Tampereen yliopisto

Petteri Tolonen

KUVAUSJÄRJESTELMÄ KUDOSNÄYTTEEN ESIKÄSITTELYN DOKUMENTOINTIIN

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2019

TIIVISTELMÄ

Petteri Tolonen: Kuvausjärjestelmä kudosnäytteen esikäsittelyn dokumentointiin
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2019

Patologian laboratoriossa kudosnäyte käy läpi monivaiheisen prosessin. Esikäsittelyvaiheessa kudosnäytteestä leikataan tarvittaessa pienempiä paloja, jotka päätyvät lopulta näytelaseiksi ja mikroskoopilla katseltaviksi. Useissa suomalaisissa patologian laboratorioissa käytetään yhä alkeellisia työkaluja kudosnäytteen esikäsittelyvaiheen dokumentointiin. Yleensä tämä tarkoittaa kaaviokuvan piirtämistä sanallisten kuvausten lisäksi. Tällainen dokumentaatio on parhaimmillaankin epätarkka ja samanlaisen näytteen dokumentaatiossa on suuria eroja henkilöiden ja organisaatioiden välillä. 2000-luvulla digitalisoituminen on toden teolla alkanut myös patologian alalla ja erityisesti virtuaalimikroskopia, eli näytelasien digitointi ja katseleminen verkon yli yleistyy nopeasti. Myös siirtymä digitaalisiin esikäsittelyvaiheen kuvausjärjestelmiin on käynnissä.

Tässä diplomityössä käsitellään kuvausjärjestelmän kehittämistä kudosnäytteen esikäsittelyvaiheen digitaalista kuvaamista ja dokumentointia varten käyttäen kuluttajaelektroniikkaa ja UWP-sovelluskehitysalustaa. Sovellus toteutetaan hybridisovelluksena, eli web-tekniikoita käyttäen, mutta käärittynä natiivisovellukseen. Projektin aikana arvoioidaan UWP-sovelluskehitysalustan ja yleisesti hybridisovellus-lähestymistavan soveltuvuutta kuvausjärjestelmän toteutukseen.

Työn lopputuloksena todetaan, että esikäsittelyvaiheen kuvausjärjestelmän toteuttaminen kuluttajaelektroniikasta on mahdollista, vaikka käyttökokemus ei ole kaikilta osin ihanteellinen. Järjestelmän hyviä puolia ovat kuvien laatu ja annotaatiotyökalut, mutta kameran WiFi-yhteyden takia viiveet esimerkiksi kuvan ottamisessa kasvavat toisinaan liian suuriksi. Projektin sivutuotteena syntyi myös kuvausjärjestelmä ruumiinavausten dokumentoimista varten, jonka toteutuksessa kameraongelmista päästiin eroon, koska WiFi-yhteyttä ei käytetty. Järjestelmä soveltui hyvin hybridisovelluksena toteutettavaksi. Olemassaolevaa web-ohjelmointiosaamista pystyttiin hyödyntämään, eikä toteutuksen aikana havaittu suorituskykyongelmia, jotka olisivat johtuneet lähestymistavasta. UWP-ympäristö web-tekniikoiden kanssa todettiin ongelmalliseksi projektin aikana vastaan tulleiden suoritus- ja kehitysympäristöjen ongelmien vuoksi. Visual Studio -kehitysympäristö ei soveltunut erityisen hyvin web-tekniikoiden kanssa käytettäväksi ja sen toiminta oli epävakaata. Lisäksi Windows-käyttöjärjestelmän päivitykset toivat esiin sovelluksen suoritusympäristön bugeja, jotka rikkoivat sovelluksen.

Avainsanat: digitaalipatologia, dissektio, histologia, HTML5, hybridisovellus, makrokuvaus, patologia, UWP, WinJS

ABSTRACT

Petteri Tolonen: Imaging system for documenting tissue sample dissection
Master of Science Thesis
Tampere University
Information Technology
May 2019

In a pathology laboratory a specimen goes through a complex laboratory process. In the dissection phase the specimen is cut into smaller pieces, and medical glass slides are created as a result of the process. In many finnish laboratories dissection documentation is textual prose with a hand-drawn sketch of how the specimen was cut. This kind of documentation is vague at best and differences between individuals and organizations are significant. In the 2000s the digitalization has truly started in the area of pathology. Especially virtual microscopy — digitization of slides and viewing virtual slide images over a network — is more and more common. The shift towards digital macroscopic imaging solutions is also underway.

In this thesis a macroscopic imaging system is implemented using consumer electronics and the Universal Windows Platform (UWP). The application is implemented as a hybrid application, which means using web-technologies wrapped in a native application. The suitability of this hybrid approach is evaluated along with the suitability of the Universal Windows Platform for desktop application development.

The result of the thesis is that it is indeed possible to develop a macroscopic imaging system using consumer electronics, but the user experience is not ideal. Pros of the imaging system are image quality and annotation tools, but the WiFi-connection to the camera causes delays in operations, which are occasionally unbearably long. As a by-product of the project a variation of the imaging system was developed for documenting obductions. This variation didn't suffer from the camera problems, because WiFi-connection wasn't used. Hybrid-application approach with web-technologies was found to fit well for the case of macroscopic imaging system. Existing web-development knowledge could be applied and no problems were detected, which had been caused by the selected implementation approach. UWP with web-technologies is found problematic because of problems in the runtime and development environments. Visual Studio development environment wasn't well suited for web-development and it had instability issues. Additionally Windows updates introduced bugs in the UWP runtime environment, which broke the application.

Keywords: digital pathology, dissection, histology, HTML5, hybrid application, gross imaging, grossing, macroscopic imaging, WinJS, UWP

ALKUSANAT

Tämä diplomityö on kirjoitettu vuosien 2016-2019 aikana. Työn tekninen osa eli sovelluksen kehitys aloitettiin vuoden 2015 lokakuussa. Haluan kiittää vaimoani Junia väsymättömästä kannustuksesta työn loppuun saattamiseksi. Kiitokset Jilab Oy:lle mahdollisuudesta tämän työn toteuttamiseen, sekä koko yhtiön henkilökunnalle avusta ja tuesta.

Helsingissä, 2. toukokuuta 2019

Petteri Tolonen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

OHJELMA- JA ALGORITMILUETTELO

1 JOHDANTO

Kudosten visuaalinen tarkastelu on ollut olennainen osa tautien diagnosointia jo vuosisatojen ajan. Ennen mikroskoopin keksimistä taudit diagnosoitiin niiden aiheuttaminen paljain silmin nähtävien muutosten, makropatologioiden perusteella. Nykyään monien tautien luotettava diagnosointi perustuu kudosnäytteiden mikroskooppiseen tarkasteluun. Jotta kudosnäytettä voidaan tarkastella mikroskoopilla, täytyy siitä valmistaa näytelaseja. Niiden valmistaminen on monimutkainen laboratorioprosessi, jossa kudosnäytettä käsitellään sekä käsin että koneellisesti. Yksi kudosnäytteen laboratoriokäsittelyn vaiheista on esikäsittely, jossa kudosnäytettä tutkitaan silmämääräisesti ja leikataan tarvittaessa pienempiin osiin. Nämä osat laitetaan näytekasetteihin, valetaan parafiiniin ja siivutetaan näyteleikkeiksi, jotka useiden käsittelyvaiheiden jälkeen päätyvät näytelaseille ja mikroskoopilla katseltaviksi. Kudosnäytteen makroskooppinen tarkastelu esikäsittelyvaiheessa on edelleen olennainen osa taudin diagnosointia. Esikäsittelyvaiheen aikana näytettä käsittelevät henkilöt voivat tehdä siitä tärkeitä havaintoja ja kirjata ne ylös, mikä auttaa näytteeseen orientoitumisessa myöhemmin sekä mahdollisten ongelmien jäljittämisessä.

