimpia kustannustehokkuudeltaan, sekä työjäljeltään.

Koneellisen raivauksen perusperiaate on sama kuin raivaussahaamisessa. Tarkoitus on katkaista ja kaataa ei-toivotut itsestään kylväytyneet puut, eli vesakko, istutettujen taimien ympäriltä, antaen taimille valoa kasvamiseen. Raivauskoneiden koot vaihtelevat paljon eri koneiden välillä. Pääsääntöisesti pienemmillä koneilla on parempi työnlaatu ja matalemmat käyttökustannukset, mutta työskentely on myös hitaampaa kuin suuremmilla koneilla. Kuvassa 2.1 esitetty pienmetsäkone on Usewood Forest Tec Oy:n valmistama taimikonraivauskone. Se on yksi pienimmistä markkinoilta löytyvistä metsäkoneista, joka kokonsa puolesta mahtuu kulkemaan taimien välissä. Työlaitteena koneessa on UW50 risuraivain, joka on kuin suurikokoinen hydraulikäyttöinen raivaussaha. Hydraulimoottori pyörittää 80 cm halkaisijaltaan olevaa terää, joka puolestaan katkaisee risut [17].

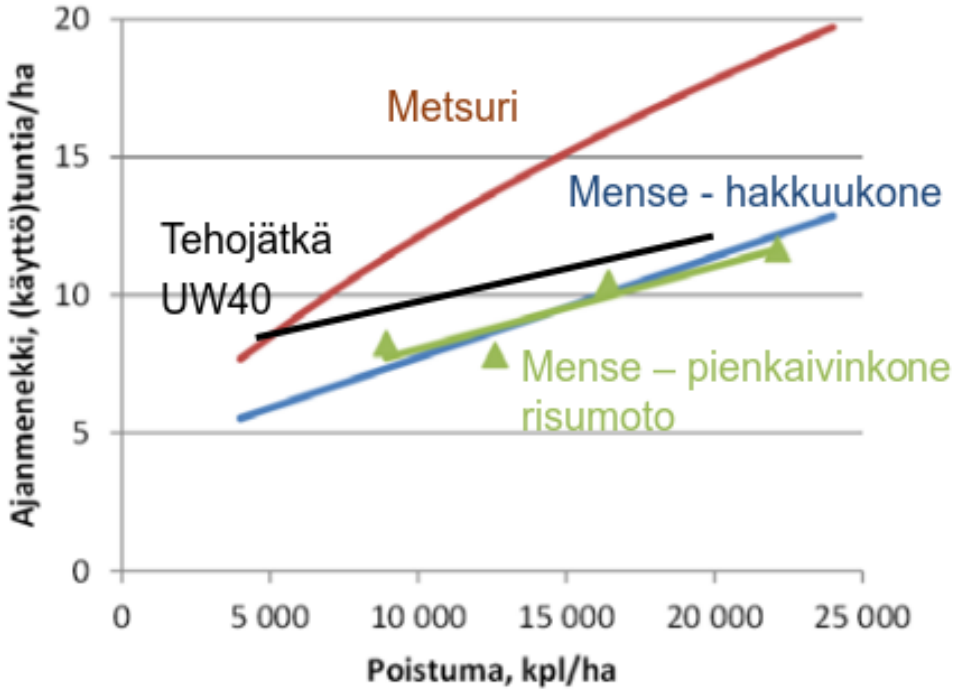
Koska perinteisessä raivauksessa vesakko katkaistaan läheltä maanpintaa säästäen niiden juuret vahingoittumattomina, kasvaa vesakko melko nopeasti takaisin. Varhaisperkauksen jälkeen vesakko saavuttaa muutamassa vuodessa valtapuuston pituuden ja usein taimikko pitää raivata toistamiseen ennenkuin valtapuusto on tarpeeksi suurta tukahduttaakseen vesakon takaisinkasvun. Toisen raivauskerran välttämiseksi käytetään raivauksen tehon lisäämiseksi soveltuvissa paikoin biologisia ja kemiallisia vesakontorjunta-aineita, kuten glyfosaattia ja purppuranahakkasientä [15].

Koneellisen raivauksen edut tulevat parhaiten näkyviin kun raivattava taimikko on haastavaa ja poistuma, eli poistettavia puiden lukumäärä hehtaaria kohden, on suuri. Raivauskoneiden ajankäyttö hehtaaria kohden kasvaa hitaammin kuin metsurin,

Kuva 2.1 *Usewood Forest Master pienmetsäkone ja UW50 risuraivain [17]*

kuten kuvasta 2.2 nähdään. Koneellinen raivaus näin ollen sopii hyvin niihin leimikoihin mitä ei ole hoidettu ajallaan ja missä miestyö on kallista.

Kitkettäessä vesakko poistetaan kokonaisuudessaan niin, että myös niiden juuret vedetään maasta. Tällöin vesakon uudelleenkasvu on huomattavasti hitaampaa, kuin pelkällä raivauksella, ja taimikonhoidossa voidaan selvitä yhdellä hoitokerralla. Kitkentään soveltuu kuiva, kivennäisperäinen maasto. Soistuneet kasvupaikat ja liian suureksi kasvanut vesakko aiheuttaa kitkennälle ongelmia, sillä kitkin saattaa repiä vesakon ohella maasta multaa ja taimia [15]. Kuvassa 2.3 on kitkentään soveltuva metsäkone, sekä kitkintyölaite. Kitkiä lasketaan poistettavan vesakon yläpuolelta, jonka jälkeen hydraulisesti liikutettava ristikko ottaa puista kiinni. Tämän jälkeen runkokone vetää puut juurineen pois maasta.



Kuva 2.2 *Koneellisen raivauksen tuntityökustannus verrattuna metsuriin eri työkoneilla [15]*

Kuva 2.3 *Kitkentätyölaite metsäkoneessa*

3. KONEELLISEN TAIMIKONHOIDON HAASTEET JA ONGELMAT

Molemmissa koneellisen taimikonhoidon työmuodoissa suurimpana haasteena voidaan pitää työtehoa. Raivaussahaan verrattuna koneet ovat hinnaltaan moninkertaisia ja koneurakoijan on saatava kuolletettua koneen hankinnasta ja käytöstä muodostuvat kulut. Koneilla ei voi ajaa ympäri vuoden, sillä talvisaikana sekä koneellinen raivaus että kitkentä ovat haastavaa jäisen maan ja lumipeitteen takia. Epätasainen maasto vaikeuttaa ja hidastaa koneilla ajoa. Oikeanlaisten työkohteiden valitseminen ja hinnoittelu voi olla urakoitsijalle haasteellista. Kitkennän kustannustehokkuus on riittävä jos voidaan taata ettei uudisvesomista tapahdu, tai että se on vähäistä, jolloin myöhemmin tehtävä raivaus onnistuu miestyönä edullisesti. Näiden oletusten paikkaansapitävyys puolestaan on riippuvainen leimikosta ja sen maatyypistä.

Kitkentä vaatii käytettävältä työkoneelta yleensä enemmän tehoa kuin raivaus, sillä koneen on jaksettava nostaa puut juurineen maasta. Siksi kitkentäkoneet on yleensä suurempikokoisia kuin raivauskoneet. Kitkentätyöhön kuluva aika hehtaarille kasvaa lineaarisesti vesakon keskipituuden ja tiheyden kasvaessa [15]. Kitkentä kannattaa siis tehdä ajallaan, kun vesakko ei ole vielä päässyt kasvamaan liian pitkäksi. Sen hehtaarikustannus on korkeampi kuin koneellisen raivauksen tai metsurin, mutta kokonaisuudessaan työ tulee edullisemmaksi kohteissa joissa kitkentä estää uudisvesomisen [15].

Kuva 3.1 vertaa koneellisen raivauksen kustannuksia eri koneille raivaussahaan verrattuna. Siitä voidaan todeta koneellisen raivauksen olevan joko samanhintaista tai kalliimpaa kuin raivaussahatyön, riippuen käytetystä koneesta. Koneellisen raivaustyön laatua on vaikea verrata käsin tehtyyn, mutta metsänhoidollisesta näkökulmasta se on käytännössä sama, vaikkei kuitenkaan yhtä siistä ja silmää miellyttävää. Tämä tarkoittaa sitä, että työtavasta riippumatta saadaan raivattua taimille kasvutilaa ja vesakon kasvu laantuu yhtä tehokkaasti. Kuitenkaan metsänomistajat

ja marjastajat eivät välttämättä tykkää koneellisen työn jäljestä, sillä metsään jää usein katkenneita risuja pystyyn, sekä kannot ovat korkeampia kuin miestyönä tehtynä. Kuten kuvasta 3.1 näkee, on koneellisen raivauksen haasteena olla miestyötä tehokkaampi. Jollei koneellinen raivaus pysty tarjomaan mitään lisäarvoa miestyöhön verrattuna, on se toissijainen ratkaisu johon päädytään vain jos metsureita ei ole saatavilla tai manuaalinen työ todetaan muuten mahdottomaksi.

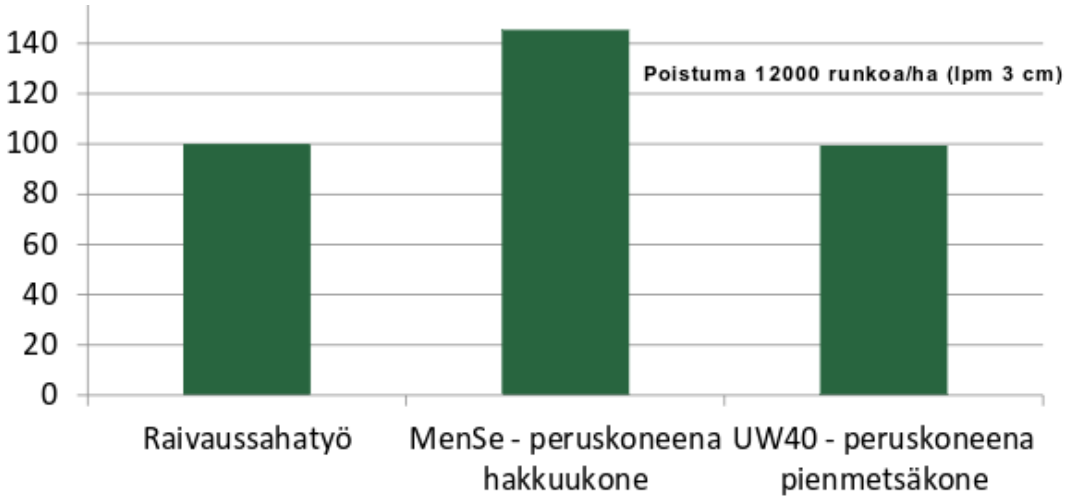
3.1 Tutkimushypoteesi ja -kysymykset

Pienmetsäkonesimulaattorilla tehdyn tutkimuksen perusteella kuljettajien tehokkuus kasvoi merkittävästi kun ajoa helpoitettiin automaatiolla. Kun työlaite lähestyessään säästettävää tainta joko hidasti liikkeitään, tai pyrki aktiivisesti välttämään osumista taimeen, kasvoi kuljettajien tehokkuus samalla kun vauriot jäljelle jäävään puustoon vähenivät. [10] Yksi tapa lisätä koneellisen raivauksen tehoa olisi siis säästettävien taimien havannointi ja kuljettajan avustaminen tämän tiedon perusteella, jolloin kuljettajan henkinen työtaakka ja vaikutus työtehoon vähenisi. Henkisesti helpon työn tekeminen on tehokkaampaa jo pelkästään siksi, että kuljettaja pystyy tekemään pidempää työpäivää. Lisäksi ihminen jaksaa tehdä henkisesti helppoa ja mukavaa työtä huomattavasti pidempään kuin henkisesti raskasta työtä ilman loppuun palamista.

Konenäkömenetelmien käyttämistä kuljettajan työskentelyn avustamiseksi on tutkittu kitkevillä taimikonhoitokoneilla ja sen on todettu toimivaksi [7]. Kitkettäessä taimia lähestytään ylhäältä päin rauhallisesti, laskien työlaite joko taimen päälle tai sen viereen. Koneellisessa raivauksessa taimia lähestytään sivulta mahdollisesti hyvinkin nopeasti, ja pienikin kosketus taimeen voi pilata sen. Näiden peruslähtökohtien eroavaisuuksien takia samaa tunnistusmenetelmää ei voida suoraan soveltaa, vaan keskeinen kysymys konenäön tuomisessa koneelliseen raivaukseen on sellaisen algoritmin kehittäminen, joka pystyy tunnistamaan kuusentaimen sivulta tai viistosta reaaliaikaisesti riittävän lyhyessä ajassa, jotta kone pystyy suorittamaan kuljettaa avustavia toimenpiteitä.

Tutkimushypoteesi:

- Kuusentaimi on mahdollista tunnistaa muusta metsämaastosta reaaliaikaisesella algoritmillä



Kuva 3.1 *Taimikon raivauksen kustannukset [15]*

Tutkimuskysymykset:

- Minkälainen konenäköalgoritmi vaaditaan kuusentaimen tunnistamiseksi?
- Onko kuusentaimet havaittavissa sekä lehdellisenä, että lehdettömänä vuodenaikana?
- Onko tunnistaminen tarpeeksi luotettavaa käytännönsovelluksissa hyödynnettäväksi?

4. TAIMEN TUNNISTUSMENETELMÄ

Suurimmassa osassa taimikonhoitoa on tarkoitus raivata tilaa kuusen-, tai männyntaimille. Ne molemmat ovat havupuita ja lähes kaikki niitä nopeammin kasvavat puulajit ovat lehtipuita. Ihmissilmä pystyy helposti erottamaan havupuun lehtipuusta, osittain tekstuurin ja osittain sävyeron perusteella. Kuvassa 4.1 on esitetty kuusentaimen ja koivunoksan välinen sävyero. Erityisesti valoisalla säällä vihreän sävyero kuusen havuissa ja koivun lehdissä on ihmissilmän helposti erotettavissa. Kyseistä kuvaa käytettiin algoritmin kehitysvaiheessa testikuvana, sillä se sisältää yleisimmät algoritmilta vaadittavat ominaisuudet:

- Kuusenhavun tunnistamisen lehtipuusta,
- kuusenhavun tunnistamisen turpeesta ja varvikosta, sekä
- kuusenhavun tunnistamisen voimakaskontrastisista varjoista

Kuusentaimen tunnistus ylhäältäpäin tekstuurin perusteella on mahdollista[7], mutta algoritmin soveltaminen sulautetuksi järjestelmäksi osana pienmetsäkoneetta on haastavaa sen vaatiman laskentatehon takia. Taimen tunnistusalgoritmia lähdettiin kehittämään nimenomaan käytettäväksi reaaliaikaisesti sulautetuissa järjestelmissä, joissa on rajallinen määrä laskentatehoa verrattuna pöytätietokoneisiin.

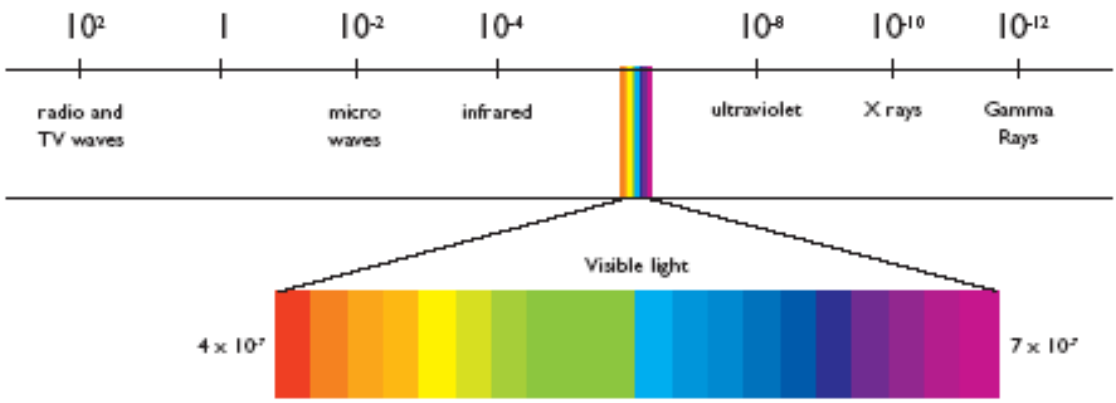
Nimenomaan reaaliaikaisuus tuottaa yhden algoritmin suurimmista haasteista. Algoritmin on pystyttävä prosessoimaan aineistonsa ennalta määritetyssä aikaikkunassa, jonka suuruus riippuu sovelluksesta. Tämän työn tapauksessa algoritmin on pystyttävä tunnistamaan yksittäinen kuva niin nopeasti, että metsäkoneen ohjausjärjestelmä ehtii tarpeen mukaan reagoida halutulla tavalla, esimerkiksi pyrkien väistämään lähestyvän taimen. Tämä tarkoittaa maksimissaan muutamia satoja millisekunteja, mutta mitä nopeammin algoritmi toimii, sitä enemmän kuvia se ehtii prosessoida ja siten voidaan olettaa sen toimintavarmuuden kasvavan ja reaktioajan laskevan.

Kuva 4.1 *Kuusentaimi ja koivunoksa vierekkäin*

Koska kuvasta värin tunnistaminen vaatii yleensä vähemmän laskentatehoa kuin tekstuurin tunnistaminen, päätettiin lähteä kehittämään väriin pohjautuvaa yksinkertaista ja siten myös sulautetuissa järjestelmissä reaaliaikaista algoritmia kuusentaimen tunnistamiseksi. Jo alkuvaiheessa arveltiin, että tällainen algoritmi ei välttämättä ole toimiva kaikista haasteellisimmissa olosuhteissa loppukesästä, kun havupuiden ympärillä on paljon lehtikasveja. Jos algoritmi tästä huolimatta toimisi edes lehdettöminä vuodenaikoina, olisi se käyttökelpoinen koneellisen raivauksen työtehon kasvattamisessa, sillä huomattava osa metsänhoidosta tehdään joko keväisin tai syksyisin, kun lehtipuut ovat lehdettömiä.