Digitalisaatio muuttaa käytäntöjä myös patologiassa. Näytelaseja digitoidaan enenevässä määrin, jolloin näytelasien katselu mikroskoopin läpi vähenee. Tietojärjestelmät ja virtuaalisten näytelasien katselusovellukset helpottavat näytelasien välillä navigointia. Useissa suomalaisissa laboratorioissa esikäsittelyvaiheen dokumentaatio on edelleen dissektiolomakkeelle täydennettyjen tietojen lisäksi käsin piirretty kaaviokuva, mutta esikäsittelyvaiheen digitaaliset kuvausjärjestelmät yleistyvät. Esikäsittelyvaiheen annotoidut kuvat voidaan tarjota katseltavaksi yhdessä kyseisen näytteen virtuaalisten näytelasien kanssa, jolloin ne auttavat huomattavasti yksittäisen näytelasin sijainnin hahmottamista koko näytteessä.

Web-tekniikoiden käyttö sovellusohjelmoinnissa on levinnyt viime vuosina mobiiliohjelmoinnista työpöytäsovellusten puolelle. Web-tekniikoilla toteutettuja sovelluksia, joita ei suoriteta selaimessa vaan käärittynä natiivisovellukseen kutsutaan yleisesti hybridisovelluksiksi. Tämä toteutustapa soveltuu hyvin sellaisten sovellusten toteuttamiseen, joilla ei ole tiukkoja suorituskykyvaatimuksia. Etuja ovat alemmat alemmat kehityskustannukset erityisesti usealle alustalle kehitettäessä ja aiemman web-ohjelmointiosaamisen hyödyntäminen.

Tässä diplomityössä käsitellään kuvausjärjestelmän kehittämistä kudosnäytteen esikäsittelyvaiheen digitaalista kuvaamista ja dokumentointia varten. Pyrkimyksenä on toteuttaa käyttökelpoinen kuvausjärjestelmä suomalaisen patologian laboratorion tarpeisiin. Jär-

jestelmän laitteisto rakkennetaan kuluttajaelektroniikasta ja sovellus toteutetaan UWP-hybridisovelluksena eli web-tekniikoita käyttäen. Lisäksi työssä arvioidaan web-tekniikoiden ja erityisesti UWP-sovelluskehitysalustan soveltuvuutta kuvausjärjestelmän toteutukseen.

Diplomityön rakenne on seuraavanlainen. Luvussa 2 esitellään patologia tieteenalana lyhyesti. Luvussa 3 käydään läpi web-tekniikoiden käyttöä sovellusohjelmoinnissa. Luvussa 4 esitellään työssä toteutettu DisseCam-kuvausjärjestelmä ja luvussa 5 arvioidaan työn tuloksia.

2 PATOLOGIA

Patologia on oppi sairauksiin solu-, kudos- ja elimistötasolla liittyvistä rakenteellisista ja toiminnallisista muutoksista [23, s.10]. Tässä työssä kehitettävä kuvausjärjestelmä on tarkoitettu käytettäväksi patologian laboratoriossa, joten kyseisen tieteenalan lyhyt esittely on aiheellista. Luvussa keskitytään toteutettavan kuvausjärjestelmän käyttötarkoituksen ja motivaation ymmärtämiseksi tarvittaviin taustatietoihin ja käytäntöihin, eikä tarkoitus ole esitellä patologiaa kattavasti. Aliluvussa 2.1 käydään läpi kudosnäytteen käsittelyvaiheet laboratoriossa ja aliluvussa 2.2 esitellään käsite digitaalipatologia taustoineen.

2.1 Histopatologisen näytteen käsittely

Patologia voidaan jakaa karkeasti tutkittavien näytetyyppien perusteella sytopatologiaan ja histopatologiaan. Histologia eli kudosoppi tarkoittaa kudosnäytteiden mikroskooppista tutkimusta ja sytologia eli soluoppi tarkoittaa yksittäisten solujen ja niiden rakenneosien tutkimusta [23, s. 1144]. Sytologiassa tutkitaan yksittäisiä soluja käytännössä ruumiin nesteistä. Histopatologialla tarkoitetaan histologian ja patologian yhdistelmää, eli tautien tutkimusta kudosnäytteistä mikroskoopin avulla, ja sytopatologialla vastaavaa tutkimusta irtosolunäytteitä tarkastelemalla. Sytologiassa tutkittavia yksittäisiä soluja ei pysty näkemään ilman mikroskooppia, joten tässä työssä toteutettava kuvausjärjestelmä on tarpeellinen vain histologisten kudosnäytteiden esikäsittelyssä, eikä sytopatologiaa tästä syystä käsitellä tämän laajemmin.

2.1.1 Vaiheet

Histopatologia edellyttää mikroskoopilla katseltavien näytelasien valmistamista kudosnäytteestä. Kudosnäytteen matka ihmisen kehosta näytelasille on monivaiheinen laboratorioprosessi ja tässä aliluvussa kuvataan tuon prosessin vaiheet lyhyesti.

Kiinnittäminen

Kun kudospala saapuu laboratorioon, ensimmäiseksi näyte kirjataan laboratoriotietokantaan [23, s. 1127]. Ensimmäinen varsinainen käsittelyvaihe on kiinnittäminen eli fiksaa-

tio. Erilaisia kiinnittämisessä käytettäviä kemikaaleja eli fiksatiiveja on olemassa paljon, mutta eniten käytetään formaliinia. Kiinnitysvaiheen tarkoituksena on säilyttää kudoksen rakenne mahdollisimman samanlaisena kuin se on ollut elävässä kudoksessa. Kudoksen rakenteen muuttumista voivat aiheuttaa esimerkiksi mikrobien tai entsyymien toiminta, joiden tehtävä luonnossa on hajottaa eloperäinen aines. [36, s. 70]

Esikäsittely

Kun näyte on kiinnitetty, se siirtyy esikäsittelyvaiheeseen [23, s. 1127]. Kiinnitysaineen haihtumisen takia esikäsittely suoritetaan yleensä vetokaapissa, tai muussa hyvin ilmastoidussa tilassa. Vetokaappi on alipaineistettu puoliavoin rakennelma, joka on tarkoitettu ihmiselle haitallisia kaasuja tuottavien aineiden tai työkalujen käsittelyyn [36, s. 19]. Jotta näytteestä voi tehdä näytelaseja, on näytepalojen sovittava näytekasetteihin, jotka ovat muovisia rei'itettyjä pieniä koteloita. Pienet näytteet sopivat näytekasetteihin sellaisenaan, mutta suuremmat näytteet vaativat käyntiinpanon, jossa näytteestä leikataan edustavat palat näytekasetteihin. Patologi tai laboratorioteknikko tutkii leikatessaan näytteitä paljain silmin ja dokumentoi havaintonsa sekä käyntiinpanoprosessin, jotta näytteeseen orientoituminen olisi helpompaa myöhemmissä vaiheissa. [23, s.1128]

Kudoskuljetus

Seuraava käsittelyvaihe on kudoskuljetus, joka itsessään sisältää useita osavaiheita, joista ensimmäinen on dehydrointi eli veden ja fiksatiivin poistaminen. Tämä tehdään yleensä nousevalla alkoholisarjalla, eli pitämällä näytettä alkoholiliuoksessa, jonka väkevyyttä nostetaan asteittain. Toinen kudoskuljetuksen osavaihe on kirkastaminen, jossa dehydroinnissa käytetty alkoholi poistetaan. Lopuksi kudos infiltroidaan, eli siihen imeytetään parafiinia. Aiemmat kudoskuljetuksen vaiheet ovat välttämättömiä, että parafiini saadaan imeytymään kudokseen. [23, s. 1128] [36, s. 106]

Valaminen

Kudoskuljetuksesta näytekappale siirtyy valamisvaiheeseen, jossa näytekappale valetaan tietyn kokoisen suorakulmaisen särmiön muotoisen parafiinikappaleen sisään [23, s. 1128]. Ensin valumuottiin valutetaan sulaa parafiinia ja muotin pohjalle asetellaan kasetti, jossa on sen yksilöivä tunniste [33, s. 75]. Näytekappale asetellaan kasetissa lämpölevyn päällä oikeaan asentoon, jonka jälkeen se siirretään kylmälevyn päälle, jotta parafiini jähmettyy ja kudos asettuu oikeansuuntaisesti [33, s. 76]. On tärkeää asetella kudospala muottiin oikein päin, koska väärin aseteltu kudos saattaa vaurioitua seuraavassa vaiheessa tai näytelasille ei saada näyteleikkeitä oikeasta kulmasta [36, s. 110]. Kovettunutta parafiinisärmiötä kutsutaan yleisesti nimellä ”blokki” [23, s. 1128].

Leikkaaminen

Seuraavaksi näyteblokista leikataan ohuita 1-10 µm paksuisia siivuja tähän tarkoitukseen tehdyllä laitteella, mikrotomilla [23, s. 1128]. Blokista leikattuja siivuja kutsutaan näyteleikkeiksi. Leikkeet asetetaan ensin kylmävesialtaaseen, jossa ne suoristetaan alustavasti ja poimitaan näytelasin päälle. Sitten leikkeet laitetaan lämpöhauteeseen, jossa ne suoristuvat kunnolla. Tämän jälkeen ne poimitaan toistamiseen näytelasille ja asetetaan lämpökaappiin, jossa leike kuivuu ja kiinnittyy lasille. [36, s. 129]

Värjääminen ja päällystäminen

Viimeiset näytteen käsittelyn vaiheet ovat värjääminen ja päällystäminen. Paljaat kudosnäytteet ovat värittömiä, eivätkä kudosrakenteet näkyisi mikroskoopilla ilman värjäystä. Värjääminen on itsessään monimutkainen prosessi ja vaatii näytteen kemiallista edelleenkäsittelyä. Näytteestä poistetaan parafiini ja se rehydroidaan, eli tuodaan vesi uudelleen kudokseen. Värjäyksiä on olemassa useita erilaisia eri tarkoituksiin. Päällystäminen tarkoittaa näytelasia ohuemman peitinlasin kiinnittämistä näytteen päälle sen suojaamiseksi. Peitinlasin kiinnittämiseen tarvitaan peitinaine, jonka valinta riippuu värjäyksessä käytettävien kemikaalien ominaisuuksista. [33, s. 78-80]

2.1.2 Esikäsittelyvaiheen dokumentointi

Mikro- ja makroskooppisten löydösten välinen korrelaatio on tärkeä osa diagnoosin tekemisessä. Esikäsittelyvaiheen dokumentaatio tarjoaa kuvaavan raportin, jonka avulla lukija voi rekonstruoida näytteen mielessään, sekä palvelee lasihakemistona, jonka avulla patologi voi jäljittää, mistä kohtaa näytepalaa lasien näyteleikkeikkeet on tarkalleen otettu. Lisäksi se kuvaa näytteen osituksen, eli mitkä osat näytepalasta on valittu ja leikattu mihinkin diagnostiseen tai tutkimustarkoitukseen. Kudosnäytteen esikäsittelyvaiheessa näytteen ominaisuuksista dokumentoidaan yleisesti koko, paino, väri, muoto ja koostumus, sekä mahdollisesti havaitut vauriot. [42, s.7]

Perinteisesti esikäsittelyvaiheen kuvailu on ollut sanallista proosaa, jossa kudoksen ominaisuuksia on pyritty kuvaamaan mahdollisimman tarkasti. Usein kudoksen väriä, ulkonäköä tai koostumusta kuvaillaan vertailemalla esimerkiksi kasveihin tai ruoka-aineisiin [22, s. 1]. Tällainen dokumentaatio vaihtelee paljon dokumentaation kirjoittajasta riippuen ja sen tulkitseminen on usein hankalaa. Valokuvaamalla näytteistä saadaan yhdenmukaisempaa dokumentaatiota ja pienennetään tulkinnanvaraisuutta.

2.2 Digitaalipatologia

Digitaalipatologia käsitteenä mainittiin ensimmäisen kerran tieteellisissä julkaisuissa vuonna 2000 [35, s. 1]. Perinteisen määritelmän mukaan sillä tarkoitetaan kuvapohjaista tietoympäristöä, joka mahdollistaa digitaalisen näytelasiin liittyvän informaation hallitsemisen tietokoneiden avulla [15]. Digitaalipatologian sovellusalue on kuitenkin laajentunut 2010-luvulla alkuperäisestä, eikä kyseinen määritelmä välitä todellista kuvaa nykyaikaisen digitaalipatologian laajuudesta. Kirjassaan ”Digital Pathology” Sucaet ja Waelput määrittelevät digitaalipatologian ”anatomisen ja mikroanatomisen patologian tietojärjestelmien alaksi” [35, s. 2]. Määritelmä on lavea, mutta kuvaa hyvin sitä, että enää ei ole kyse pelkästään näytelasiin liittyvän tiedon hallitsemisesta, vaan koko näytteeseen liittyvän digitaalisen kuva- ja muun informaation hallitsemisesta sekä käsittelystä. Tässä aliluvussa kuvataan digitaalipatologian kehittymistä ja sen osa-alueita sekä soveltamista käytäntöön.

2.2.1 Virtuaalimikroskopia

Digitaalipatologian käytännön soveltaminen alkoi näytelasien digitoimisesta, ja sen tuloksena syntyvien virtuaalisten näytelasien katselemisesta tietokoneen avulla. Tätä kutsutaan virtuaalimikroskopiaksi, joka on tärkeä osa digitaalipatologian sovellusaluetta.