4.1 Kuvan tunnistusmenetelmät

Käytettävän tunnistusmenetelmän tulee pystyä minimissään erottamaan havupuut lehtipuista ja metsämaastosta, mutta mieluusti myös kuusentaimet männyntaimista.



Kuva 4.2 *Sähkömagneettisen säteilyn spektri*

Menetelmä voi hyödyntää eri valon aallonpituuksia ja erilaisia kuvankasäittelyalgoritmeja. Kuvankäsittelyalgoritmit voidaan karkeasti luokitella tekstuuriin perustuviin algoritmeihin, sekä väriin perustuviin algoritmeihin. Vaikka algoritmit yleensä ovat monikäyttöisiä eikä niitä sellaisinaan voida luokitella näin, on tässä tapauksessa käytännöllistä lajitella algoritmit niiden käyttötavan mukaan tekstuuripohjaisiin ja väripohjaisiin. Tekstuuripohjaisilla algoritmeilla tarkoitetaan sellaisia algoritmejä, jotka pyrkivät etsimään tai muuten käsittelemään kuvasta löytyviä värialueita tekstuureina, tai alueina. Väripohjaiset algoritmit taas eivät ota kantaa kuvan pikseleiden välisiin relaatioihin, vaan ne käsittelevät yksittäiset pikselit sellaisinaan.

4.1.1 Käytettävän valon aallonpituus

Ultraviolettivalolla tarkoitetaan sähkömagneettista säteilyä, jonka aallonpituus on 10 - 400 nanometriä. Se on suurienergisempää kuin näkyvä valo, joka sijoittuu aallonpituuksille 400 - 750 nm. Infrapunavalon aallonpituus taas on lyhyempi kuin näkyvän valon, 750 nm - 1 mm. Ihmissilmä pystyy havaitsemaan käytännössä vain näkyvänvalon aallonpituudet, joka on alueena hyvin kapea verrattuna ultraviolettiin ja infrapunaan, kuten kuvassa 4.2 on esitetty. [8]

Voidaan olettaa, että metsä näyttää hyvin erilaiselta, jos mukaan otetaan ihmissilmälle näkymättömät taajuudet. Professori Ari Visan mukaan havu- ja lehtipuut eroavatkin toisistaan huomattavasti infrapunavalossa, etenkin auringonpaisteella. Sää vaikuttaa kuitenkin paljon infrapunataajuuksiin ja etenkin vesisateella puiden tunnistaminen toisistaan infrapunalla on haasteellista [18].

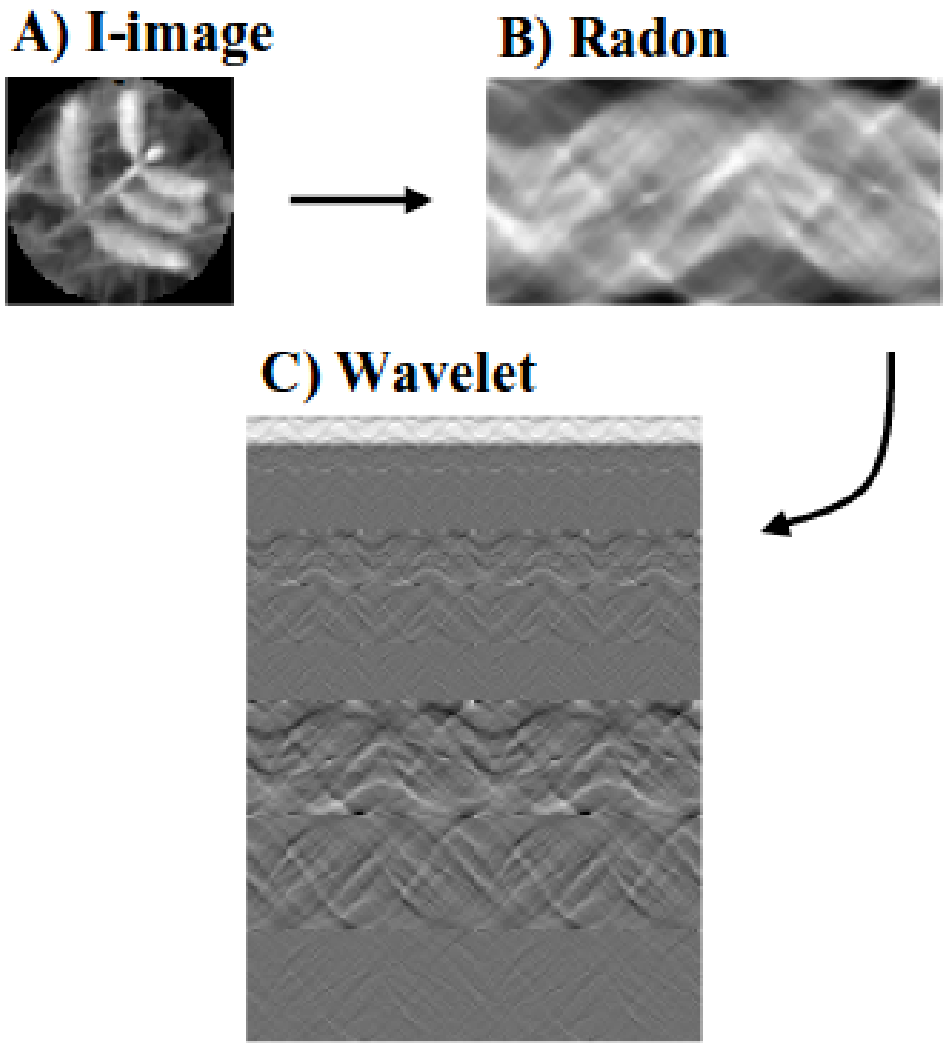
Ihmissilmä pystyy havaitsemaan sähkömagneettisesta säteilystä vain hyvin pienen alueen, aallonpituudeltaan 400 - 750 nanometriä. Vaikka taajuusalue on pieni, pystyy silmämme erottelemään eri aallonpituudet tarkasti eri väreiksi. Toisinkuin ultravioletin ja infrapunan kanssa, on markkinoilta saatavissa paljon edullisia näkyvän valon kameroita. Osittain tästä syystä päädyttiin kehittämään kuusentunnistusalgoritmia juuri näkyvän valon pohjalta. Jos näkyvä valo riittäisi havupuun tunnistamiseen, olisi se kustannustehokkuudeltaan hyvä ratkaisu. Jos taas pelkän näkyvän valon todettaisiin olevan riittämätön havun tunnistamiseen, voisi algoritmia kehittää tulevaisuudessa lisäämällä perinteisen kameran rinnalle ultravioletti- tai infrapunakameran, jolloin saataisiin lisättyä järjestelmän erottelukykyä kustannusten nousun ja laskentatehon hinnalla.

4.1.2 Värialgoritmit

Värialgoritmeilla tarkoitetaan tässä yhteydessä kuvankäsittelyalgoritmeja, jotka luokittelevat kuvan pikselit ainoastaan niiden värin perusteella, ottamatta kantaa pikseleiden sijaintiin kuvassa. Yksi tunnetuimmista algoritmeista on k-NN, eli k-Nearest Neightbour. Se on suhteellisen yksinkertainen ja nopea algoritmi, jota voidaan soveltaa helposti pikseleiden värin tunnistamiseen jopa reaaliaikaisesti sulautetuissa järjestelmissä. Nimenomaan sen yksinkertaisuus oli syy miksi se valittiin käytettäväksi tässä projektissa. k-NN:n yksi huonoista puolista on se, että sen suoritukseen kuluva aika on verrannollinen sen käsittelemään materiaaliin. Reaaliaikaisessa sovelluksessa tämä asettaa ylärajan prosessoitavien kuvien resoluutioille. Resoluutio ei kuitenkaan tässä työssä ole rajoittava tekijä, sillä havupuita ei tarvitse tunnistaa kovinkaan kaukaa. Itseasiassa huolimattomasti otettu, liian suuriresoluutioinen kuva voisi aiheuttaa vääriä tuloksia, kuten esimerkiksi yksittäisen maassa makaavan kuusentaimen oksan luokittelun pieneksi kuusentaimeksi.

4.1.3 Tekstuurialgoritmit

Usein laskentaintensiivisemmät kuvnakäsittelyalgoritmit pyrkivät luokittelemaan pikseleitä sekä niiden värin, että niiden sijainnin perusteella. Esimerkki tästä voisi olla tietyn värisen alueen löytäminen kuvasta. Koska luokittelu on usein riippuvainen käsiteltävän pikselin viereisistä pikseleistä, ja koska *viereisyys* voi olla myös rekursiivista, vaativat tekstuuripohjaiset algoritmit usein reilusti laskentatehoa prosessorilta. Yksi esimerkki tekstuuripohjaisista algoritmeista on hahmontunnistaminen.



Kuva 4.3 *Kuusen neulasten tunnistaminen radon ja wavelet muunnoksilla [7]*

Hahmontunnistuksessa pyritään löytämään tietyn värisiä ja muotoisia kuvioita, kuten esimerkiksi kasvonpiirteitä. Kuva 4.3 kuvastaa kuusenhavun tekstuuripohjaista tunnistamista käyttäen radon ja wavelet muunnoksia [7].

Vaikka tekstuuripohjainen havupuun tunnistus tarjoaisi huomattavasti enemmän mahdollisuuksia verrattuna väripohjaiseen, on väripohjaiset algoritmin yksinkertaisuutensa asiosta tehokkaampia sulautetuissa järjestelmissä. Mitä nopeammin tunnistusalgoritmi pystyy prosessoimaan kuvia, sitä useamman otoksen se pystyy ottamaan samasta puusta, keskiarvoistaen tuloksen.

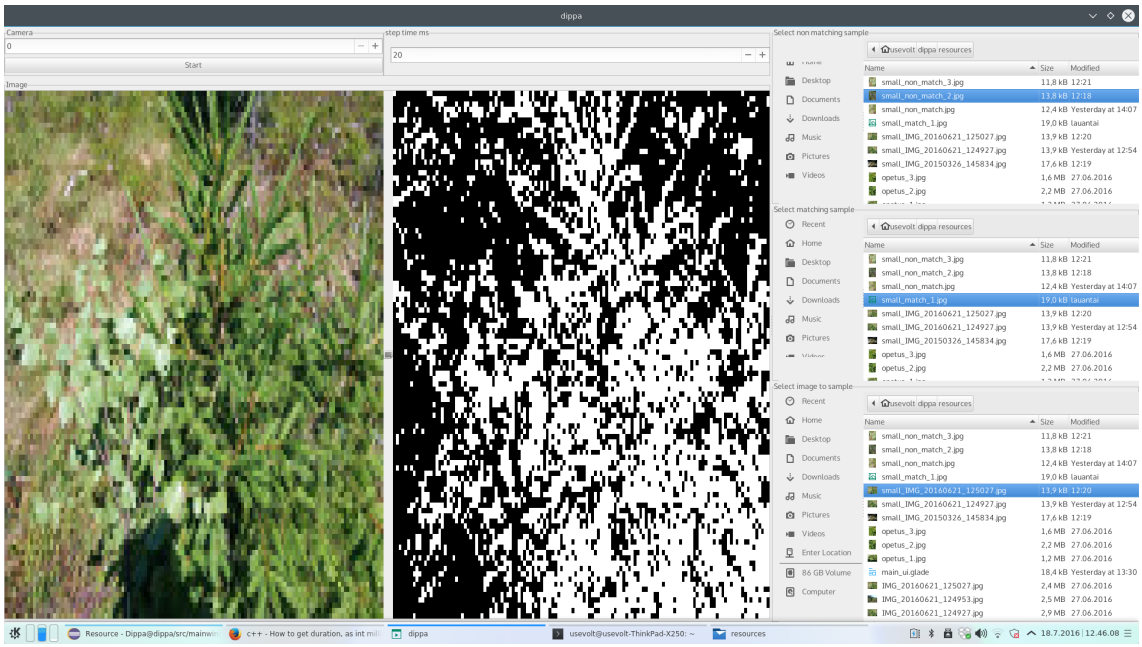
5. ALGORITMIN KEHITYS

Algoritmin kehitykseen käytettiin useita avoimen lähdekoodin työkaluja ja ohjelmistokirjastoita. Kuvankäsittelyalgoritmikirjastona toimi OpenCV [12], jonka tarjoamia algoritmeja käytettiin kuvankäsittelyyn. Testauksen sujuvoittamiseksi tehtiin yksinkertainen ohjelma, jonka avulla eri algoritmien testaaminen oli helppoa. Kuvassa 5.1 näkyy kyseisen ohjelman käyttöliittymä. Ohjelma näyttää prosessoitavan kuvan algoritmin erivaiheissa, sekä sillä pystyy valitsemaan prosessoitavan kuvan ja opetusaineiston algoritmille. Ohjelma kirjoitettiin C++:lla [3], ja sen kääntämiseen käytettiin CMake:a [1]. CMake mahdollisti ohjelman lähdekoodin kääntämisen usealle kehitysalustalle sopivaksi. Graafinen käyttöliittymä kirjoitettiin käyttäen gtkmm-käyttöliittymäkirjastoa [5]. Liitteessä A on listattu koko kyseisen testiohjelman lähdekoodi. Jotta ohjelman kääntö sujuisi ilman virheilmoituksia, tulee OpenCV, gtkmm ja CMake olla asennettuna tietokoneella.

Koska algoritmin on oltava käyttökelpoinen työskenneltäessä metsäkoneella, on sen oltava jopa algoritmeihin perehtymättömän käyttäjän helposti opetettavissa. Opetusaineisto on myös hyvin rajallinen, jolloin laajalla aineistolla opetettavat algoritmit eivät olleet käyttökelpoisia. Luokittelualgoritmiksi valittiin k-Nearest Neighbour sen yksinkertaisuuden ja siten suoritustehokkuuden takia. Algoritmin kehittäminen C++:lla itse kirjoitetulla kehitysohjelmalla tuotti enemmän työtä, mutta samalla se tarjosi vapauden soveltaa kaikkia saatavilla olevia kuvankäsittelykirjastoja, tulevaisuudessa mahdollistaen algoritmin siirtämisen sulautetussa järjestelmässä toimivaksi suhteellisen pienellä vaivalla. K-NN:lle syötettävän väridatan esikäsittelyssä lähestymistapoja oli useita. Niiden testaamisen jälkeen päädyttiin syöttämään data sellaisenaan RGB-muodossa algoritmin välivaiheiden vähentämiseksi [21].

5.1 Esikäsittely

Esikäsittelyllä tarkoitettiin tässä työssä algoritmille annettavan opetusaineiston ja prosessoitavan kuvan muokkausta siten, että siihen sovellettava k-NN algoritmi toi-



Kuva 5.1 *Algoritmin testaukseen kirjoitetun ohjelman käyttöliittymä*

misi halutulla tavalla, tunnistaen kuusenhavun muusta kuvasta. Digitaalisen värikuvan esittämisessä yleisesti käytettyjä väriavaruuksia ovat RGB ja HSV. HSV:n [4] erottelee värisävyn, saturaation ja kirkkauden omaksi kanavikseen, jolloin vihreänsävy voidaan karkeasti tunnistaa pelkästään yhdestä kanavasta. HSV-väriavaruuden kirjo onkin sylinterimäinen kuten kuvasta 5.3 näkyy ja siitä eri värisävyjen, eli *Hue*:n erottaminen onnistuu värivektorin kulman perusteella. RGB [21] lajittelee väritiedon eritavalla jakaen sen punaisen, vihreän ja sinisen sävyihin. HSV väriavaruuden käyttämisestä kuusentaimen tunnistamiseen olisi se hyöty, että värisävyjen eroavaisuuksia olisi helpompi korostaa painottamalla Hue-arvoa k-NN-algoritmissa. Kuitenkin kuten kuvasta 5.4 nähdään, ei yhdessäkään kanavassa kuusenhavu erotu erityisen selkeästi taustasta. Havun sävyn erottaminen vaatiikin käytännössä aina kaikkien kolmen kanavan käyttämistä tunnistamiseen, jolloin HSV-väriavaruuden suurin hyöty menetetään. Koska valtaosa markkinoilta löytyvistä kameroista tallentaa kuvat RGB-väriavaruudessa, olisi HSV:tä käyttäessä kuva jouduttu esikäsittelyssä muuntamaan HSV-muotoon, tuottaen yhden ylimääräisen laskentaoperaation algoritmiin. Väriavaruusmuunnos ei itsessään tuo lisää dataa alkuperäiseen kuvaan, vaan se osittain jopa hukkaa dataa. RGB-kameran antamassa kuvassa on jo hukattu kaikki se data mikä ei mahdu RGB-värikirjon määrittävän punaisen, sinisen ja vihreän värikanavan muodostaman kolmion sisään. Kuten kuvasta 5.2 nähdään, sisäl-

tää digitalinen RGB-kuva huomattavasti pienemmän väridatan kuin mitä ihmissilmä näkee. Ihmissilmä on erityisen herkkä tunnistamaan vihreän erisävyjä, kun taas RGB-väriavaruus ei painota vihreää. Kuusentaimen tunnistaminen keskittyy pitkälti nimenomaan vihreänsävyjen erottamiseen toisistaan, joten perinteiset ja edulliset RGB-kamerat eivät tarjoa yhtä hyvää lähtökohtaa vihreänsävyjen tunnistamiselle kuin ihmissilmä. Jo hukattua dataa ei saada palautettua muuttamalla RGB-kuvaa HSV-väriavaruuteen, joten on todennäköistä ettei RGB-HSV-muunnoksella saavutettaisi olleellista hyötyä algoritmin kannalta. Vaikka RGB-väriavaruus ei tarjoakaan parasta mahdollista lähtökohtaa kuusentaimen tunnistamiselle, päädyttiin käyttämään sitä RGB-kameroiden helpon saatavuuden takia.

Artikkelissaan [7] Hyyti, Kalmari ja Visala esittelivät kolme eri tapaa yhdistää RGB-väriavaruuden kanavat luonnossa nähtävien vihreänsävyjen tunnistamiseksi. Nämä kolme eri yhtälöä ovat:

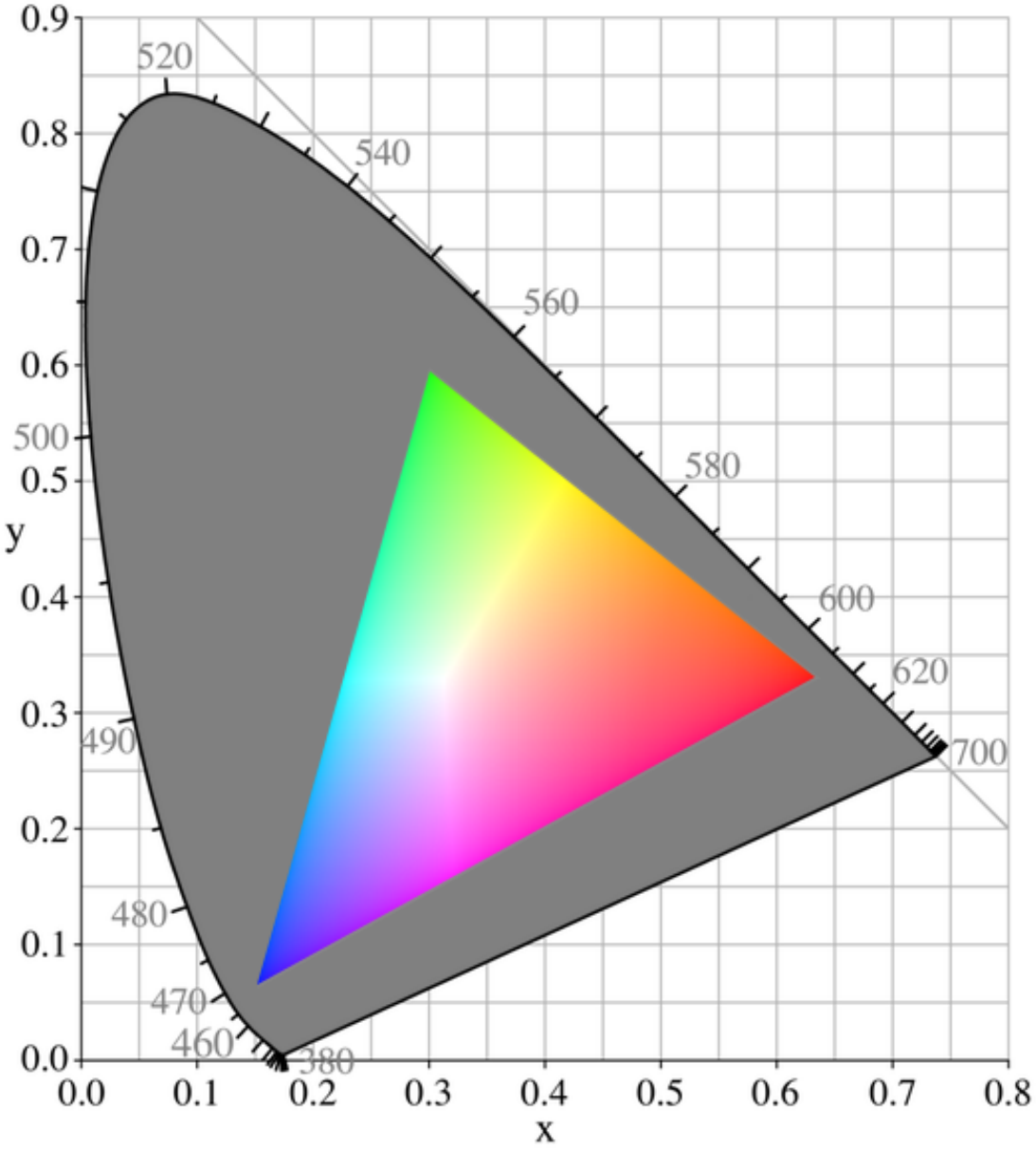
$$I = (R + G + B)/3 \tag{5.1}$$

$$EG = R - B \tag{5.2}$$

$$RB = (2G - R - B)/2 \tag{5.3}$$

Esimerkit yhtälöiden tuloksista näkyvät kuvassa 5.5. Näistä EG erottaa kuusentaimen vihreänsävyn melko hyvin ympäröivästä heinikosta, mutta kun rahkasammaleisessa maastossa olevista kuusista otettuja kuvia käsittelee kyseisellä yhtälöllä, on tulos kuvan 5.6 kaltainen. Vaikka tästäkin kuvasta paistaa hyvinkin vaalealla kuusen runko, on ero ympäristöön liian pieni luotettavan tunnistuksen takaamiseksi mahdollisimman monissa eri tilanteissa. Kuusen luotettava tunnistaminen vaatisi värikäsittelyn jälkeen tekstuuripohjaisia algoritmeja, joita pyrittiin välttämään niiden vaatiman laskentatehon takia. Siksi myös RGB-kanavien yhdistäminen hylättiin ja päädyttiin käsittelemään "raakaa"RGB-kuvaa sen omissa kanavissaan.

K-NN-algoritmille annettava kuvadata oli esikäsiteltävä kevyesti, jotta algoritmista saataisiin reaaliaikainen sulautetuissa sovelluksissakin. k-NN:n suoritukseen kulutettava aika on verrannollinen sen opetukseen käytettävän materiaalin määrään [6]. Tämä tarkoittaa sitä, että opetuskuvien ja analysoitavien kuvien resoluutio kannattaa pitää mahdollisimman pienenä. Kuvadatan resoluutio onkin avainasemassa niihin peruslähtökohtiin mistä algoritmiä lähdettiin alunperin kehittämään. Tekstuuriin



Kuva 5.2 *RGB-väriavaruuden määrittelemä kirjo on ihmissilmän näkemän väriavaruuden osajoukko [20]*

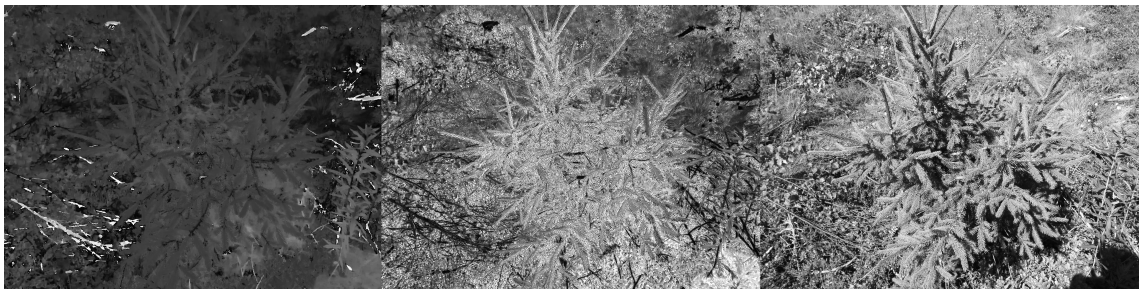
pohjautuvat algoritmit ovat jo sellaisinaan usein monimutkaisempia kuin väriin perustuvat, mutta sen lisäksi ne vaativat usein kuvilta myös suurempaa resoluutiota, kuin väripohjaiset algoritmit. Tämä kaikki näkyy laskentatehossa ja reaaliaikasovelluksessa siinä, kuinka suurella taajuudella pystytään kuvia prosessoimaan.

Koska koneellisessa raivauksessa sovellettavan taimentunnistusalgoritmin tarvitsee

Kuva 5.3 *HSV-väriavaruuden sylinterimäinen kirjo*

tunnistaa kuusentaimet vain suhteellisen lyhyeltä etäisyydeltä, on kuvien resoluutio usein paljon suurempi kuin mitä pelkän värin tunnistamiseen edes tarvittaisiin. Algoritmin esikäsittelynä toimikin vain kuvien resoluution pienentäminen tarpeeksi pieneksi. Koska "sopivan pieni"riippuu paljon käyttötarkoituksesta, on sopivan resoluution määritteleminen sovelluskohtaista. Testivaiheessa kuvien resoluutiona käytettiin $133 * 100$ pikseliä, mistä ihminen pystyy hyvin erottamaan muutaman metrin päässä olevat kuusentaimet toisistaan. Resoluutiota olisi siis todennäköisesti mahdollista pienentää vielä reilusti tehokkuuden kasvattamiseksi.

Algoritmin ensimmäisissä versioissa käytettiin OpenCV-konenäkökirjaston tarjoamia kuvankäsittely- ja k-NN-funktioita. Koska OpenCV käyttää sisäisesti BRG-



Kuva 5.4 *Kuusentaimen kuvan HSV-muodossa. Kanavat ovat järjestyksessä H (värisävy) S (saturaatio) ja V (kirkkaus) vasemmalta lukien*

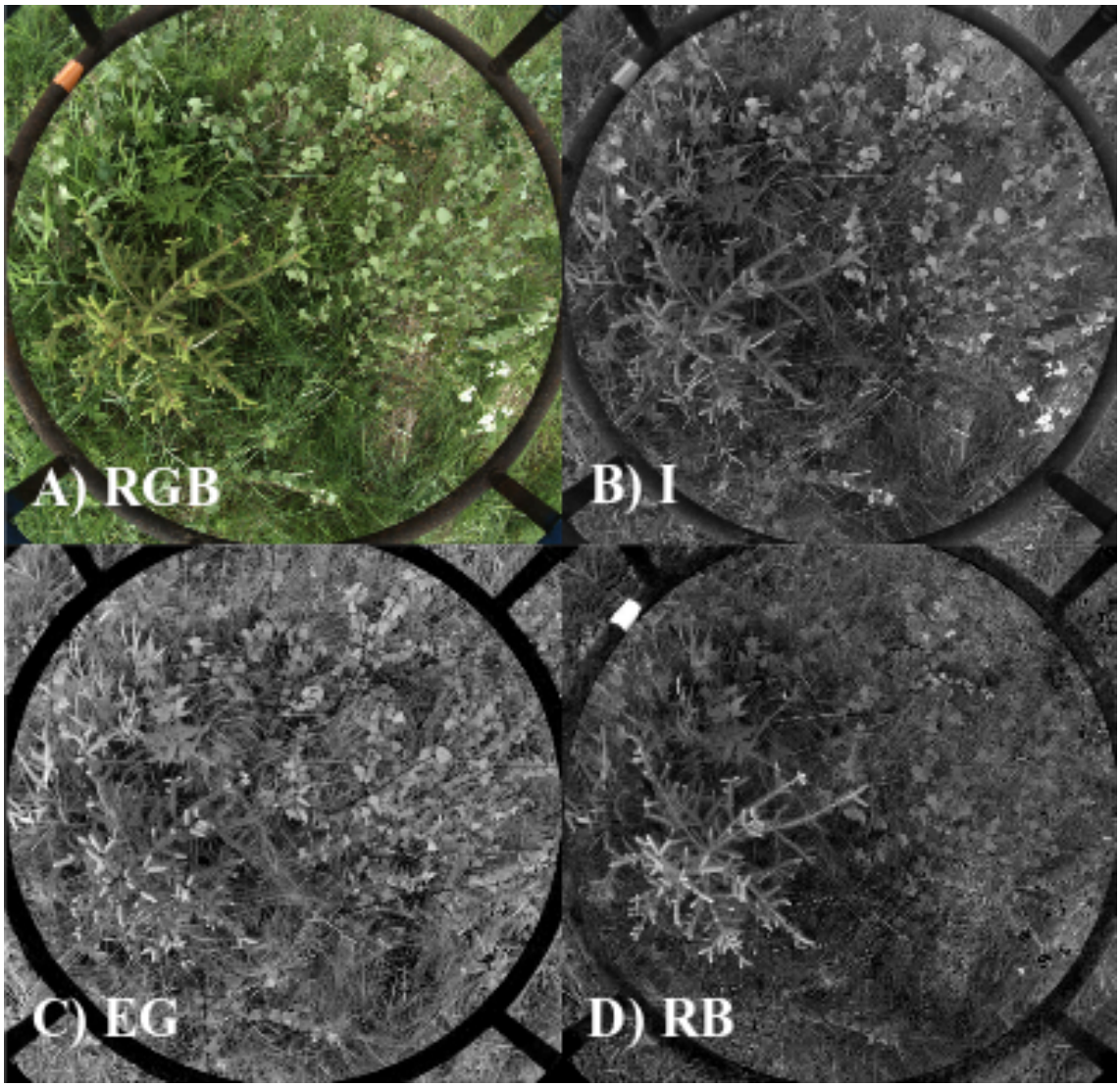
väriavaruutta RGB:n sijaan, ohjelmiston sovittaminen graafisen käyttöliittymän ja webkameran antaman kuvadatan kanssa vaati värikanavien uudelleenjärjestämisen. Tietokoneelle tämä on nopea ja helppo muunnos, mutta sulautetuissa järjestelmissä muisti on usein hyvinkin rajallista ja sen ylimääräinen käsittely voi vaatia hetkellisesti hyvinkin suuren määrän ylimääräistä muistia, sekä laskentatehoa. Tämän takia ylimääräiset muunnokset tulisi minimoida.

5.2 K-nearest neighbour

K-nearest neighbour algoritmi, eli k-lähin naapuri, on paljon käytetty, laskennaltaan yksi yksinkertaisimmista lajittelualgoritmeista. k-NN:n tarkoitus on lajitella data ennalta määritettyihin luokkiin etsien annetusta malliaineistosta samantyyppisiä alkioita. Algoritmin nimessä esiintyä k määrittää kuinka monta *lähintä naapuria* jokaiselle alkiolle etsitään ja näistä äänestetään mihin luokkaan alkio kuuluu. K-NN on yksinkertaisuutensa ansiosta hyvin muokkautuva algoritmi ja sitä voidaan soveltaa testiaineistoon huolimatta siitä, kuinka moniulotteinen testiaineiston joukko on kyseessä.

K-NN-algoritmia voidaan käyttää n-ulottuvuuksisen avaruuden pistejoukon lajitteluun. Jokaisen pisteen sijainti on määritelty n-ulottuvuuksisena vektorina

$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ \dots \\ n_n \end{bmatrix}, \tag{5.4}$$



Kuva 5.5 *Vihreän sävyjen erottuminen yksinkertaisilla värimuutoksilla [7]*

jonka jokainen koordinaatti vastaa pisteen sijaintia kyseisen akselin suuntaan. K-NN lajittelee pisteitä niiden eukliidisen etäisyyden

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n}(x_i(n_r) - x_j(n_r))^2}, \tag{5.5}$$

perusteella. Koska absoluuttiset etäisyydet eivät ole k-NN:lle merkittäviä, vaan pisteitä luokitellaan ainoastaan niiden etäisyyksillä suhteessa toisiinsa, voidaan yksittäisen pisteen etäisyys origosta määrittää ilman tietokoneelle laskentaintensiivistä

Kuva 5.6 *Kuusen vihersävy haastavammassa maastossa*

neliöjuurilaskentaa [6]. Tällöin pisteen etäisyys on

$$d(x_i) = n_1^2 + n_2^2 + ... + n_n^2, \tag{5.6}$$

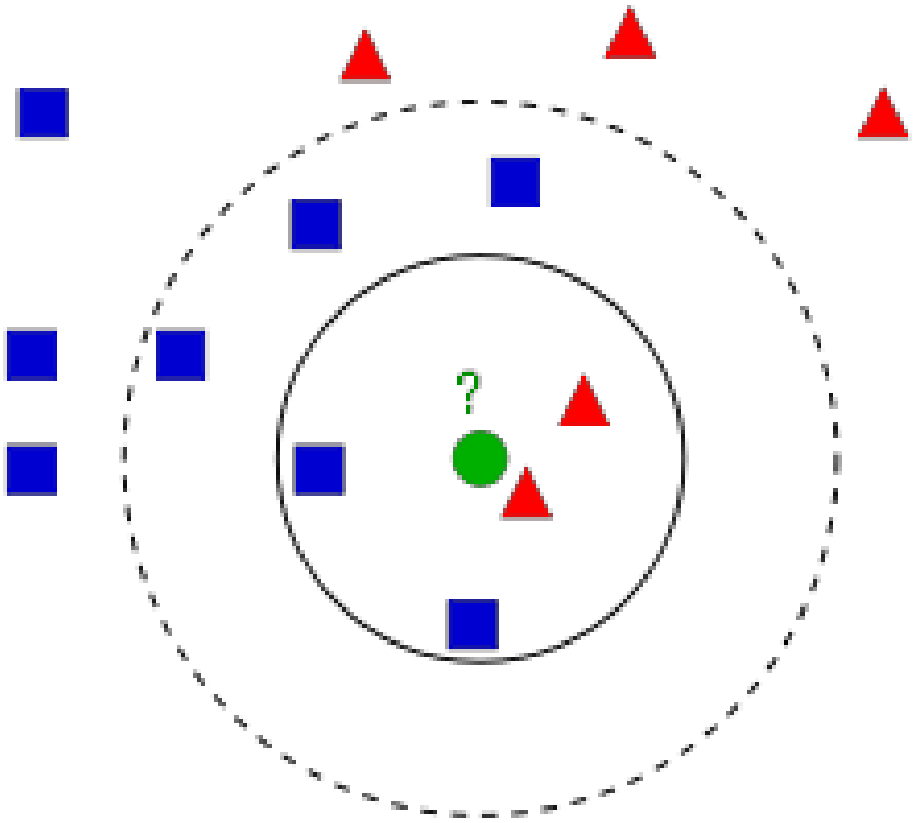
Näitä neliöllisiä etäisyyksiä käytetään usein reaaliaikaisessa 3D-laskennassa, esimerkiksi peleissä ja simulaattoreissa [19].