Digitoidusta näytelasista käytetään yleisesti lyhennettä WSI (Whole Slide Image). Virtuaalisten näytelasien luomiseksi näytelasit täytyy digitoida korkealla resoluutiolla. Yleisesti virtuaaliselta näytelasilta vaaditaan resoluutio 0.20 tai 0.40 µm/pikseli riittävän laadun saavuttamiseksi. Näillä resoluutioilla mikroskoopin läpi näkyvä alue kattaa vain murtoosan koko näytteen alueesta, joten näytelasi täytyy digitoida pala kerrallaan. Yleisesti skannerit liikuttavat näytelasia moottoroidulla pöydällä objektiivin alla ja käyvät näytealueen läpi kuvaten sen järjestelmällisesti. Useimmat skannerit siirtävät näytettä askeleittain ja luovat yksittäisistä mikroskooppisista valokuvista koostuvan mosaiikkikuvan. Näytteen liikuttamiseen käytetty mekaniikka ei ole koskaan täydellinen, joten yksittäiset palat täytyy sulauttaa yhteen ohjelmallisesti. Toinen digitointitekniikka on viivaskannaus, jossa taltioidaan kuvaraitoja tasaisesti liikkuvasta näytteestä käyttäen lineaarista kamerasensoria. Tällöin kuvien yhdistäminen helpottuu, mutta lineaarisen kamerasensorin heikompi herkkyys hidastaa digitointia. [38, s. 38-39]

Näytelasien digitointi tuottaa valtavat määrät kuvadataa. Tyypillinen 20 x 15 mm näyte digitoituna resoluutiolla 0.40 µm/pikseli tuottaa kolmella 8-bittisellä värikanavalla noin 5 gigatavua pakkaamatonta dataa. Patologian laboratorio voi tuottaa tuhansia näytelaseja vuorokaudessa, joten on ilmeistä, että kuvadataa täytyy pakata ja sen hallintaan tarvitaan erityisesti tarkoitukseen suunniteltuja ohjelmistoja. [38, s. 43]

Virtuaalisten näytelasien katseleminen tavallisella kuvankatseluohjelmalla ei ole järkevää, eikä yleensä edes mahdollista, koska yksittäinen näytelasi on usein suurempi kuin

tietokoneen käyttömuisti. Sen sijaan WSI:stä noudetaan vain sillä hetkellä katseltava alue. Katseltaessa WSI:ä pienemmällä suurennoksella palvelin pienentää kuvan resoluutiota ja vähentää samalla siirrettävän datan määrää. Samaa toteutustapaa käytetään myös WSI:en katselemisessa paikalliselta kiintolevyltä. [38, s. 45]

Fyysisten näytelasien käytössä on monenlaisia ongelmia. Ne rikkoutuvat herkästi ja ajan mittaan kudos rappeutuu sekä värjäykset haalistuvat. Fyysisten näytelasien lähettäminen patologilta toiselle on aikaavievää ja kallista, sekä lisää lasien katoamisen tai tuhoutumisen riskiä. Fyysisten lasien varastointi ja arkistointi lisäävät myös kustannuksia. Virtuaalisten näytelasien avulla voidaan luoda verkon yli käytettäviä digitaalisia arkistoja, joiden ylläpitokustannukset ovat matalat verrattuna fyysisiin arkistoihin, ja jotka voivat tarjota kenen tahansa katseltavaksi laajan kirjon erilaisia näytteitä. [38, s. 45]

Virtuaalimikroskopia ja WSI:t tarjoavat monenlaisia etuja verrattuna perinteiseen mikroskopiaan ja näytelaseihin. Katseluergonomian voi virtuaalimikroskopiassa säätää mukavammaksi kuin tavallisella mikroskoopilla, jossa silmät väsyvät pitkäaikaisessa katselussa. Perinteisessä mikroskopiassa mikroskoopin läpi näkyvää aluetta voi muuttaa vaihtamalla käytettävää objektiivia, joten mahdollisuus vaihtaa mikroskoopin läpi näkyvää kuva-aluetta riippuu mikroskooppiin liitetyistä objektiiveista. Virtuaalimikroskopiassa ei tarvitse vaihtaa objektiivia tai säätää tarkennusta kohdalleen, vaan WSI:ä voi zoomata ja navigoida näytteessä helposti tietokoneen avulla. Useaa WSI:ä voi katsella samaan aikaan vierekkäin, mikä helpottaa näytteiden vertaamista. Lisäksi virtuaaliset näytelasit mahdollistavat automaattisten kuva-analyysien suorittamisen kuvadatalle. [38, s. 46].

2.2.2 Telepatologia

Yksi digitaalipatologian sisältämistä palveluista on telepatologia, jonka voisi kääntää suomeksi myös etäpatologiana. Se tarkoittaa nimensä mukaisesti patologian palveluiden tarjoamista matkan päästä. Telepatologiaa voidaan pitää digitaalipatologian esivaiheena, koska ensimmäiset telepatologian kokeilut olivat täysin analogisia. 1960-luvun lopulla mustavalkoista televisiokuvaa mikroskoopin läpi kuvatuista näytteistä siirrettiin Bostonin Logan-lentokentältä Massachussets General Hospitaliin, mikä voidaan katsoa ensimmäiseksi askeleeksi telepatologian soveltamisessa. Termi telepatologia keksittiin kuitenkin vasta vuonna 1986. [37, s. 56]

Telepatologiajärjestelmät on aiemmin jaettu kolmeen kategoriaan: dynaamisiin, staattisiin ja hybridijärjestelmiin. Dynaamiset järjestelmät ovat käytännössä etähallittavia mikroskooppeja, jotka lähettävät patologeille videokuvaa mikroskoopin läpi ja antavat etähallita mikroskooppia. Staattisissa järjestelmissä lähetetään yksittäisiä mikroskooppivalokuvia verkon välityksellä esimerkiksi sähköpostia käyttäen. Hybridijärjestelmissä etähallittavalla mikroskoopilla mahdollistetaan korkearesoluutioisten valokuvien ottaminen ja lataaminen. Uusin telepatologian laji on WSI-telepatologia, jonka virtuaalimikroskopia käytännössä mahdollistaa. [37, s. 56]

2.2.3 Makrokuvaaminen

Makroskooppista kuvaamista voidaan tehdä useissa näytteen käsittelyn vaiheissa, mutta yleensä sillä tarkoitetaan esikäsittelyvaiheen dokumentoimista valokuvaamalla. Näytteiden valokuvaamista tehtiin pitkään analogisesti filmille, mutta 2000-luvulla on siirrytty vähitellen käyttämään digitaalisia kuvausjärjestelmiä [22, s. 64].

Makroskooppisen ja mikroskooppisen kuvaamisen edut ja haitat poikkeavat hieman toisistaan, koska esikäsittelyvaihe voidaan kuvata suoraan digitaalisesti ilman fyysisen median valmistusta. Makroskooppisessa kuvaamisessa jokaisen analogisen kuvan valmistaminen veisi aikaa ja rahaa. Digitaalinen kuva on heti saatavilla, joten sitä voi arvioida välittömästi ottamisen jälkeen ja tarvittaessa ottaa uuden kuvan. Alkuinvestoinnin jälkeen esikäsitelyn kuvaamisen kulut ovat pienet.

3 WEB-TEKNIIKAT SOVELLUSOHJELMOINNISSA

Työpöytäsovellukset on perinteisesti ohjelmoitu käännettävillä ohjelmointikielillä. Web-tekniikoiden kehittymisen myötä niitä on alettu käyttää mobiilisovellusten ohjelmoinnissa ja viime vuosina myös työpöytäsovellusten ohjelmoinnissa. Tässä luvussa tutkitaan kyseisen ilmiön taustoja ja esitellään tämän työn kannalta olennainen UWP-alusta esimerkkinä web-tekniikoiden käytöstä sovellusten ohjelmoinnissa.