Tämän työn sovelluksessa jokainen kuvan pikseli vastaa yhtä pistettä, ja pikselin jokainen värikanava vastaa yhtä ulottuvuutta. K-NN-algoritmi toimii siis kolmiulotteisessa avaruudessa, jossa yksittäisen pikselin värin etäisyys origosta on

$$d(px) = R^2 + G^2 + B^2, \tag{5.7}$$

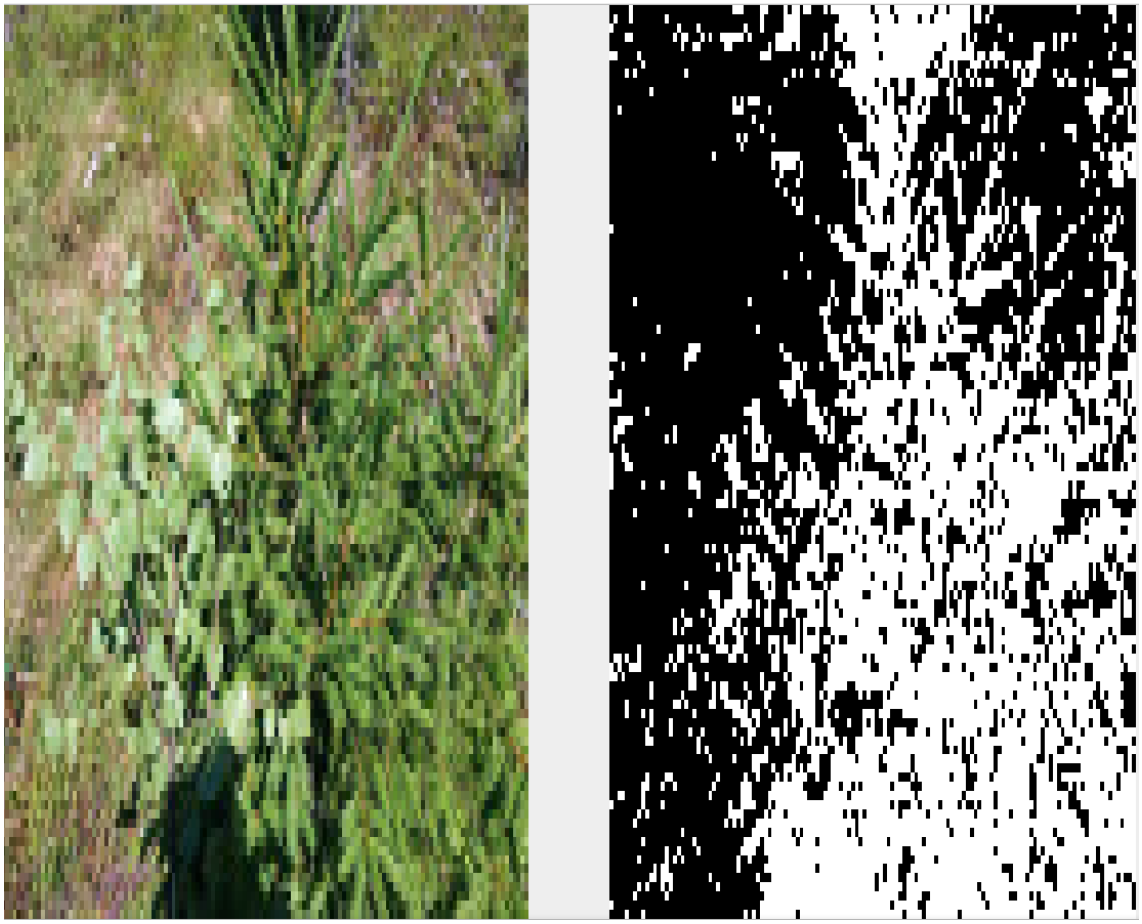
jossa R on sinisen, G punaisen ja G vihreän värikanavan intensiteetti.

Kun yksittäisen pisteen etäisyys on laskettu, verrataan sitä opetusmateriaalin ennalta luokiteltuihin pisteisiin. Opetusmateriaalista valitaan k-kappaletta pisteitä joiden etäisyydet ovat lähimpänä prosessoitavan pisteen etäisyyttä. Se mihin luokkaan



Kuva 5.7 *k-NN algoritmin havainnollistus [13]*

kuuluvia pisteitä näistä k-kappaleesta pisteitä on eniten, määritetään prosessoitavan pisteen luokaksi. Prosessia havainnollistaa kuva 5.7. Kuvassa jokainen piste luokitellaan joko siniseksi neliöksi tai punaiseksi kolmioksi. Siniset ja punaiset pisteet on jo ennalta luokiteltuja ja vihreää ympyrää ollaan luokittelemassa. Kyseessä on havainnollisuuden vuoksi 2-ulotteinen avaruus, jolloin pisteiden sijainti on helppoa kuvata. Punaisten ja sinisten pisteiden joukosta valitaan k-kappaletta vihreää ympyrää lähinnä olevaa pistettä. Kuvassa 5.7 näkyvät kaki ympyrää havainnollistavat tilanteita kahdella eri k:n arvoilla. Sisempi musta ympyrä kuvastaa tilannetta $k = 4$ ja ulompi, katkoviivainen ympyrä kuvastaa tilannetta $k = 7$. Koska k-kappaleesta tunnettuja pisteitä äänestetään vihreän ympyrän tuleva luokka, nähdään kuvasta ongelma mikä esiintyy, jos $k = 4$. Tällöin vihreä ympyrää lähimpänä on kaksi sinistä neliötä ja kaksi punaista kolmiota. Äänestystulos menee tasan eikä vihreälle neliölle voida määrittää luokkaa. Tilanteessa $k = 7$ sinisiä neliöitä on 5 ja punaisia kolmioita vain 2, joten vihreä ympyrä luokitellaan siniseksi neliöksi. On siis tärkeää, että k:n arvoksi valitaan pariton luku. Näin varmistetaan että luokkaäänestykset



Kuva 5.8 *k-NN algoritmin tulos*

eivät päädy tasan ja kaikki alkiot saadaan luokiteltua. K:n arvon valinta voi olla sovelluksesta riippuen tärkeää, sillä kuten kuvasta 5.7 näkee, vaikuttaa k:n arvo paljon siihen miten alkiot luokitellaan. Kuten algoritmin kehityksen myöhemmissä vaiheissa todettiin, ei k:n arvo kuitenkaan vaikuta oleellisesti algoritmin nopeuteen, sillä algoritmi pystytään optimoimaan niin, ettei lähdeaineistoa tarvitse selata läpi kuin kerran jokaista tarkasteltavaa pistettä kohden k:n arvosta huolimatta.

Jokaiselle prosessoitavan RGB-värikuvan pikselille suoritetaan sama k-NN-algoritmi, joka tapahtuu 3-ulotteisessa väriavaruudessa. Koska tässä sovelluksessa tarvitsee vain tunnistaa onko kuvassa havupuuta vai ei, on opetusmateriaali yksinkertaista ja jokainen pikseli luokitellaan totuusarvoksi *true* tai *false*. Yksinkertainen luokittelu kasvattaa algoritmin toimintavarmuutta ja helpottaa sen opettamista, mutta jos luokittelua lisättäisiin, voitaisiin erotella pikseleiden värit useampaan luokkaan samalla kertaa, erotellen esimerkiksi maasto, havupuut ja lehtipuut toisistaan.

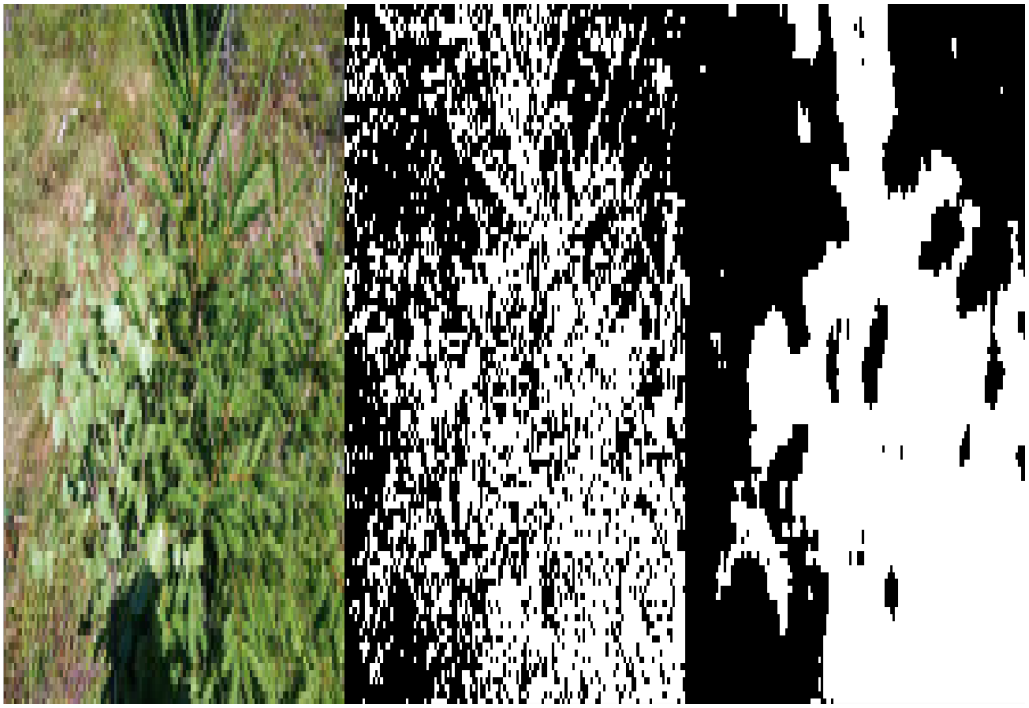
Kuva 5.9 *k-NN algoritmin opetuskuvat*

Kuva 5.8 esittää testikuvaa ja sopivalla testiaineistolla opetetun k-NN-algoritmin antamaa ihanteellista tulosta. Kuvan 5.8 oikea puolisko havainnollistaa k-NN-algoritmin tulosta. Algoritmi on opetettu kuvassa 5.9 olevalla aineistolla tunnistamaan kuusenhavut taustasta. Näistä vasemmanpuoleinen kuva on luokiteltu "osumaksi"ja oikeanpuoleinen kuva "hudiksi". Tärkeää on että "huti"sisältää sekä lehtipuun lehtiä, että maastoa, kun taas "osuma"sisältää vain ja ainoastaan tunnistettavaa materiaalia, tässä tapauksessa kuusenhavuja. Kuten kuvasta 5.8 näkyy, on k-NN luokitellut hyvinkin tarkasti kuusenhavut erilleen taustasta. Huomattavissa oleva virhe on varjot, jotka algoritmi on luokitellut myös "osumaksi". Tämä johtuu siitä, että opetuskuvista kuusenhavukuva on sävyiltään huomattavasti tummempi kuin "huti-kuva, jolloin tummilla pikseleillä on taipumusta tulla luokitelluksi osaksi havuja. Ongelma tulisi ratkaista sisällyttäen opetusaineistoon materiaalia myös lehtipuiden varjoista, jolloin algoritmi ei luokittelisi pikseleitä pelkästään niiden tummuuden perusteella.

5.3 Jälkikäsittely

K-NN-algoritmin jälkeiset laskennallisesti suhteellisen kevyet toimenpiteet luokiteltiin jälkikäsittelyksi. Terminä jälkikäsittely on vähättelevä verrattuna sen tärkeyteen soveltaessa algoritmia mihin tahansa mekatroniseen järjestelmään, sillä jälkikäsittely vastaa k-NN-algoritmin antamien tulosten tulkitsemisesta ja täten myös lopullisesta päätöksestä, mikä luokitellaan taimeksi ja mikä ei. Jälkikäsittely on avainasemassa

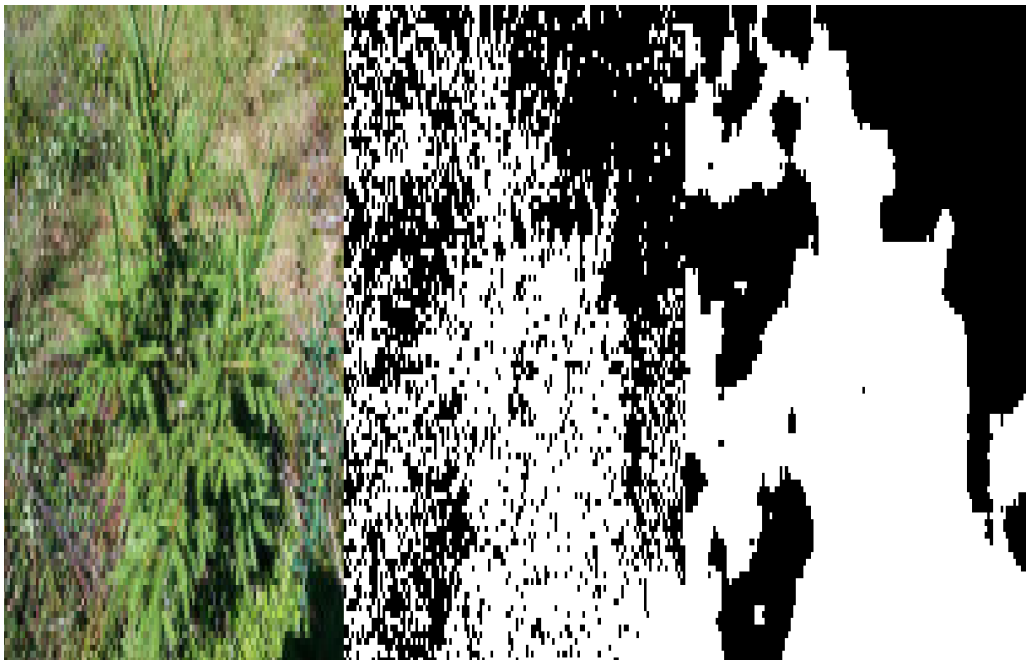
sovelluksen toiminnasta ja tunnistuksen herkkyydestä.



Kuva 5.10 *Kuusentaimen tunnistusalgoritmin välivaiheet*

Kuva 5.10 havainnollistaa algoritmin toimintaa. Vasemmanpuoleinen kuva on algoritmille annettu kuva josta kuusentaimi tulee tunnistaa. Keskimmäinen kuva on k-NN:n käsittelyn jälkeinen vaihe ja oikeanpuoleisin kuva on sopivan jälkikäsittelyn tulos. Jälkikäsittelynä toimii kuvan sumentaminen ja kontrastin lisääminen niin, että saadaan keskiarvoistettua k-NN:n lajittelemat mustat ja valkoiset pikselit toisistaan. Tämän jälkeen pystytään laskemaan valkoisten "läiskien"pinta-ala ja määrittelemään raja kuinka suuri valkoinen läiskä hyväksytään taimeksi. Toleranssien määrittäminen kuuluu osaksi algoritmin opettamista ja varsinaisessa sovelluksessa sen pitäisi olla käyttäjän säädettävissä.

Kuvassa 5.11 on sovellettu samaa kuvan 5.9 opetusaineistoa eri kuvaan kuusentaimesta. Algoritmin voidaan tulkita löytäneen kuusi kuvan keskeltä, mutta huomattavaa epätarkkuutta on aiheutunut kuvan vasemmassa laidassa oleva varvikko. Koska kuvan 5.9 opetusaineisto on melko rajoittunut, on luonnollista että algoritmi ei toimi täydellisesti erilaisessa aineistossa. Tässä sovelluksessa haluttiin erityisesti tunnistaa havupuu muusta ympäristöstä. Intuitiivisesti tämä voisi tarkoittaa sitä, että algoritmin opetusaineistossa tulisi olla painotettuna kuvat kuusenhavuista. Kuten



Kuva 5.11 *Sama opetusaineisto kuin kuvassa 5.10 sovellettuna eri kuvaan*

kuvasta 5.11 nähdään, asia on kuitenkin juuri toisinpäin. On tärkeämpää opettaa algoritmille se, mikä vihreänsävy *ei* ole havua. Metsämaastossa on kuusenhavua sekä tummempia, että myös vaaleampia vihreänsävyjä kuin havut, minkä takia algoritmi on opetettava tunnistamaan ne molemmat, välttäen niiden virheellistä tulkitsemista havuiksi.