3.1 Web-sovellukset

Web-sovellus on asiakas-palvelin-sovellus, jossa sovelluksen asiakasosa suoritetaan selaimessa. Web-sovelluksen ja WWW-sivun ero on epäselvä, mutta web-sovelluksella viitataan yleisesti WWW-sivuun, joka muistuttaa toiminnaltaan työpöytäsovellusta tai mobiilisovellusta. Web-sovellusten kehittämiseen selaimessa on olemassa useita tekniikoita. Esimerkiksi Adobe Flash [2], Microsoft Silverlight [25] sekä Oracle Java [21] mahdollistavat web-sovellusten ohjelmoinnin, mutta vaativat selaimen lisäosan toimiakseen. Web-sovellus voidaan toteuttaa myös käyttämällä selainten tukemia HTML, CSS ja JavaScript-kieliä, jolloin lisäosia ei tarvita. Tässä työssä tarkastellaan vain viimeksi mainittuja web-tekniikoilla toteutettuja sovelluksia, joita kutsutaan usein myös HTML5-sovelluksiksi.

3.1.1 Yhden sivun sovellukset

Perinteisillä WWW-sivuilla sivunäkymät on toteutettu itsenäisinä HTML-sivuina, joiden välillä navigoitaessa sivudokumentti ladataan palvelimelta. Yhden sivun sovellukset (SPA, Single Page Application) ovat web-sovelluksia, joissa koko sivuston tarvitsemat tiedostot ladataan joko heti alussa tai taustalla ilman, että käyttäjän tarvitsee odottaa koko sivun latautumista.

Näkymien muokkaaminen toteutetaan yhden sivun sovelluksissa manipuloimalla WWW-sivun dokumenttioliomallia (DOM, Document Object Model). DOM on ohjelmointirajapinta, joka määrittää HTML- tai XML-dokumentin loogisen rakenteen sekä mahdollistaa hakujen suorittamisen ja muokkaamisen [9]. Näkymää päivitetään muokkaamalla DOM-puun haaroja: Lisäämällä ja poistamalla DOM-elementtejä tai vaihtamalla elementtien attribuutteja tai sisältöä.

Jos sovelluksen täytyy hakea tietoja palvelimelta ilman sivun uudelleenlatausta, käytetään yleensä AJAX-tekniikkaa (Asynchronous JavaScript and XML). AJAX on joukko tekniikoita, joita käyttäen sovellus voi kommunikoida palvelimen kanssa asynkronisesti. JavaScript-ohjelmointikielellä HTTP-pyynnöt lähetetään palvelimelle *XMLHttpRequest*-rajapinnan avulla. AJAX-tekniikan nimen lopun ”X” on hieman harhaanjohtava, koska palvelimen lähettämät vastaukset eivät rajoitu XML-muotoon. Itse asiassa XML-formaattia huomattavasti useammin käytetään JSON-formaattia (JavaScript Object Notation), jonka käsittely JavaScript ohjelmointikielellä on luontevaa, koska sen syntaksi on luotu JavaScript kielen olioliteraalien pohjalta. Käytännössä HTTP-pyyntöjen vastaukset voivat olla mitä tahansa formaattia, esimerkiksi paljasta tekstiä, HTML-katkelmia tai CSV-muotoista dataa. [14]

Reitityksellä tarkoitetaan web-sovellusten yhteydessä tiettyjen toimintojen suorittamista URL:n muuttuessa niitä vastaaviksi. Perinteisellä WWW-sivulla URL:n muuttaminen on ainoa tapa kommunikoida palvelimen kanssa, joten reititys on palvelimen vastuulla, mutta yhden sivun sovelluksissa reititys on asiakaspäässä eli selaimessa [11, s. 116].

3.1.2 Asynkroninen ohjelmointi

Asynkronisuudella tarkoitetaan ohjelmistotekniikassa joidenkin tapahtumien tai operaatioiden valmistumisen odottamista ja niihin reagointia. Synkronisesta ohjelmasta asynkroninen eroaa suoritusjärjestykseltään; asynkronista ohjelmaa ei suoriteta järjestyksessä, vaan tapahtumien käsittelijät suoritetaan silloin kun tapahtumia tulee. Myös websovellusten JavaScript-ohjelmointikoodi ajetaan järjestyksessä, joten pitkään kestävien operaatioiden odottaminen synkronisesti tekee käyttökokemuksesta tuskallisen. Websovellusten ohjelmoinnissa asynkronisten funktiokutsujen käyttäminen onkin käytännössä välttämätöntä. Asynkronisten funktioiden toteuttamiseen on olemassa JavaScript-ohjelmointikielellä kaksi tapaa: takaisinkutsut ja promise-oliot. [11, s. 41]

Siinä missä takaisinkutsu on pelkkä funktio, joka kutsutaan kun asynkroninen operaatio on valmis, *promise*-olio kuvaa tulevaisuuden tapahtuman. *Promise*-olioita voi välittää eteenpäin ja niiden valmistumista voi odottaa usea asiasta kiinnostunut taho kutsumalla *then*-funktiota [28]. Toimintamekanismi on verrattavissa tarkkailija-suunnittelumalliin sillä erotuksella, että *promise*-olion valmistuminen on kertaluontoinen tapahtuma. Yksi syntaksia takaisinkutsuihin verrattuna selkeyttävistä eduista on *promise*-olioiden ketjuttaminen. Jokainen *then*-funktio palauttaa *promise*-olion, jonka tulos välittyy seuraavalle *then*-funktiolle [28].

3.1.3 Moduulijärjestelmät ja riippuvuuksien hallinta

Web-sovellusten kehittäminen on muuttunut 2000-luvulla merkittävästi. Muiden sovellusten tavoin web-sovelluksista kehitetään suurempia ja monimutkaisempia kuin aiem-

min. Tämä on vaatinut kehitysympäristöjen ja -työkalujen mukautumista. Aiemmin websovellusten JavaScript-koodin jakaminen moduuleihin on tehty sijoittamalla moduulit erillisiin tiedostoihin ja lataamalla ne oikeassa järjestyksessä HTML:n *script*-elementin avulla. Ongelma tässä toteutustavassa on riippuvuuksien määrittelyn puuttuminen. Moduulien latausjärjestystä on ylläpidettävä käsin, jotta jokaisen moduulin riippuvuudet muihin moduuleihin saadaan tyydytettyä. Ohjelmointikoodin jakaminen moduuleihin, joiden välille voi määrittää riippuvuuksia helpottaa ohjelman rakenteen hahmottamista [11, s. 71].

JavaScriptissä on useita tapoja määritellä moduuleita. AMD (Asynchronous Module Definition) on vanhempi moduulistandardi, joka tukee moduulien lataamista asynkronisesti [11, s. 80]. Yleisempi moduulityyppi on CommonJS, joka on todellisuudessa joukko standardeja, jotka pyrkivät tekemään JavaScript-moottoreista keskenään yhteensopivia [11, s. 82]. CommonJS:ssä tiedosto on moduuli ja moduulit määrittelevät riippuvuutensa synkronisella *require*-funktiolla [11, s. 82]. Moduulin ulkopuolelle näkyvät osat määritellään sijoittamalla olio *module.exports*-muuttujaan [29]. CommonJS on huomattavasti yksinkertaisempi kuin AMD, koska se on synkroninen, eikä moduuleja tarvitse kääriä funktioihin [11, s. 82].