6. TAIMENTUNNISTUSMENETELMÄN TOIMINTA JA SUORITUSKYKY

Taimentunnistusalgoritmiä kehitettiin testikuvilla, jotka oli otettu sekä lehdellisenä, että lehdettömänä vuodenaikana. Näin pystyttiin todentamaan algoritmin toiminta molemmissa tapauksissa. Kuten aiemmassa luvussa todettiin, on k-NN-algoritmin toiminnalle tärkeää valita sopiva k:n arvo. Taulukossa 6.1 on esitetty algoritmin suorituskyky eri k:n arvoilla käytettäessä *1 000x750* resoluutioista kuvaa, sekä kahta *20x27* resoluutioista opetusaineistokuvaa. Opetusaineistokuvista toinen esitti kuusentaimea ja toinen muuta maastoa. Siitä nähdään, että k:n muuttaminen ei tässä tapauksessa vaikuttanut algoritmin tarkkuuteen, mutta lisäsi algoritmin suoritusaikaa hieman. Mittaukset tehtiin suuriresoluutioisella kuvalla, jotta erot saataisiin paremmin näkyviin. Näiden tulosten pohjalta päädyttiin käyttämään k:n arvoa 3, sekä todettiin opetusmateriaalin määrän ja laadun olevan k:n arvoa merkittävämpiä seikkoja algoritmin toiminnan kannalta. Vaikka k arvolla 1 antoi nopeimman mittaustuloksen, jätettiin se valitsematta, sillä *k=1* tarkoittaa nearest-neightbour -algoritmia. Taulukon 6.1 tulokset viittaavat siihen, että tulosten jakauma on symmetrinen, jolloin k-NN algoritmi ei tuo erityistä hyötyä. Koska algoritmin testaamisessa käytetty testiaineisto oli melko suppea, ei voida riittävällä varmuudella sanoa, onko kaikki metsämaastosta otetut kuvat jakaumaltaan symmetrisiä. Häntivän jakauman tapauksessa k-NN toimii paremmin kuin nearest-neighbour, ja tästä syystä päädyttiin arvoon *k=3*.

Taulukko 6.1 *k-NN algoritmin toiminta eri k:n arvoilla*

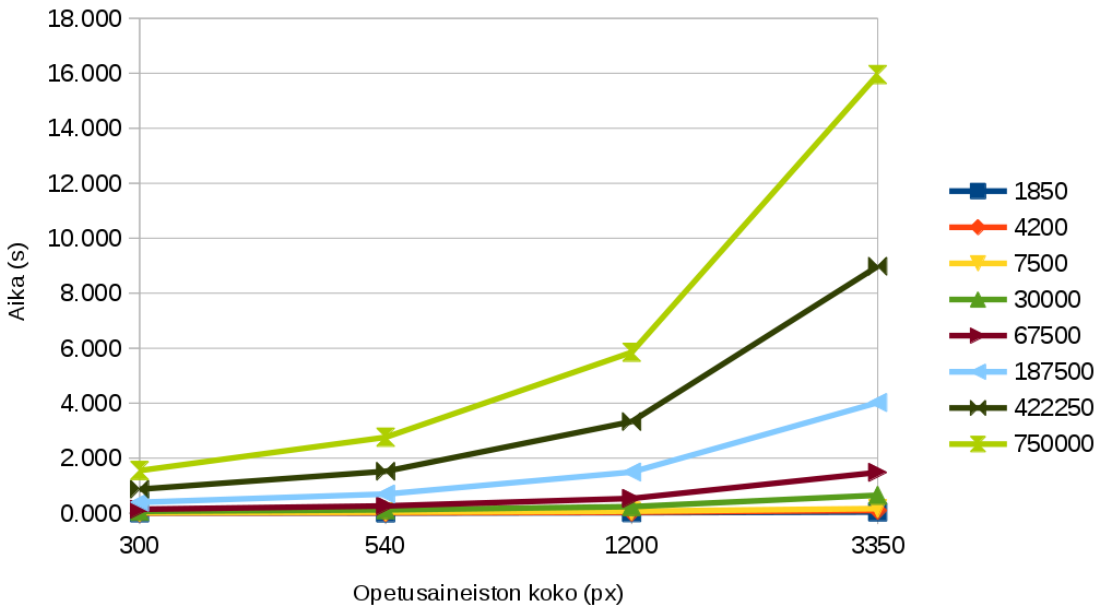
k	Tarkkuus	Suoritusaika
1	71 %	3,47 s
3	71 %	3,59 s
5	71 %	3,73 s
7	71 %	3,84 s
9	71 %	3,96 s

Kuva 6.1 *Kuva kuusentaimesta resoluutioilla 3 264 x 2 448 ja 100 x 75.*

6.1 Suorituskyky

Havupuuntaimen tunnistusalgoritmin suorituskyvyn pullonkaula on k-NN-algoritmi. Yksi k-NN:n heikkouksista on sen suoritusajan verrannollisuus opetusmateriaalin määrään. Koska prosessoitavan kuvan jokaista pikseliä verrataan kaikkiin opetusmateriaalin pikseleihin, on helpoin tapa redusoida algoritmin suoritusaikaa vähentämällä sekä opetusmateriaalia, että prosessoitavan kuvan resoluutiota. Luonnollisesti tämä laskee algoritmin erotuskykyä, sillä informaatiota menetetään resoluutiota heikentämällä. Väripohjaisen tunnistuksen hyvä puoli on kuitenkin se, ettei algoritmin toiminta ole riippuvainen pienistä yksityiskohtaisista tekstuureista, vaan tietyn alueen keskiarvoistetusta väristä. Resoluutiota pienennettäessä värit keskiarvoistuvat pikseleiden kesken riippuen käytetystä skaalausalgoritmista.

Kuvassa 6.1 on kaksi esimerkkikuvaa kuusesta. Vasemmanpuoleisen kuvan alkuperäinen resoluutio on *3 265 x 2 448* pikseliä, joka on niin tarkka, että kuusesta erottuvat selkeästi yksittäiset neulasetkin. Oikeanpuoleisessa kuvassa sama kuva kuusentaimesta on pienennetty resoluutioon *100 x 75* pikseliä. Ihmissilmä pystyy vielä erottamaan kuvan keskeltä kuusen muodon ilman suurempia vaikeuksia. Näiden kahden kuvan suurin ero on niiden koossa. Oikeanpuoleinen kuva sisältää $\frac{3\ 264 * 2\ 448}{100 * 75} = 1\ 065,3696$ kertaa vähemmän informaatiota kuin vasemmanpuoleinen kuva. Tämä tarkoittaa, että k-NN-algoritmi pystyy käsittelemään pienempiresoluutioisen kuvan arviolta tuhat kertaa nopeammin kuin suuremman kuvan. Jos tämän lisäksi k-NN:n opetusaineisto on annettu samoilla resoluutioilla, nousee algoritmin suoritusaika opetuskuvien lukumäärän mukaiseen potenssiin. Tämä tarkoittaa, ettei ole järkevää edes yrittää käyttää algoritmia turhan suuriresoluutioisiin



Kuva 6.2 *Taulukon 6.2 data kuvaajana*

kuviin, vaan kuvien resoluutio kannattaa pitää niin pienenä kuin sovelluskohde suinkin sallii. Ratkaisevana tekijänä toimii se, miten pienen objektin algoritmin täytyy pystyä kuvasta tunnistamaan.

Taulukko 6.2 *Algoritmin suoritusnopeus eri resoluutioisella datalla ja opetusaineistolla*

Datan koko (px)	**Opetusaineisto 300 px (s)**	**Opetusaineisto 540 px (s)**	**Opetusaineisto 1 200 px (s)**	**Opetusaineisto 3 350 px (s)**
1 850	0,010	0,007	0,016	0,042
4 200	0,009	0,015	0,034	0,098
7 500	0,018	0,038	0,070	0,165
30 000	0,064	0,119	0,236	0,651
67 500	0,140	0,262	0,535	1,481
187 500	0,394	0,692	1,492	4,022
422 250	0,870	1,522	3,331	8,979
750 000	1,548	2,760	5,851	15,952

Taulukossa 6.2 on listattu algoritmin testiversion suorituskykyä eri lähdemateriaaleilla. Taulukkon arvoja vastaavat lähdekuvat ja tulokset puolestaan on kuvassa 6.7. Testi suoritettiin Intel i5-prosessorisella Lenovo ThinkPad X250 kannettavalla tietokoneella ja suoritusajat mitattiin ohjelmallisesti. On syytä huomioida, että mittausaikoihin on voinut vaikuttaa tietokoneen muut yhtäaikaiset prosessit. Täten

varsinkin pieniresoluutioisella lähdemateriaaleilla tehtyjen mittausten mittaustarkkuus ei todennäköisesti ole millisekuntiluokkaa. Tämä voi selittää variaation eri mittausten välillä niissä tapauksissa, joissa opetusaineiston resoluution kasvattaminen nopeutti algoritmin suoritusta epäloogisesti.

Tietokoneen, jolla mittaukset tehtiin, suorituskyvyn voidaan olettaa olevan moninkertainen sulautettuihin järjestelmiin verrattuna. Täytyy pitää mielessä, että taulukossa 6.2 mitatut arvot voivat sulautetussa järjestelmässä suoritettuna olla huomattavasti suuremmat, helposti kymmenkertaiset. Koska yksi tärkeimmistä algoritmin käytännöntoimivuuden vaatimista ominaisuuksista on sen reaaliaikaisuus, voidaan olettaa yksittäisen kuvan käsittelyyn kuluvan maksimiajan olevan joitakin satoja millisekunteja. Jos oletetaan sulautetun järjestelmän kykenevän toistamaan taulukon 6.2 mittaukset kymmenkertaisesti hitaammin, tarkoittaa se käytännössä sitä, että prosessoitavaa dataa ei voi olla enempää kuin 30 000 pikseliä. Tämän jälkeen tietokoneen suoritusajat nousevat satoihin millisekunteihin, joka sulautetussa järjestelmässä voisi tarkoittaa joitain sekunteja.

Kuvaan 6.2 on piirretty kuvaajat taulukon 6.2 datasta. Eri käyrät vastaavat käsiteltävää dataa, kun vaaka-akselilla on opetusdatan määrä. Pystyakselilla on suoritusaika. Kuten kuvasta nähdään, suoritusaika kasvaa eksponentiaalisesti opetus- ja käsiteltävän datamäärän kasvaessa. Tämä selittyy k-NN-algoritmin luonteella, sillä algoritmi joutuu vertaamaan jokaista käsiteltävää data-alkiota kaikkiin opetusdatan alkioihin. Ratkaisu algoritmin suoritusajan minimoimiseen on siis prosessoitava data määrän, kuin myös opetusaineiston datamäärän pitäminen pienenä.

Datan resoluution laskemisen seurauksena tulee algoritmin tunnistusresoluution pienentyminen. Jossain vaiheessa algoritmi ei siis pysty enää tunnistamaan havua ympäröivästä maastosta. Kuvan 6.7 algoritmin tuloksista nähdään lähdeaineiston resoluution vaikutus tunnistustulokseen. Jokaisessa kuvassa on kuusentaimi tunnistettu ympäröivästä maastosta. Suuriresoluutioisessa testituloksissa algoritmin tuloskuvat ovat huomattavasti yksityiskohtaisemmat kuin pieniresoluutioisessa päässä. Testikuvan tunnistettava kuusentaimi on riittävän suuri näkymään pienelläkin resoluutiolla, mutta algoritmin opetusaineksen koko vaikuttaa tulokseen selvästi. Kun algoritmin tuloksia verrataan ihannetulokseen, jossa on käsin eroteltu kuusentaimi taustasta, kuten kuvassa 6.3 on esitetty, voidaan algoritmin toimivuutta tarkastella numeerisesti pikselitasolla. Jokaista k-NN-algoritmin prosessoimaa pikselia verrataan ihannetuloksessa vastaavalla paikalla olevaan pikseliin. Näin saadaan al-

Kuva 6.3 *Käsin eroteltu tunnistuksen ihannetulos*

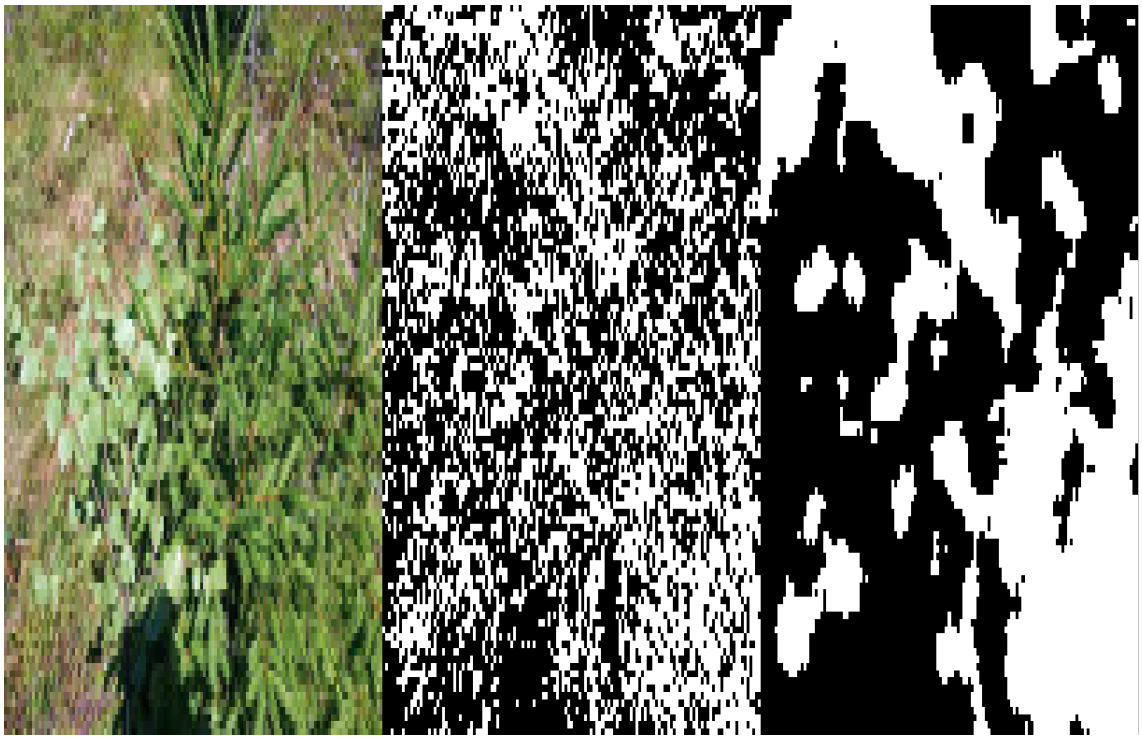
goritmin osumatarkkuuden todennäköisyys, joka on esitetty taulukossa 6.3 samalla opetusmateriaalilla ja lähdeaineistolla, kuin taulukossa 6.2. Huomioitavaa on, ettei opetusmateriaalin ja datan koko vaikuta oleellisesti algoritmin tarkkuuteen. Kaikissa testitapauksissa tarkkuus on 53 ja 67 % välissä, mitä voidaan pitää riittävänä. Todennäköisyyksiä analysoitaessa on syytä pitää mielessä malliesimerkinkin olevan käsin tehty aproksimaatio, joten täyttä 100 % todennäköisyyttä ei ole mielekästä tavoitellakaan.

Taulukko 6.3 *Algoritmin pikselikohtainen tunnistuksen todennäköisyys*

Datan koko (px)	**Opetusaineisto 300 px (%)**	**Opetusaineisto 540 px (%)**	**Opetusaineisto 1 200 px (%)**	**Opetusaineisto 3 350 px (%)**
1 850	63 %	65 %	67 %	67 %
4 200	57 %	58 %	59 %	60 %
7 500	56 %	58 %	59 %	59 %
30 000	54 %	56 %	56 %	58 %
67 500	54 %	55 %	56 %	56 %
187 500	53 %	55 %	55 %	56 %
422 250	59 %	61 %	61 %	62 %
750 000	53 %	54 %	55 %	56 %

6.2 Opettaminen ja ihmisriippuvaisuus

Monen hahmontunnistusalgoritmin toiminta on riippuvainen sen opettamisesta. Opettamisella tarkoitetaan testiaineiston syöttöä algoritmille, jonka pohjalta se osaa tulevaisuudessa luokitella dataa. Monikaan algoritmi ei siis osaa tehdä älykkäitä, omia



Kuva 6.4 *Huonosti opetetun tunnistusalgoritmin tulos*

johtopäätöksiä kuten ihminen, vaan ne vain luokittelevat aineiston tuttuihin ja tuntemattomiin piirteisiin. Tällaisten algoritmien toiminta käytännönsovelluksissa riippuu paljon siitä, miten hyvin se on opetettu. Tämän projektin algoritmikaan ei tee tästä poikkeusta, vaan järjestelmä olettaa sen opettavan henkilön tietävän mitä tekee. Kuva 6.4 esittää tunnistusalgoritmin tulosta silloin, kun se on huonosti opetettu. Tulos ei vastaa milläänlailla lähtöaineistoa ja tässä tapauksessa kuusentaimi jäisi tunnistamatta.

Algoritmin riippuvuus sen opettajasta voidaan käsittää yhdeksi sen suurimmista heikkouksista. Koska metsämaaston värimaailma vaihtelee voimakkaasti riippuen paikan valoisuudesta ja maaperästä, tulisi tunnistualgoritmin olla "juosten opetettavissa", eli käyttäjän tulisi toimia takaisinkytkentänä algoritmille hyväksyen tai hyläten algoritmin ehdotuksia tunnistetuista kuusentaimista. Algoritmi pystyisi käyttämään käyttäjän hyväksyntää itsensä jatkuvaan opettamiseen, jolloin se mukautuisi nopeammin muuttuviin maasto-olosuhteisiin. Tämä kuitenkin vaatii erillisen käyttöliittymän suunnittelua siten, että metsäkoneen kuljettaja pystyy työnsä ohella helposti hallinnoimaan myös tunnistusalgoritmia. Työn sujuvuuden kannalta on tärkeää, että kuljettaja keskittää mahdollisimman paljon huomiostaan itse työn te-

kemiseen koneen hallintajärjestelmien säätämisen sijaan. Tällöin jokainen ylimääräinen vilkaisu esimerkiksi koneen näyttöön kasvattaa kuljettajan henkistä työtaakkaa, jonka seurauksena epävarmatoiminen tunnistusalgoritmi voi helposti jopa hidastaa työn etenemistä.