Node.js on selaimen ulkopuolella käytettävä JavaScript-suoritusympäristö, jonka avulla JavaScript-koodia voi ajaa palvelimella [26]. Node.js käyttää riippuvuuksien hallintaan npm-pakettienhallintajärjestelmää, ja npm on myös maailman suurin ohjelmistopakettivarasto [1]. Node.js-moduulit ovat CommonJS-moduulimäärittelyn toteutus [11, s. 82]. Npm:n kasvun myötä sen käyttö on laajentunut myös web-sovelluksiin.

JavaScript-kielen kehittyessä siihen on myös esitelty moduulijärjestelmä ECMAScript-standardin 6:nnen version mukana. ES6-moduulien syntaksi käyttää *import*-avainsanaa riippuvuuksien määrittelyyn ja *export*-avainsanaa moduulin rajapinnan määrittelyyn. [29]

Hyvin harvat selaimet tukevat edelleenkään moduulien lataamista. Moduulien käyttämiseen selainympäristössä on olemassa kahdenlaisia työkaluja: moduuliniputtimia (module bundler) ja moduulilataajia (module loader). Moduuliniputin nimensä mukaisesti niputtaa selaimessa ajettavan ohjelman lähdekoodin yhdeksi tai useammaksi tiedostoksi ja moduulilataaja lataa moduulit ajonaikaisesti. Moduuliniputtimet ovat saavuttaneet enemmän suosiota ja monet niistä sisältävät lisäominaisuuksia kuten JavaScript-koodin koodin lataamisen osissa laiskasti ja käyttämättömän koodin karsimisen [29].

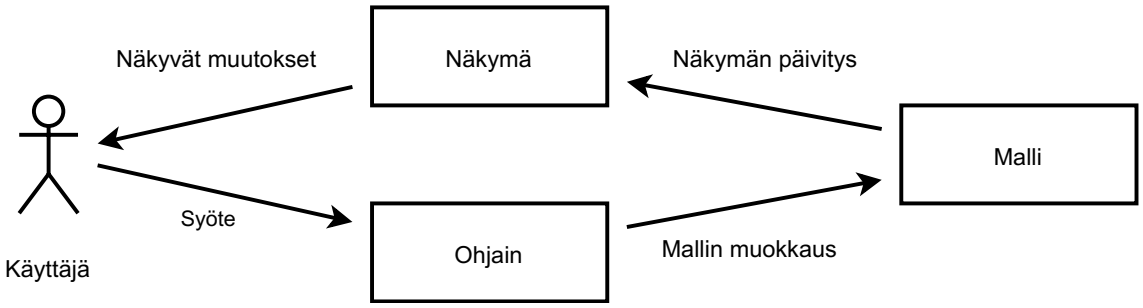
3.1.4 Arkkitehtuureista

Malli-näkymä-ohjain (MVC, Model-View-Controller) on käyttöliittymien ohjelmointiin tarkoitettu arkkitehtuurityyli, jonka kehitti alun perin norjalainen Trygve Reenskaug vierailevana tutkijana Xeroxin Palo Alto Research Centerissä 1970-luvun lopulla. Hänen mukaansa MVC:n perimmäinen tarkoitus on kuroa umpeen ihmisen käsitteellisen mallin ja tietokoneen digitaalisen mallin välinen kuilu, jolloin ideaalitapauksessa käyttäjälle tulee illuusio sovellusalueen tietojen katselemisesta ja muokkaamisesta suoraan. [31]

Reenskaugin alkuperäisessä MVC:ssä (models-views-controllers) *malli* esittää tietoa. Se voi olla yksi olio tai jokin olioiden yhdistelmä. *Näkymä* on visuaalinen representaatio mallista. Se on yhteydessä malliin ja noutaa siltä tarvitsemansa tiedot pyytämällä. Se voi myös päivittää mallia lähettämällä viestejä. *Ohjain* on linkki käyttäjän ja järjestelmän välillä. Se sjoittaa näkymät oikeille paikoilleen tarjoten näin käyttöliittymän ohjelman hallintaan. Ohjain muuntaa käyttäjän syötteet oikeanlaisiksi viesteiksi ja välittää ne eteenpäin yhdelle tai useammalle näkymälle. Lisäksi alkuperäisessä dokumentissa on kuvattu neljäs osa ”muokkain” (editor), joka muokkaa näkymän esitystapaan liittyvää tietoa. [30]

Vuosikymmenien aikana MVC:ä on toteutettu laajasti erilaisissa käyttöliittymäkirjastoissa, se on kehittynyt edelleen ja siitä on tehty useita variaatioita. Myös Reenskaug toteaa kotisivullaan myöhemmissä MVC-toteutuksissaan siirtäneensä osan syötteiden käsittelystä näkymiin [31].

MVC-arkkitehtuurityyli on esitetty kuvassa 3.1. Ohjain käsittelee käyttäjän syötteet, jotka se välittää mallille. Malli muuttaa sisäistä tilaansa ohjaimen pyynnöstä ja tieto muutoksista välittyy näkymälle, joka muuttaa näkymän vastaamaan muuttunutta mallia. MVC:n yksityiskohdat vaihtelevat toteutuksesta riippuen, joten riippuvuuksia ei ole kuvattu tarkemmin.



Kuva 3.1. *MVC-arkkitehtuurityyli*

Selain on käyttöliittymien kehityksen kannalta hyvin erilainen ympäristö verrattuna työpöytäsovellusten ohjelmoinnissa käytettyihin natiivi-sovelluskehyksiin. Käyttäjän syötteet tulevat selaimessa tapahtuma-olioina URL:n muutoksia lukuunottamatta suoraan DOM-elementteihin, joita muokkaamalla näkymät toimivat. Koska MVC:ssä ohjain-komponentti käsittelee tapahtumat, sen pitäisi jollakin tavalla saada tietoonsa DOM-elementtien tapahtumat. Selainympäristössä ohjaimen vastuulle kuuluvia toimintoja on luontevaa siirtää malliin ja näkymään, mikä on synnyttänyt MVC:stä useita web-ohjelmointiin tarkoitettuja variaatioita, joita kutsutaan yleisesti nimellä MV* [11, s. 115]. Useissa JavaScript-sovelluskehyksissä MVC:n ohjain typistyykin usein reitittimeksi, joka reagoi URL:n muutoksiin [11, s. 115].

Pääajatus kaikissa MV*-arkkitehtuureissa on sovellusaluelogiikan ja käyttöliittymälogiikan erottaminen toisistaan. Kun nämä vastuualueet on selkeästi eriytetty, voidaan sovellusta kehittää osissa, jotka ovat yhteydessä rajapintojen kautta. Käyttöliittymän eli näkymän toteutuksen muutokset eivät vaikuta malliin, eikä päinvastoin.