6.3 Vuodenaikojen vaikutus algoritmin toimintaan

Pohjoismainen metsänhoito keskittyy pitkälti yleisimpien havupuiden, kuusen ja männyn ympärille. Vaikka tunnistusalgoritmia voidaan yrittää soveltaa lähes minkä tahansa ominaisuuden löytämiseen metsämaastosta, niin sen tärkeimpänä tehtävänä voidaan pitää havun tunnistamista muusta ympäristöstään. Eri vuodenajat muuttavat tätä tehtävää huomattavasti, sillä lehtipuut tiputtavat lehtensä talven ajaksi, mutta havupuut eivät pudota havujaan. Täten lehdellisten vuodenaikojen voidaan olettaa olevan algoritmille haasteellisinta aikaa. Lehdettömänä vuodenaikana suurin osa metsämaaston vihreistä sävyistä koostuu havupuista ja sammaleesta, jolloin tunnistaminen on helpompaa. Talvella maan peittävä lumi voi joko helpottaa algoritmia peittäen maaston vihreänsävyt lähes täysin, mutta se voi myös haitata algoritmia peittäen myös osan havupuiden oksista, jolloin tunnistusvaiheessa ei voida luottaa siihen, että yksittäinen puu tunnistettaisiin yhtenä suurena vihreänsävyn alueena. Luminen maasto aiheuttaa koneelliselle taimikonhoidolle haasteita aina, sillä koneet ovat pieniä ja ne joutuvat kulkemaan kokoonsa nähden suuressa lumihangessa. Koska talvi ei tästä syystä ole keskeinen vuodenaika koneellisessa taimikonhoidossa, voidaan se jättää vähemmälle tarkastelulle. Koneellisen taimikonhoidon vuosittainen työaika jakautuu sulanmaan ajalle, josta arviolta puolet ajasta on lehdellistä ja puolet lehdetöntä aikaa.

Algoritmia testattiin lehdettömänä vuodenaikana otetulla kuvalla samalla tapaa kuin lehdelliseen vuodenaikaan otetulla kuvalla aiemmassa luvussa. Lehdettömään aikaan otetun kuvan tunnistuksen todennäköisyydet on esitetty taulukossa 6.4. Kun sitä vertaa taulukkoon 6.3, tunnistaa algoritmi yksittäiset pikselit 30 prosenttiyksikköä todennäköisemmin lehdettömänä vuodenaikana. Ero on ymmärrettävä, sillä lehdettömänä aikana ympäröivä maasto on pääpiirteissään ruskeaa, kun taas lehdellisenä aikana pääväri on vihreä.

Lisäksi algoritmi tunnisti pikselit oikein hieman todennäköisemmin mitä pienempi kuva oli ja mitä enemmän opetusaineistoa käytettiin. Tämä voidaan kuitenkin selittää sillä, että malliesimerkistä erotettiin kuusi käsin, jolloin tulos ei ole ideaalinen ja

virhe kasvaa kuvan resoluution mukana. Pieniresoluutioisissa kuvissa malliesimerkki oli tarkempi ja tämän vuoksi tuloksissa näkyy todennökäisyyden lasku resoluution kasvaessa.

Taulukko 6.4 *Algoritmin pikselikohtainen tunnistuksen todennäköisyys lehdettömässä testikuvassa*

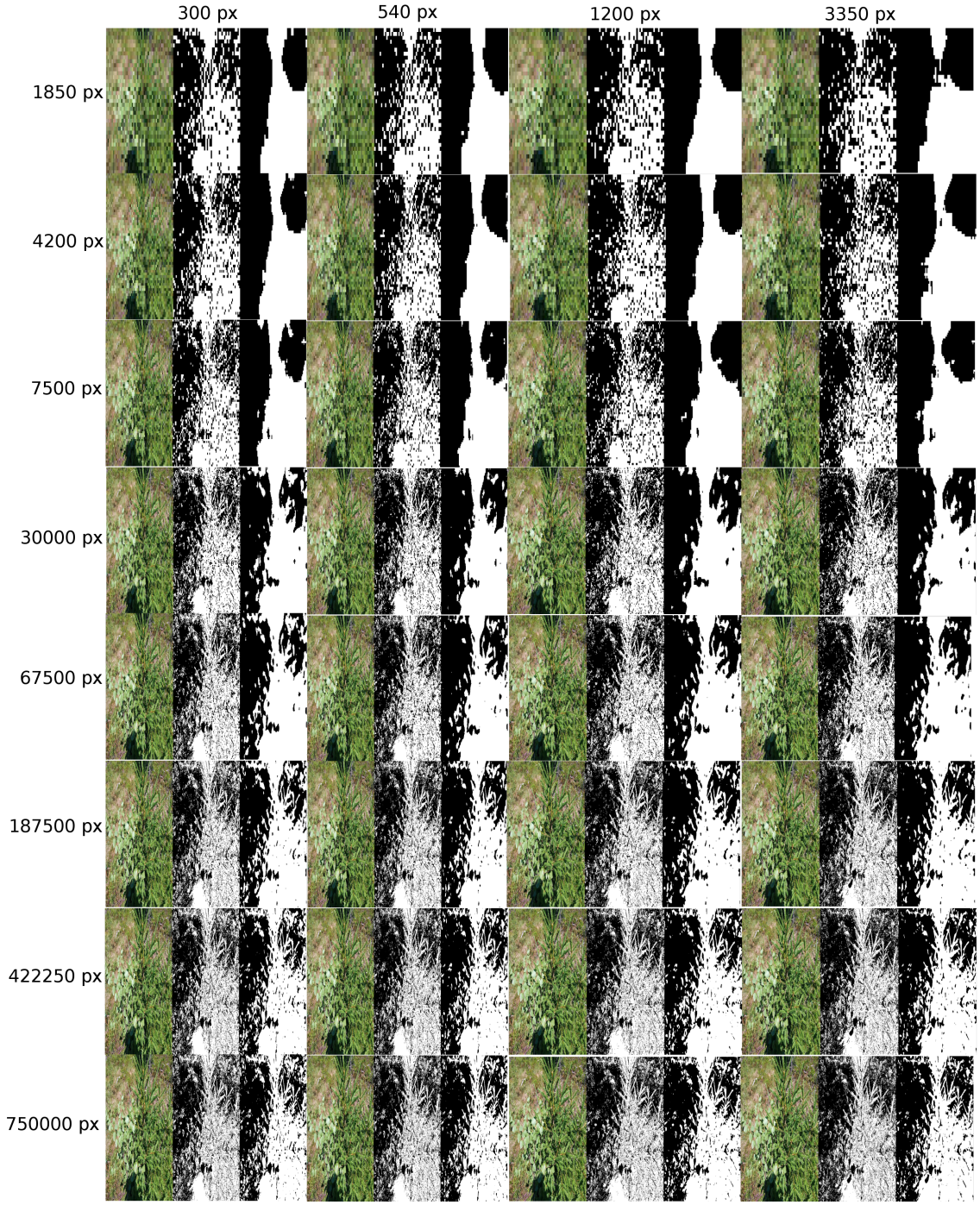
Datan koko (px)	**Opetusaineisto 300 px (%)**	**Opetusaineisto 540 px (%)**	**Opetusaineisto 1 200 px (%)**	**Opetusaineisto 3 350 px (%)**
1 850	80 %	80 %	81 %	82 %
4 200	78 %	78 %	79 %	80 %
7 500	77 %	78 %	79 %	80 %
30 000	76 %	76 %	78 %	79 %
67 500	74 %	74 %	76 %	78 %
187 500	73 %	73 %	76 %	77 %
422 250	72 %	72 %	75 %	77 %
750 000	72 %	71 %	74 %	76 %

Esimerkki taulukon 6.4 tuloksista on esitetty kuvassa 6.5. Vasemmalla on alkuperäinen kuva, keskellä on algoritmin ulostulo ja oikealla on malliratkaisu. Algoritmi on numeerisesti tunnistanut 70-80 % pikseleistä oikein, joka voidaan käsittää erittäin hyväksi tulokseksi. Tässä tapauksessa kuusentaimen tunnistaminen kuvasta onnistuu vaivattomasti.

Kuva 6.5 *Lehdettömän ajan testikuva ja algoritmin tulos verrattuna mallitulokseen*

Kynnyskysymys tunnistusalgoritmille onkin sen toiminta lehdellisenä vuodenaikana. On kyseenalaista, millä todennäköisyydellä järjestelmä pystyisi tunnistamaan taimet esimerkiksi kesäisestä horsmikosta, kuten kuvassa 6.6. Tämä haaste ei päde vain väripohjaisiin algoritmeihin, vaan myös tekstuuripohjaisiin algoritmeihin. Kuvan 6.6 horsmikko on tekstuuriltaan hyvin samankaltaista kuin kuvan etuosalla sijaitseva kuusentaimi. Jopa ihmissilmällä on vaikeuksia rajata taimi erilleen taustasta. Metsämaasto vaihtelee paljon riippuen maastosta ja maan ravinteista. Suinkaan kaikkialla ei maasto kasva horsmaa ja muuta kasvistoa tiheästi, joten algoritmi voi silti olla toimintakelpoinen kesäaikana kohteesta riippuen.

Kuva 6.6 *Kesäinen maasto josta taimen tunnistaminen on lähes mahdotonta*



Kuva 6.7 *Taulukon 6.2 arvoja vastaavat tunnistuskuvat*

7. MENETELMÄN SOVELTAMINEN KÄYTÄNNÖSSÄ

Yksi käytännönläheisen ja toimivan algoritmin haastavimmista vaatimuksista on järjestelmän toimintavarmuus, eli robustisuus. Kuusentaimen tunnistusalgoritmia hyödyntävää testilaitetta ei tässä työssä toteutettu. Tässä luvussa on silti määritelty prototyypiltä vaadittuja vaatimuksia, mitkä tulisi ottaa huomioon prototyyppiä kehittäessä.

7.1 Toteutus sulautettuna järjestelmänä

Mekaaninen kestävyys on lähes itsestäänselvyys toimivalle järjestelmälle. Tämän vuoksi jo kuusentaimentunnistusalgoritmin suunnitteluvaiheessa päätettiin tehdä algoritmista sellainen, että sitä pystytään käyttämään sulautetussa järjestelmässä. Sulautettu järjestelmä on elektroninen laite, joka suorittaa vain yhtä tai muutamaa sille ennalta määriteltyä tehtävää [14] ja juuri se on suurin yksittäinen tekijä joka eroittaa sulautetut järjestelmät tietokoneista. Tietokone itsessään on lähes hyödytön laite, sillä se vaatii erilaisten ohjelmistojen käyttämisen, jotta se pystyisi antamaan käyttäjälleen lisäarvoa. Tietokoneen tehtävä onkin tarjota käyttöalusta käyttäjän määrittämille ohjelmistoilla. Sulautettu järjestelmä taas toimii sellaisenaan, riippumatta kolmannen osapuolen tarjoamista palveluista tai ohjelmistoista.

Sulautetut järjestelmät sisältävät usein yhden tai useamman mikrokontrollerin. Mikrokontrollerit ovat kansankielisesti *minitietokoneita*. Mikrokontrolleri on yhden komponentin tietokone, joka liitetään osaksi piirilevyä. Kuva 7.1 havainnollistaa pientä mikrokontrolleria. Se sisältää kaikki samat peruskomponentit kuin nykyajan tietokoneetkin, kuten esimerkiksi prosessorin ja muistin, mutta ne ovat luonnollisesti huomattavasti pienemmät ja heikkotehoisemmat kuin tietokoneissa. Mikrokontrollereiden hinnat ovat muutaman euron luokkaa, kun taas tietokoneet, tai edes mikroprosessorit, ovat huomattavasti kalliimpia.

Kuva 7.1 *Mikrokontrolleri, IC-komponentti [2]*

Kamera on luonnollisesti tärkeä toimivan laitteen kannalta. Tämän projektin kannalta resoluutio ei ole avainasemassa, sillä nykyään markkinoilta on saatavissa hyvinkin edullisesti, ja kuten aiemmassa luvussa todettiin, k-NN-algoritmin suoritusaika on paljon riippuvainen kuvien resoluutiosta. Suuriresoluutioista kameraa ei siis todennäköisesti edes käytettäisi maksimiresoluutiolla. Toisaalta kameraan kohdistuu muita sovelluskohtaisia vaatimuksia, joita ovat esimerkiksi:

- Valkotasapaino
- Kameran näkökenttä
- Pölyn, veden ja tärinän sietoisuus
- Kuvanottonopeus

Automaattinen valkotasapainon säätö on tärkeää, sillä kuusentaimentunnistusalgoritmi tukeutuu vain vihreänsävyyn kuusen tunnistuksessa. Suurin osa markkinoilta saatavista kameroista säätävät valkotasapainonsa itse automaattisesti, mutta erityisesti matalan tason kamerakomponentit eivät tätä välttämättä tee. Kameran näkökenttä on riippuvainen sovelluksesta, ja esimerkiksi pienmetsäkoneen työlaitteen kanssa toimiessa kameran näkökenttä tulee olla laaja, jotta se pystyy havaitsemaan

mahdollisimman kattavasti koko työlaitteen ympäristön. Pölyn, veden ja tärinän sietoisuus puolestaan on luonnollista toimiessa ulkoilmassa. Kameran kuvanottonopeus on tärkeää, sillä kameran on pystyttävä ottamaan riittävän tarkkoja kuvia metsäkoneen liikkuessa.

7.1.1 Rajoitetun laskentatehon asettamat vaatimukset

Sulautettujen järjestelmien mittakaavassa jo pelkästään kuvien ottaminen ja prosessoiminen on vaativa tehtävä. Esimerkiksi NXP:n mikrokontrolleri LPC1785 sisältää RAM-muistia vain 80 kilotavua, kun taas yksittäinen valokuva sisältää pikselidataa useita megatavuja [11]. On siis selvää, että kuvankäsittely vaatii erikseen sitä varten valmistetun laitteen, josta löytyy tarpeeksi muistia, sekä laskentatehoa. Perinteinen prosessori, jota suurin osa mikrokontrollereista käyttää, on suunniteltu laskemaan kokonaislukulaskuja. Kuvankäsittelyssä kuitenkin usein tulee tilanteita, joissa olisi hyödyllistä pystyä suoritamaan laskutoimitus suoraan liukulukulaskuna, joka on perinteiselle prosessorille laskentaintensiivinen toimenpide. Tämän vuoksi olisi hyödyllistä käyttää erillistä DSP:tä, eli digitaalista signaaliprosessoria, joka on liukulukulaskentaan suunniteltu prosessorityyppi. DSP:llä liukulukulaskenta on tehokasta ja kameran datan reaaliaikainen prosessointi on mahdollista, mutta jokatapauksessa haastavaa reaaliaikaisuuden takia.

Joissain teollisissa kameroissa on sisäinen DSP-prosessori, sekä sen kanssa FPGA-piiri, eli ohjelmoitavaa logiikkaa [22]. Samalla tavalla kuin tietokoneen tai mikrokontrollerin prosessorille pystyy kirjoittamaan ohjelman, jonka prosessori suorittaa, pystyy FPGA:lle ohjelmoimaan käytännössä mitä tahansa logiikkaa. FPGA:ta voi käyttää tehokkaana datan esikäsittelijänä, sillä kun perinteinen prosessori on hyvä suorittamaan monimutkaisia toimenpiteitä yksi kerrallaan, on FPGA hyvä toteuttamaan suhteellisen yksinkertaisia tehtäviä useita samanaikaisesti. Jos teollisuuskamerassa on FPGA-esiaste, voidaan sitä käyttää kuvadatan esikäsittelemiseen ennenkuin data annetaan prosessorin käsiteltäväksi. Esimerkiksi koko k-NN-algoritmi on mahdollista antaa FPGA:n tehtäväksi, jolloin useampi pikseli pystytään käsittelemään rinnakkain, säästäen prosessorin laskentatehoa varsinaista tuloksen päättelemistä varten.