3.2 Hybridisovellukset

Merriam-Webster-sanakirjan mukaan hybridi on jotakin, joka koostuu kahdesta tai useammasta erillisestä elementistä eli on heterogeeninen alkuperältään, koostumukseltaan tai ulkonäöltään [19]. Hybridisovellus yhdistää natiivisovelluksen ja web-sovelluksen pyrkien saavuttamaan osan molempien eduista. Natiivisovelluksilla tarkoitetaan yleisesti sovelluksia, jotka on ohjelmoitu käyttäen käyttöjärjestelmän tarjoamaa kehitysalustaa ja rajapintoja. Erityisesti työpöytäsovellusten yhteydessä käsite on epätarkka, koska käytettävissä on useita ohjelmointikieliä, kehitysympäristöjä, kääntäjiä, tulkkeja ja käyttöliittymäkirjastoja.

Hybridisovelluksessa web-ohjelmointikoodia ajetaan käytännössä natiivisovelluksen sisällä selainkomponentissa. Tavallisen web-sovelluksen käytettävissä olevat laitteistorajapinnat ovat hyvin rajalliset. Hybridisovelluksella on käytettävissään lisäksi sovelluskehyksen tarjoamia rajapintoja, kuten tiedostojen muokkaus ja GPS-paikannus. Hybridisovellus käsitteenä ei määrää mitkä ohjelman osat ovat natiivia ohjelmointikoodia ja mitkä web-ohjelmointikoodia. Samassa sovelluksessa on joissain tapauksissa mahdollista käyttää jopa natiiveja käyttöliittymäkomponentteja yhdessä web-pohjaisten näkymien kanssa, mutta tällöin sovelluksen siirrettävyys alustalta toiselle hankaloituu.

Hybridisovelluksen olennaisimpia etuja ovat yhteisen lähdekoodin käyttö ohjelmoitaessa monelle alustalle ja alemmat kehityskustannukset [20]. Usealle alustalle kehitettäessä voidaan käyttää samaa web-ohjelmointikoodia, joka kääritään alustan mukaiseen sovellukseen, eikä kehittäminen eri alustoille vaadi osaamista jokaisesta alustasta erikseen. Web-kehittäjien löytäminen on helpompaa kuin natiivi-kehittäjien ja monesta organisaatiosta heitä voi löytyä jo valmiiksi [18, s. 9].

Web-sovelluksen etuna natiivisovelluksiin on niiden päivittämisen automaattisuus. Koska web-sovellus avataan aina selaimella, kaikilla käyttäjillä on varmasti käytössään sovelluksen uusin versio. Natiivisovellusten päivittäminen on käyttäjän vastuulla, vaikka usein käyttöjärjestelmä ja erilaiset päivitysapurit pitävät huolen siitä, että päivitykset asennetaan. Hybridisovelluksen kehittäjä voi päättää, kumpaa tapaa käytetään. Äärimmäisessä tapauksessa hybridisovellus voi olla vain ohut kuori web-sovelluksen päällä, eikä sovellus toimi lainkaan ilman verkkoyhteyttä. Edistyneempi toteutus voi tallentaa sovelluksen laitteen välimuistiin siten, että se toimii myös ilman verkkoyhteyttä. Joissakin tapauksissa päivittäminen halutaan pitää manuaalisena natiivi-sovellusten tapaan. [18, s. 7]

Hybridisovelluksilla on tietysti myös heikkoutensa. Ylimääräisen abstraktiokerroksen vuoksi hybridisovelluksen suorituskyky on väistämättä huonompi kuin vastaavan natiivisovelluksen [20] [18, s. 7]. Suorituskyvyn erojen merkittävyys riippuu sovelluksella tavoiteltavasta käyttäjäkokemuksesta ja sovelluksen tyypistä. Esimerkiksi peleissä on tärkeää, että sovellus vastaa käyttäjän syötteeseen viiveettä ja grafiikan piirtäminen on sulavaa. Toisaalta esimerkiksi kauppalistasovelluksessa suorituskykyvaatimukset ovat huomattavasti löysemmät. Rajapinnat alustan ominaisuuksien käyttöön ovat rajoitetummat kuin natiivi-

sovelluksessa, koska kaikkia alustan rajapintoja ei yleensä ole toteutettu hybridisovelluskehyksessä. Usein saatavilla olevia rajapintoja on jopa tarkoituksella rajoitettu tietoturvan parantamiseksi. Ulkoasun luominen alustan mukaiseksi voi olla haastavaa, mutta usein natiivien käyttöliittymäkomponenttien käyttö on myös mahdollista. Aina ulkoasua ei edes haluta mukauttaa alustan mukaan, vaan pyritään yhdenmukaiseen käyttöliittymään kaikilla alustoilla. Koska hybridisovelluksia ei ajeta selaimessa kuten web-sovelluksia, osoiterivi ei myöskään ole näkyvissä, eikä käyttäjä voi navigoida sovelluksessa sen avulla.

3.3 Universaali Windows-alusta (UWP)

Universal Windows Platform (UWP) on Microsoftin Windows 8 -käyttöjärjestelmässä esittelemä ohjelmistokehitysalusta, jossa Windows-ohjelmia on ensimmäistä kertaa mahdollista kehittää myös web-tekniikoiden avulla. UWP-sovelluksia kehitetään Microsoftin Visual Studio -kehitysympäristöä käyttäen. Sama UWP-sovellus voidaan ajaa Windows 10 -käyttöjärjestelmässä tabletissa, älypuhelimessa tai tavallisessa tietokoneessa. Muita ohjelmointikielivaihtoehtoja ovat perinteisemmät C# ja C++, mutta saman ohjelman toteutuksessa on myös mahdollista käyttää useita näistä tekniikoista. Tämä voi olla tarpeellista esimerkiksi, jos jotkin ohjelman osat vaativat laitteistoläheisempää ohjelmointia esimerkiksi suorituskykysyistä. [43]

Microsoft on muuttanut UWP-alustalle kehitettyjen sovellusten nimeämistä useaan kertaan. Alun perin sovelluksia kutsuttiin nimellä ”Metro-sovellukset” (Metro apps), mutta nimi jouduttiin vaihtamaan, koska Metro-tavaramerkki oli jo varattu. Nykyään alustalle tehdyistä sovelluksista käytetään Microsoftin WWW-sivuilla yleisesti nimeä ”Windows-sovellukset”. Kehittäjien dokumentaatiossa käytetään myös selkeämpää ”UWP-sovellukset”-nimeä. [24, 39]

Tässä työssä käytetään nimeä UWP-sovellukset, koska Windows-sovellus käsitteenä sekoittuu helposti perinteisiin Win32-työpöytäsovelluksiin. UWP-sovelluksista käsitellään pelkästään web-tekniikoilla toteutettuja UWP-sovelluksia, eikä C# tai C++-kielillä toteutettuja sovelluksia.

3.3.1 Erot web-ohjelmointiin

UWP-sovellukset suoritetaan hiekkalaatikkoympäristössä, joten niiden toiminta on huomattavasti rajoitetumpaa kuin perinteisten työpöytäsovellusten [7, s. 40]. Suoritusympäristö muistuttaa suuresti Microsoftin Edge-selainta, koska sivut renderöidään EdgeHTML-selainmoottorilla ja JavaScript-koodi suoritetaan Chakra-moottorilla, aivan kuten Microsoftin Edge-selaimessa [40]. UWP-sovelluksen suoritusympäristö poikkeaa selaimesta muun muassa seuraavilla tavoilla:

- Useat selainympäristön sovellusliittymän funktiot eivät ole käytettävissä tai toimivat

eri tavalla. Esimerkiksi ikkunoiden avaamiseen, siirtämiseen ja koon muuttamiseen liittyvät window-oliosta löytyvät löytyvät funktiot kuten *alert*, *prompt*), *open*, *moveTo*, *moveBy*, *resizeBy* ja *resizeTo* on poistettu käytöstä.