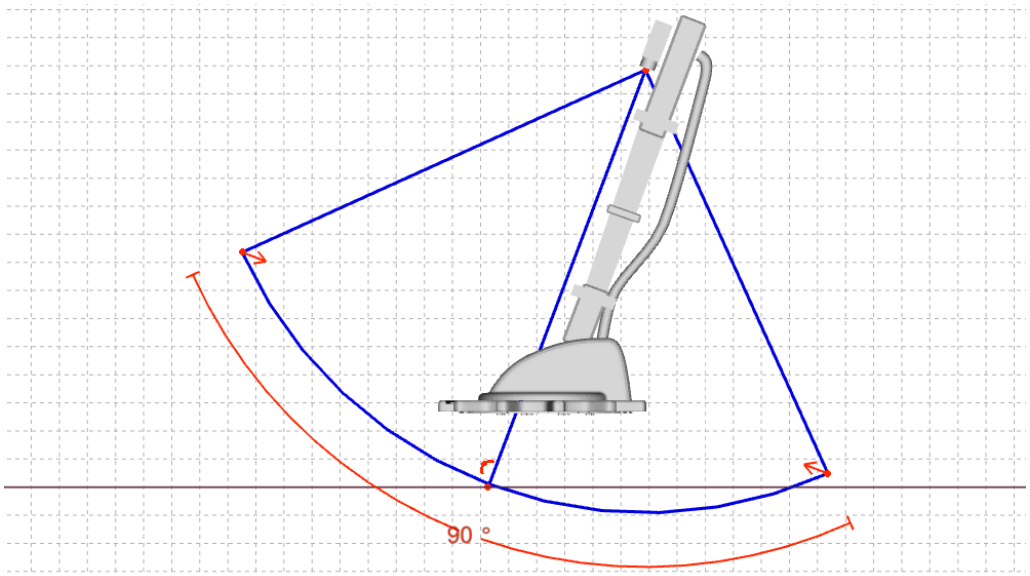
7.2 Sovelluskeskeiset muutokset algoritmiin

Työkoneen liikkeiden hidastaminen työskennellessä taimen lähettyvillä on yksinkertainen sovellus, joka ei välttämättä vaadi minkäänlaista kohteen seurantaa. Pelkkä kuusentaimen tunnistaminen yksittäisistä valokuvista voi hyvinkin riittää, sillä järjestelmää ei varsinaisesti kiinnosta, missä taimi on. Kuitenkin järjestelmän kehittäminen yhtään pidemmälle tulisi vaatimaan kohteen seurantaa, edes yksinkertaisessa muodossaan. Esimerkiksi niin, että sovellus pystyy tunnistamaan missä suunnassa taimi on suhteessa raivuriin ja raivurin liikesuuntaan. Tällöin järjestelmä pystyisi hidastamaan raivurin liikkeita vain silloin, kun kuljettaja on liikuttamassa raivuria tainta kohti. Liikuttaessa taimesta poispäin liikkeitä ei puolestaan tarvitsisi hidastaa. Tällaisen kohteenseurannan toteuttaminen tässä työssä esitellyn algoritmin päälle vaatisi hieman aikaa, mutta sulautetun järjestelmän laskentateho todennäköisesti riittäisi oikein hyvin, kun kehitetään algoritmi vain ja ainoastaan sovelluksen vaatimalla tarkkuudella. Tässä tapauksessa järjestelmän ei tarvitse tietää tarkkaa kuusen sijaintia, eikä edes tarkkaa etäisyyttä, joten riittäisi, jos taimen sijainti pystyttäisiin ilmoittamaan sektoreina raivurin keskipisteestä, esimerkiksi ilmoittaen taimen sijaitsevan raivurin vasemmalla puolella. Ohjausjärjestelmälle tämä on riittävä tieto. Jos kamera on asennettu siten, ettei se näe muualle kuin raivurin lähistöön, ei ohjausjärjestelmän tarvitse tietää edes taimen etäisyyttä.

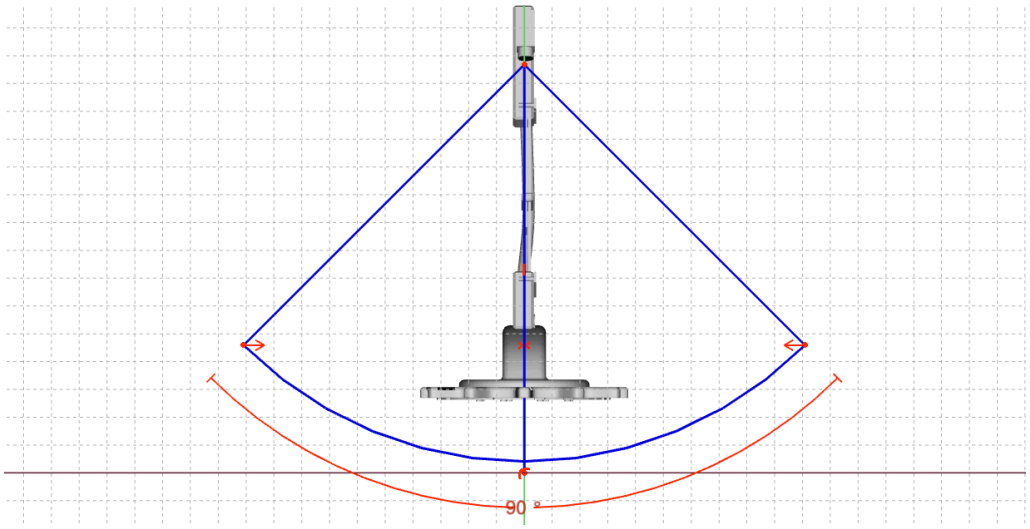
7.2.1 Mekaaninen asennus

Työskennellessä koneellisen raivauksen työlaitetta liikutetaan maan pinnan lähellä vaakatasossa, tavoitteena katkoa suurin osa vesakosta, mutta jättäen kuusentaimet vahingoittumattomina kasvamaan. Jotta kuusentaimet pystyttäisiin tunnistamaan raivurin lähistöltä, on sulautettu kamera asetettava siten, että se näkee suurimman osan raivurin välittömästä ympäristöstä, mutta mahdollisimman vähän taustametsää. Lisäksi kameran pitää olla suojattu siten, ettei katkaistut puut revi sitä rikki. Kameraa ei siis voida sijoittaa vapaasti esimerkiksi suoraan raivurin yläpuolelle pitkän varren päähän. Parhaimman mekaanisen suojan kameralle antaa itse metsäkoneen puomi ja puomiin on myös helppo lisätä erilaisia suojarakenteita kameran ympärille. Kuvat 7.2 ja 7.3 havainnollistavat kameran sijoitusta UW50-risuraivaimen [17] jatkopuomiin siten, että se näkee mahdollisimman laajalti raivurin ympärille, mutta ei horisonttiin. 90 astetta on riittävä kuvakulma, jolla kamera näkee tarpeeksi ympärilleen. Asennettaessa kamera puomiin kiinni on mahdotonta saavuttaa täy-

dellistä näkökenttää raivurin joka puolelle. Puomi aiheuttaa aina varjostuksen jollekin sektorille näkökentästä. Asennettaessa kamera puomin päälle sijoittuu varjostus symmetrisesti raivurin taakse, joka voidaan hyväksyä, sillä raivaustyöskentely on usein sivuttaissuuntaista ja raivurin takaosaa käytetään vain harvoin työskentelyyn.



Kuva 7.2 *Sulautetun kameran asennus UW50-risuraivaimeen sivulta*



Kuva 7.3 *Sulautetun kameran asennus UW50-risuraivaimeen edestä*

7.2.2 Tietojärjestelmän sovitus

Työkoneiden ohjausjärjestelmien de-facto standardiksi on muodostunut CAN-väylä [9]. Myös suurimmassa osassa metsäkoneissa runkokoneen ja työlaitteen kommunikaatio on toteutettu CAN-väylän avulla. CAN-väylä olisikin siis hyvä valinta rajapinnaksi sulautetun kameran kommunikointiin pienmetsäkoneen ohjausjärjestelmän kanssa. CAN-väylä ei kuitenkaan ole tarkoitettu suuren datamäärän siirtämiseksi, eikä sen kapasiteetti riitä esimerkiksi kuvadatan lähettämiseksi järkevässä aikaikkunassa. Algoritmin opettaminen on haastavaa, jollei käyttäjä näe kameran ottamia kuvia. Siksi voi olla tarpeellista käyttää joko USB tai Ethernet väyliä kuvadatan siirtämiseksi näytölle, josta kuljettaja opettaa algoritmin. CAN-väylä on kuitenkin toimintavarma ratkaisu prosessidatan lähettämiseksi koneen ohjausjärjestelmälle [9]. Tämä prosessidata tulisi sisältämään vain tiedon havainnoiduista taimista ja niiden sijainnista, mutta ei varsinaista kuvadataa. Myös kameran ohjausviestit, kuten tunnistetun taimen hyväksyntä tai hylkääminen, voidaan lähettää CAN-väylän kautta, jolloin kuljettajan vaikutus algoritmin opettamiseen on helppo liittää osaksi metsäkoneen omaa ohjausjärjestelmää.

8. POHDINTA

Kuusentaimentunnistusalgoritmin kehittämisen yhteydessä pohditut ongelmat reaaliaikaisen taimentunnistusalgoritmin kehittämisestä pystyttiin työn laajuuden mittakaavassa ratkaisemaan. Algoritmi saatiin toimimaan käytössä olleella testiaineistolla, mutta koska sulautettua kamerasovellusta ei kehitetty, jäi algoritmin varsinainen testikäyttö vähäiseksi.

Testiaineiston perusteella algoritmi toimi hyvin lehdettömänä aikana, saaden yli 70 % tunnistettavan kuvan pikseleistä luokiteltua oikein, sekä melko hyvin lehdellisenä vuodenaikana, tunnistaen 55- 67 % pikseleistä oikein. Tulevaisuudessa tulisikin jatkotestata algoritmia erilaisissa maastoissa, eri vuodenaikoihin. Näin saataisiin kattavempi aineisto tilanteista, missä väripohjainen tunnistus ei riitä kuusentaimen havaitsemiseksi ympäröivästä maastosta. On todennäköistä, että väripohjainen algoritmi ei riitä taimen tunnistamiseen keskikesällä rehevissä maastoissa, sillä kuusentaimen tunnistaminen horsmikon joukosta on ihmissilmällekin haastava tehtävä. Mutta vaikka algoritmi ei kaikissa olosuhteissa toimisikaan, voi se valikoiduissa kohteissa olla hyvinkin varmatoiminen ja käyttökelpoinen.

Tässä työssä esitettiin yksi mahdollinen käyttökohde kuusentaimentunnistusalgoritmille pienmetsäkoneessa, tunnistaen taimia työlaitteen lähistöltä ja avustaen kuljettajaa esimerkiksi hidastaen puomin liikkeitä operoitaessa lähellä taimia. Tällaisessa sovelluksessa kameran asentaminen työlaitteeseen on olennaisessa osassa järjestelmän toimivuuden kannalta. Kameran fyysinen kestävyys saattaa koitua ongelmaksi, sillä kamera olisi asennettava lähelle työlaitetta hyvän näkyvyyden saamiseksi. Mitä lähempänä kamera työlaitetta sijaitsee, sitä enemmän se on alttiina fyysiselle rasitukselle. Kameran asentamisessa olisi myös pidettävä huoli ettei kameran näkökenttä ulotu liian kauas työlaitteesta. Tällöin taimentunnistusta saattaisi häiritä esimerkiksi taustametsä ja sovellus saattaisi vaatia ylimääräistä ohjelmistoa taustan aiheuttamien häiriöiden poistamiseksi.

Muita käyttökohteita algoritmille olisi esimerkiksi kitkennässä samantyylinen sovel-

lus, mutta havaiten maastoa ylhäältä päin ja ohjaten kitkiää niin, ettei se laskeudu kuusentaimen päälle. Myös metsänhoitajien työssä algoritmi voisi olla hyödyllinen taimikon tiheyden laskennassa. Nykyisin puuston tiheys mitataan ottamalla pyöreitä koealoja taimikosta, joka on hidasta ja virheherkkää käsityötä. Käyttäen taimentunistusalgoritmia voitaisiin esimerkiksi pienoiskopterilla ottaa valokuva taimikon yläpuolelta ja tunnistaa tuosta kuvasta kuusentaimet. Tällöin taimien tunnistus ei myöskään tarvitsisi olla reaaliaikaista, jolloin voitaisiin käyttää monimutkaisempia ja tarkempia algoritmeja laskentavirheiden minimoimiseksi. Tekstuuripohjaisilla algoritmeilla, sekä infrapuna- tai ultraviolettitaajuuksia hyödyntämällä voitaisiin saada eroteltua kuusentaimet jopa keskikesällä horsmien seasta.

9. YHTEENVETO

Projektin aikana saatiin kehitettyä toimiva kuusentaimentunnistusalgoritmi, joka pystyi tunnistamaan kuusentaimen sen havujen värin perusteella suomalaisesta metsämaastosta otetusta valokuvasta. Tunnistuksessa ei tarvinnut käyttää tekstuuripohjaisia tunnistusalgoritmeja, vaan se perustui täysin värien tunnistamiseen. Algoritmi pyrki luokittelemaan sille aiemmin opetetun aineiston perusteella kuvan pikselit osaksi kuusentainta. Algoritmi käytti k-NN, eli k-lähin naapuri -algoritmia, joka on yksinkertainen ja nopea algoritmi datan luokitteluun. Algoritmin suoritusnopeus oli pöytätietokoneella suoritettuna hyvä. Vaadittava resoluutio on sovelluskohtaista, mutta koneellisen taimikonhoidon oletettuihin käyttökohteisiin riittävällä resoluutiolla algoritmin suoritusajat olivat muutamia kymmeniä millisekunteja. Koska suoritusnopeus oli voimakkaasti riippuvainen opetusmateriaalin ja prosessoitavan kuvan resoluutioista, todettiin resoluutioiden olevan syytä pitää madollisimman pieninä riippuen sovelluksen vaatimuksista. Kuuset tunnistettiin käyttäen k:n arvoa 3. Eri k:n arvoilla väliltä 1-9 todettiin olevan vain vähän vaikutusta algoritmin suorituskykyyn, opetusaineiston laadun taas todettiin olevan ratkaiseva tekiä algoritmin toimminnalle. Jos algoritmia ei opetettu huolella, ei se myöskään pystynyt tunnistamaan kuusentainta maastosta. Koska tunnistettava havupuun vihreän sävy on hyvin tarkka, tuli algoritmin opettamisen painopiste olla keskitetty kuusentaimiin kuulumattomiin värisävyihin. Keskikesällä otetutut nuoren kuusitaimikon valokuvat osoittautuivat haastaviksi algoritmille, sillä keskikesällä horsmat ovat pitkiä ja hyvin samansävyisiä kuin kuusentaimet. Lehdettömänä aikana otetuista kuvista taimen tunnistus osoittautui algoritmille helpoksi. Talvisia metsäkuvia ei algoritmia kehittäessä tutkittu, sillä talvi on koneellisen taimikonhoidon kannalta epäedullinen vuodenaika.

Tutkimuskysymyksenä esitettiin, että onko mahdollista kehittää konenäköalgoritmi, joka pystyy tunnistamaan kuusentaimen lehdellisenä, sekä lehdettömänä aikana ja että onko se riittävän toimintavarma käytönnössä sovellettavaksi. Tämän tutkimuksen perusteella algoritmi on mahdollista kehittää toimivaksi lehdellisenä, sekä

lehdettömänä vuodenaikana. Lumista vuodenaikaa ei työssä käsitelty lainkaan, sillä se ei ole koneellisen taimikonhoidon kannalta oleellinen vuodenaika. Algoritmin toimintaa ei työn laajuuden takia testattu käytännönsovelluksissa. Jatkotutkimusta tarvitaan todentamaan algoritmin toimintavarmuus todellisessa metsäympäristössä osana metsäkoneen ohjausjärjestelmää. Myös koneen kuljettajan ja algoritmin välinen vuorovaikutus työn edetessä olisi tutkittava, jotta saataisiin tietoa, kokeeko ihminen koneen avustamat toiminnot hyödyllisiksi vai epämieluisiksi.

KIRJALLISUUTTA

[1] CMake, "Cmake," [online], https://cmake.org/, 2016.

[2] Conrad, "Microcontroller," [online], http://www.conrad.com/medias/global/ce/1000_1999/1600/1650/1652/165258_RB_00_FB.EPS_1000.jpg, 2016.

[3] cplusplus.com, "C++," [online], http://www.cplusplus.com/doc/tutorial/, 2016.

[4] V. Design, "The fundamentals of color: Hue, saturation, and lightness," [online], http://vanseodesign.com/web-design/hue-saturation-and-lightness/, 2016.

[5] gtkmm, "gtkmm," [online], http://www.gtkmm.org/en/, 2016.

[6] D. F. S. Gustavo E.A.P.A. Batista, "How k-nearest neighbor parameters affect its performance," [online], http://www.labic.icmc.usp.br/pub/gbatista/BatistaASAI09.pdf, 2016.

[7] A. V. Heikki Hyyti, Jouko Kalmari, "Real-time detection of young spruce using color and texture features on an autonomous forest machine," *Aalto yliopisto, Automaatio- ja systeemitekniikan laitos*, 2013.

[8] Hyperphysics, "Electromagnetic spectrum," [online], http://hyperphysics.phy-astr.gsu.edu/hbase/ems3.html, 2016.

[9] C. in Automation (CiA), "History of can technology," [online], https://www.can-cia.org/can-knowledge/can/can-history/, 2017.

[10] J. Matilainen, "Metsänhoitaja joona matilaisen opinnäytetyön julkaisematon ennakkotieto," 2014.

[11] NXP, "Lpc1785 microcontroller," [online], http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/lpc-mcus/lpc1700-cortex-m3/scalable-mainstream-32-bit-microcontroller-mcu-based-on-arm-cortex-m3-core:LPC1785FBD208, 2016.

[12] OpenCV, "Opencv - open source computer vision," [online], http://opencv.org/, 2016.

[13] ——, "Understanding k-nearest neighbour," [online], http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_knn/py_knn_understanding/py_knn_understanding.html, 2016.

[14] E. TEC, "Embedded systems," [online], http://www.tik.ee.ethz.ch/education/lectures/ES/, 2016.

[15] K. Uotila, "Koneellinen taimikonhoito," METLA presentation, [online], http://www.metla.fi/metinfo/metsanhoitopalvelut/pdf/Koneellinen_taimikonhoito_KUot4.pdf, 2014.

[16] ——, "Taimikonhoito ajoissa kannattaa," *Savotta*, 2014.

[17] Usewood, "Uw50-risuraivain," [online], http://usewood.fi/index.php/fi/taimikonhoitolaitteet/uw50, 2016.

[18] P. A. Visa, "Suullinen haastattelu," 2016.

[19] Wikipedia, "Euclidian distance," [online], https://en.wikipedia.org/wiki/Euclidean_distance, 2016.

[20] ——, "Gamut," [online], https://en.wikipedia.org/wiki/Gamut, 2016.

[21] R. World, "Understanding color," [online], https://www.rgbworld.com/color.html, 2016.

[22] Xilinx, "Wat is an fpga," [online], https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm, 2017.

APPENDIX A. TESTIOHJELMAN LÄHDEKOODI

ufilepaths.h

```
/****************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy. For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
****************************************************************************/


#ifndef UFILEPATHS_H
#define UFILEPATHS_H


#include <string>

class UFilePaths
{
public:
        UFilePaths();

        static std::string getResourcePath();
};

#endif // UFILEPATHS_H
```

ugraphicwidget.h

```
/****************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy. For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
****************************************************************************/


#ifndef UGRAPHICWIDGET_H
#define UGRAPHICWIDGET_H

#include <gtkmm/drawingarea.h>
#include <opencv2/imgproc.hpp>


class UGraphicWidget : public Gtk::DrawingArea
{
public:
        UGraphicWidget();

        void draw(cv::Mat &img);
```

```
private:
        cv::Mat sourceImg;

        virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
};

#endif // UGRAPHICWIDGET_H
```

mainwindow.hh

```
/****************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy.  For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
****************************************************************************/


#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/builder.h>
#include <vector>
#include <opencv2/videoio.hpp>
#include "ugraphicwidget.h"
#include <opencv2/opencv.hpp>
#include <opencv2/ml.hpp>


namespace Gtk {
class Window;
class Image;
class SpinButton;
class ToggleButton;
class FileChooser;
}
namespace cv {
namespace ml {
class KNearest;
}
}

namespace Usevolt {
}

typedef sigc::connection utimer;


/// @brief: Main window class
class MainWindow
{
public:
        MainWindow();
        virtual ~MainWindow();


        Gtk::Window* getMainWindow();


protected:

private:
```

```
        void startButtonToggled();
        void imgChooserFileActivated();
        void trainkNN(cv::Ptr<cv::ml::KNearest> nearest);

        /// @brief: Processes the given data the way that the data
        /// is ready to be given to other algorithms. The returned data
        /// is one channel, CV_32F type data.
        void processColorSpace(cv::Mat &data, cv::Mat &dest);

        void drawImage(int imgIndex, cv::Mat data, int width, int height);

        void separateChannelsToCols(cv::Mat &src, cv::Mat &dest);
        void separateChannelsToRows(cv::Mat &src, cv::Mat &dest);

        bool step(int timerID);

        cv::VideoCapture camera;

        utimer timer;

        struct ui_st {
        Glib::RefPtr<Gtk::Builder> builder;
        Gtk::Window* window;
        Gtk::ToggleButton *startButton;
        Gtk::SpinButton *camChooser;
        Gtk::SpinButton *stepTimeChooser;
        Gtk::Image *img[3];
        Gtk::FileChooser *imgChooser;
        Gtk::FileChooser *matchingChooser;
        Gtk::FileChooser *nonMatchingChooser;
        };
        ui_st ui;


};

#endif // MAINWINDOW_H
```

main.cc

```
#include <iostream>
#include <gtkmm/application.h>
#include <inc/mainwindow.hh>

int main(int argc, char* argv[])
{
        Glib::RefPtr<Gtk::Application> app =
        Gtk::Application::create(argc, argv, "org.usevolt.gtkmm.example");

        MainWindow w;
        return app->run(*w.getMainWindow());
}
```

ufilepaths.cc

```
/*************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy.  For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
*************************************************************************/
```

```
#include <inc/ufilepaths.h>

UFilePaths::UFilePaths()
{

}
std::string UFilePaths::getResourcePath()
{
        return std::string("./resources/");
}
```

ugraphicswidget.cc

```
/****************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy.  For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
****************************************************************************/


#include <inc/ugraphicwidget.h>
#include <iostream>
#include <gdkmm.h>


UGraphicWidget::UGraphicWidget() : sourceImg()
{
}

void UGraphicWidget::draw(cv::Mat &img) {
        sourceImg = img;
        this->queue_draw();
}

bool UGraphicWidget::on_draw(const Cairo::RefPtr<Cairo::Context> &cr) {

        if (sourceImg.empty()) {
                return true;
        }

        std::cout << "drawing " << sourceImg.rows << " rows and "
        << sourceImg.cols << "cols" << std::endl;

        sourceImg.convertTo(sourceImg, CV_8U);
        Glib::RefPtr<Gdk::Pixbuf> pix = Gdk::Pixbuf::create_from_data(
                        sourceImg.data, Gdk::COLORSPACE_RGB, false, 8,
                        sourceImg.cols, sourceImg.rows, sourceImg.step);


        Gdk::Cairo::set_source_pixbuf(cr,
                        pix->scale_simple(this->get_width(), this->get_height(),
            Gdk::INTERP_NEAREST));

        cr->rectangle(0, 0, pix->get_width(), pix->get_height());

        cr->fill();

        return true;
}
```

mainwindow.cc

```
/****************************************************************************
**
** Copyright (C) 2014 Usevolt Oy
**
** Commercial License Usage
** Licensees holding valid commercial Usevolt Oy licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and Usevolt Oy.  For licensing terms and
** conditions contact Usevolt Oy.
**
**
** $QT_END_LICENSE$
**
****************************************************************************/

#include <iostream>
#include <gtkmm/builder.h>
#include <glibmm/fileutils.h>
#include <gtkmm/window.h>
#include <gtkmm/frame.h>
#include <gtkmm/spinbutton.h>
#include <gtkmm/togglebutton.h>
#include <gtkmm/messagedialog.h>
#include <gtkmm/filechooser.h>
#include <glibmm/main.h>
#include <boost/filesystem.hpp>
#include <inc/mainwindow.hh>
#include <inc/ufilepaths.h>
#include <fstream>
#include <string>
#include <vector>
#include <opencv2/opencv.hpp>
#include <opencv2/ml.hpp>
#include <chrono>


using namespace Usevolt;

/// @brief: A little bit easier syntax to connect gtkmm signals to member functions
/// @param src: The signal function call of the gtkmm emitter
/// @param ptr: A pointer to the object which will receive the signal callback
/// @param dest: Function name (without argument parentheses) of the callback
#define signal_connect(src, ptr, dest) (src.connect(sigc::mem_fun(*ptr, &dest)))

#define widget_build(name, type, dest, builder) {\
        Gtk::Widget *w = nullptr; \
        builder->get_widget(name, w);\
        if (!w) { \
                throw std::runtime_error("No widget '" name "' found in main ui");\
        } \
        dest = dynamic_cast<type>(w);\
        if (!dest) {\
                throw std::runtime_error("Widget '" name "' cannot be casted to specified type.");\
        }\
}

#define timer_start() {\
                timer = Glib::signal_timeout().connect( \
                                sigc::bind(sigc::mem_fun(*this, &MainWindow::step), 0), \
                                ui.stepTimeChooser->get_value_as_int()); \
}
#define timer_stop() {\
        timer.disconnect();\
}
#define timer_step_time() ui.stepTimeChooser->get_value_as_int()


MainWindow::MainWindow(): ui()
{
        this->ui.builder = Gtk::Builder::create();
        std::ifstream ifs;
        try {
                ifs.open(UFilePaths::getResourcePath() + "main_ui.glade");
        }
```

```
        catch (const std::runtime_error& e) {
                throw std::runtime_error(std::string("Caught an exception while building the ui: ")
+e.what());
        }
        std::string line;
        std::string file;
        while (std::getline(ifs, line)) {
                file.append(line);
        }
        try {
                ui.builder->add_from_string(file);
        }
        catch (const Glib::Error& e) {
                throw std::runtime_error(std::string("Gtk builder threw an exception: ") + e.what());
        }

        widget_build("window", Gtk::Window*, ui.window, ui.builder);
        ui.window->add_events(Gdk::BUTTON_PRESS_MASK | Gdk::BUTTON_RELEASE_MASK);

        widget_build("camchooser", Gtk::SpinButton*, ui.camChooser, ui.builder);

        widget_build("step_time_chooser", Gtk::SpinButton*, ui.stepTimeChooser, ui.builder);

        widget_build("cambutton", Gtk::ToggleButton*, ui.startButton, ui.builder);
        signal_connect(ui.startButton->signal_toggled(), this, MainWindow::startButtonToggled);
        ui.startButton->set_can_focus(true);
        ui.startButton->grab_focus();

        widget_build("img1", Gtk::Image*, ui.img[0], ui.builder);

        widget_build("img2", Gtk::Image*, ui.img[1], ui.builder);

        widget_build("img3", Gtk::Image*, ui.img[2], ui.builder);

        widget_build("img_chooser", Gtk::FileChooser*, ui.imgChooser, ui.builder);
        signal_connect(ui.imgChooser->signal_file_activated(), this, MainWindow::imgChooserFileActivated);

        widget_build("training_chooser", Gtk::FileChooser*, ui.matchingChooser, ui.builder);

        widget_build("training_chooser1", Gtk::FileChooser*, ui.nonMatchingChooser, ui.builder);

}

MainWindow::~MainWindow()
{
}

void MainWindow::processColorSpace(cv::Mat& data, cv::Mat& dest) {
        cv::Mat rgb[3];
        cv::split(data, rgb);
        dest = rgb[0] - rgb[2];
        dest.convertTo(dest, CV_32F);
}

void MainWindow::drawImage(int imgIndex, cv::Mat data, int width, int height) {
        // show the original picture
        data.convertTo(data, CV_8U);
        if (data.channels() == 1) {
                std::cout << "Only 1 channel in the picture" << std::endl;
                std::cout << "size before: " << data.cols << " x "
                        << data.rows << ", channels: " << data.channels() << std::endl;
                cvtColor(data, data, CV_GRAY2RGB);
                std::cout << "Size after: " << data.cols << " x "
                        << data.rows << ", channels: " << data.channels()<< std::endl;
        }
        Glib::RefPtr<Gdk::Pixbuf> pix = Gdk::Pixbuf::create_from_data(
                        data.data, Gdk::COLORSPACE_RGB, false, 8, width, height, data.step);

        ui.img[imgIndex]->set(pix->scale_simple(
                        ui.img[imgIndex]->get_allocation().get_width(),
                        ui.img[imgIndex]->get_allocation().get_height(),
                        Gdk::INTERP_NEAREST));
        ui.img[imgIndex]->queue_draw();

}
```

```
void MainWindow::separateChannelsToCols(cv::Mat &src, cv::Mat &dest) {
        // make sure src is of type CV_32F
        src.convertTo(src, CV_32F);
        // split R, G and B channels to their own columns
        cv::Mat rgb[3];
        cv::split(src, rgb);
        for (int i = 0; i < 3; i++) {
                rgb[i] = rgb[i].reshape(1, 1);
        }
        dest.release();
        cv::vconcat(rgb, 3, dest);

}

void MainWindow::separateChannelsToRows(cv::Mat& src, cv::Mat& dest) {
        // make sure src is of type CV_32F
        src.convertTo(src, CV_32F);
        // split R, G and B channels to their own columns
        cv::Mat rgb[3];
        cv::split(src, rgb);
        for (int i = 0; i < 3; i++) {
                rgb[i] = rgb[i].reshape(1, rgb[i].rows * rgb[i].cols);
        }
        dest.release();
        cv::hconcat(rgb, 3, dest);
}

bool MainWindow::step(int timerID) {

        //processing may take some time, so we disconnect the timer first
        timer_stop();

        if (camera.isOpened()) {
                cv::Mat framein, frameout;
                camera.read(framein);
                cv::cvtColor(framein, frameout, CV_BGR2RGB);



                cvtColor(frameout, frameout, CV_RGB2GRAY);
                cvtColor(frameout, frameout, CV_GRAY2RGB);


        }


        timer_start();

        return true;
}


Gtk::Window *MainWindow::getMainWindow()
{
        return ui.window;
}

void MainWindow::startButtonToggled() {

        if (ui.startButton->get_active()) {
                std::cout << "Starting the camera" << std::endl;

                // button is pressed
                ui.startButton->set_label("Stop");

                // open the selected camera
                camera.open((int)(ui.camChooser->get_value() + 0.5f));
                if (!camera.isOpened()) {
                        Gtk::MessageDialog d("Camera ID entered was not valid. "
            "Try with a smaller camera ID.");
                        d.set_title("Unknown web cam");
                        d.set_size_request(300, 300);
                        d.run();
                }
```

```
                else {
                        timer_start();
                }

        }
        else {
                std::cout << "Stopping the camera" << std::endl;
                // button was released
                ui.startButton->set_label("Start");
                camera.release();
                timer_stop();
        }

}

void MainWindow::trainkNN(cv::Ptr<cv::ml::KNearest> nearest) {
        std::cout << "Training k-NN algorithm..." << std::endl;
        std::cout << "Selected training images:" << std::endl;
        std::vector<std::string> v = ui.matchingChooser->get_filenames();
        std::vector<std::string> nonmatch = ui.nonMatchingChooser->get_filenames();
        if (v.empty() && nonmatch.empty()) {
                std::cout << "Training data not selected. Select at least one training image." << std::endl;
                return;
        }

        cv::Mat data = cv::imread(v[0]);
        if (!data.empty()) {
                cvtColor(data, data, CV_BGR2RGB);
                separateChannelsToCols(data, data);
        }

        cv::Mat nonmatchData = cv::imread(nonmatch[0]);
        if (!nonmatchData.empty()) {
                cvtColor(nonmatchData, nonmatchData, CV_BGR2RGB);
                separateChannelsToCols(nonmatchData, nonmatchData);
        }

        // create a matrix with responses. With training data all pixels are considered to be matching
        cv::Mat responses(1, data.cols, CV_32F, 1.0f);
        cv::Mat nonresponses(1, nonmatchData.cols, CV_32F, 0.0f);

        if (!data.empty() && !nonresponses.empty()) {
                cv::hconcat(responses, nonresponses, responses);
                cv::hconcat(data, nonmatchData, data);
        }


        std::cout << "data type: " << data.type() << std::endl;

        std::cout << "Training sample rows: " << data.rows << ", columns: " << data.cols << ", type: "
                        << data.type() << std::endl;


        std::cout << "Responses data rows: " << responses.rows << ", columns: " << responses.cols
                        << ", type: " << responses.type() << std::endl;

        std::cout << "Creating train data..." << std::endl;

        cv::Ptr<cv::ml::TrainData> t = cv::ml::TrainData::create(data, cv::ml::COL_SAMPLE, responses);

        std::cout << "Training..." << std::endl;
        // train the data
        nearest->train(t);

        std::cout << "Training complete" << std::endl;

        return;
}

typedef std::chrono::high_resolution_clock Time;
typedef std::chrono::milliseconds ms;
typedef std::chrono::duration<float> fsec;

void MainWindow::imgChooserFileActivated() {
        std::cout << "File opened: " << ui.imgChooser->get_filename() << std::endl;
```

```
        cv::Ptr<cv::ml::KNearest> nearest = cv::ml::KNearest::create();
        trainkNN(nearest);

        cv::Mat img = cv::imread(ui.imgChooser->get_filename());
        cvtColor(img, img, CV_BGR2RGB);

        int width = img.cols;
        int height = img.rows;

        drawImage(0, img, width, height);

        std::cout << "Finding nearest..." << std::endl;

        separateChannelsToRows(img, img);

        cv::Mat results;
        std::cout << "kNN sample rows: " << img.rows << ", cols: " << img.cols << std::endl <<
                        ", type: " << img.type() << " should be: " <<  CV_32F << std::endl;
        auto t0 = Time::now();
        nearest->findNearest(img, 3, results);

        auto t1 = Time::now();
        fsec fs = t1 - t0;
        ms d = std::chrono::duration_cast<ms>(fs);
        std::cout << "took " << fs.count() << "seconds\n";

        std::cout << "kNN done. Result dimensions: " << results.cols << " x " << results.rows << std::endl;

        results *= 255;

        results = results.reshape(1, height);
        drawImage(1, results, width, height);

        // calculate the distances to black pixels
        results.convertTo(results, CV_8UC1);
//      cv::distanceTransform(results, results, CV_DIST_L2, 0);
//      cv::normalize(results, results, 0, 255, cv::NORM_MINMAX);
        cv::blur(results, results, cv::Size(8, 8));
        cv::threshold(results, results, 255/2, 255, CV_THRESH_BINARY);
        drawImage(2, results, width, height);
}
```

CMakeLists.txt

```
project(Dippa)
cmake_minimum_required(VERSION 2.8)


# included directories listed here
INCLUDE_DIRECTORIES("${PROJECT_BINARY_DIR}"
        "${PROJECT_SOURCE_DIR}"
        "${PROJECT_SOURCE_DIR}/src"
        "${PROJECT_SOURCE_DIR}/inc")

#linked libraries listed here
find_package(PkgConfig REQUIRED)
pkg_check_modules(GTK3MM REQUIRED gtkmm-3.0)
pkg_check_modules(OpenCV REQUIRED opencv)
find_package(Boost COMPONENTS system filesystem REQUIRED)

#OpenCV libraries are located here. Pkg config doesnt add directory path so we do it the dirty way
LINK_DIRECTORIES("/usr/local/lib")

INCLUDE_DIRECTORIES(
        ${GTK3MM_INCLUDE_DIRS}
        ${Boost_INCLUDE_DIRS}
        ${OpenCV_INCLUDE_DIRS}
)

set (LIBRARIES
        ${GTK3MM_LIBRARIES}
        ${Boost_LIBRARIES}
```

```
        ${OpenCV_LIBRARIES}
)

message(STATUS ${OpenCV_LIBRARIES})

#included header files
set (INCLUDES
        inc/mainwindow.hh
        inc/ufilepaths.h
        inc/ugraphicwidget.h
)

#included source files
set (SOURCES
        src/main.cc
        src/mainwindow.cc
        src/ufilepaths.cc
        src/ugraphicwidget.cc
)

#enable c++11
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")


add_executable(dippa ${INCLUDES} ${SOURCES})

TARGET_LINK_LIBRARIES(dippa ${LIBRARIES})
```