- UWP-suoritusympäristössä monille elementeille (audio, video, canvas) on käytössä rajapintoja, joita selainympäristössä ei ole olemassa.
- Sovellukset ajetaan lokaalissa kontekstissa, jossa JavaScript-ympäristöllä on käytettävissä WinRT-sovellusliittymä Windows-olion kautta. HTTP-pyyntöjä voi tehdä toisiin verkkotunnuksiin, mutta skriptien lataaminen muualta kuin paikallisesti on estetty.

Kaikki UWP-sovellukset suoritetaan sovellussäiliössä, joka estää prosessien välisen kommunikaation ja välittää pyynnöt järjestelmälle. UWP-sovelluksella on luku- ja kirjoitusoikeus vain sovelluksen omiin data-hakemistoihin. Kaikki muut tiedostojärjestelmäpyynnöt menevät välittäjän läpi. Sovelluksen oikeuksia hallitaan manifestitiedoston avulla. Esimerkiksi verkkoyhteyden käyttämiseen ja USB-muistien lukemiseen sovellus tarvitsee niitä varten manifestitiedostossa määritellyt oikeudet. [7, s. 40]

3.3.2 Sovellusprosessin elinkaari

Sovelluksen elinkaari käsittää sen tilan muutokset käynnistämisen ja sammuttamisen välillä. Ennen UWP:a Windows-sovellusten elinkaari oli yksinkertainen: sovellus oli joko käynnissä tai sammutettu. Mobiililaitteiden yleistyessä virransäästöstä tuli tärkeämpää, joten sovelluksen elinkaareen lisättiin lepotila (suspended). Myöhemmin Windows 10:n versiossa 1607 esiteltiin tilat ”tausta-ajossa” ja ”ajossa etualalla”. UWP-sovellus asetetaan lepotilaan, kun käyttäjä pienentää sen tehtäväpalkkiin tai vaihtaa toiseen sovellukseen. Tällöin sovelluksen suorittaminen pysäytetään ja sovellus jätetään muistiin, ellei käyttöjärjestelmä jostain syystä tarvitse muistia muiden ohjelmien käyttöön. Kun käyttäjä palaa UWP-sovellukseen, se saadaan nopeasti uudelleen ajoon. Asettamalla sovelluksia lepotilaan voidaan säästää virtaa ja saada etualalla ajossa olevalle ohjelmalle enemmän resursseja. Käyttöjärjestelmä saattaa sammuttaa lepotilassa olevan UWP-sovelluksen, mutta se näkyy edelleen käyttäjälle tehtäväpalkissa. Kun käyttäjä palaa käyttöjärjestelmän sammuttamaan sovellukseen, täytyy sen tila palauttaa lepotilaan siirtymistä edeltäneeseen tilaan. [44]

3.3.3 WinJS

WinJS (Windows library for JavaScript) on Microsoftin kehittämä avoimen lähdekoodin JavaScript-kirjasto, jota voi käyttää sekä UWP-sovellusten että web-sovellusten ohjelmointiin. WinJS:n käyttäminen UWP-sovellusten ohjelmoinnissa ei ole pakollista, mutta se helpottaa etenkin ohjelman käyttöliittymän toteuttamisessa ja auttaa tekemään ohjelman ulkoasusta yhdenmukaisen muiden UWP-sovellusten kanssa. WinJS ei ole sovelluskehys,

joten se ei pakota kehittäjää muotoilemaan sovellustaan tietyn rakenteen ja arkkitehtuurin mukaan, vaan tarjoaa rakennuspalikoita ja apuvälineitä sovelluksen kehittämiseen.

Asynkroninen ohjelmointi

WinRT-rajapinnat palauttavat *WinJS.Promise*-olioita, joten selkeyden vuoksi niitä on järkevää käyttää myös oman sovelluksensa sisäisissä rajapinnoissa. WinJS:n promise-olio on yhteensopiva aiemman *Promises/A*-standardiluonnoksen kanssa, joten WinJS:n then-funktiolle on *onSuccess* ja *onError* -takaisinkutsujen lisäksi mahdollista antaa *onProgress*-takaisinkutsu, jota kutsutaan tapahtuman edistyessä mikäli tällaisia tietoja on saatavilla [28]. Kyseistä *onProgress*-takaisinkutsua ei ole lopullisessa *Promises/A+*-standardissa, eikä myöskään WinJS:n promise-olion sisältämää *done*-funktiota [27]. WinJS:n promise-olio ei heitä käsittelemätöntä virhettä edelleen, ellei promise-oliolle ole kutsuttu *done*-funktiota [7, s. 155]. Onkin syytä olla tarkkana, etteivät asynkronisten kutsujen virheet jää piiloon vahingossa [5].

HTML:n prosessoinnista

WinJS käyttää HTML:n data-alkuisia attribuutteja lisätoiminnallisuuden määrittelyyn elementtien yhteydessä [7, s. 258]. Nämä attribuutit eivät kuitenkaan ilman HTML-elementtien käsittelyä tuota minkäänlaista toiminnallisuutta [7, s. 261]. WinJS tarjoaa useissa rajapinnoissaan *process* ja *processAll*-funktioita tällaisten elementtien käsitelyyn [7, s. 262]. Näiden kahden funktion ero on siinä, että *process* käsittelee vain yhden sille annetun elementin, mutta *processAll* käy läpi koko sille annetun elementin DOM-puun [7, s. 262]. Attribuuteilla määriteltyjen ominaisuuksien toteuttaminen on tietysti myös mahdollista suoraan JavaScript-koodista käsin, eikä kyseisiä HTML-attribuutteja, eikä siten käsittelyäkään ole pakko tehdä lainkaan. Seuraavissa aliluvuissa esitellään kontrollit, sidonnat ja resurssit, joiden toteuttamiseen voi käyttää edellä mainitun kaltaisia HTML-attribuutteja.

Kontrollit

WinJS:ssä ”kontrolli” on käyttöliittymäelementti, jonka avulla käyttäjä hallitsee sovellusta tai joka esittää tietoa [8]. Myös tavallisten HTML-elementtien voi ajatella olevan ”kontrolleja”, koska ne tarjoavat samanlaisen käyttäjäkokemuksen [7, s. 249]. WinJS tarjoaa kattavasti valmiita kontrolleja käyttöliittymän rakentamiseen, kuten päivämäärän ja ajan valitsimet sekä työkaluvihje-kontrollin [7, s. 258]. Monet tavalliset HTML-elementit kuten painikkeet ja syötekentät WinJS muuntaa CSS-tyyleillä yhdenmukaisiksi omien tyyliensä kanssa [7, s. 252]. WinJS tarjoaa myös muutamia kokoelmakontrolleja, jotka esittävät JavaScript-taulukon eri tavoilla käyttöliittymässä [7, s. 345]. Näistä kattavin on *ListView*-kontrolli, joka esittää taulukon parametrien mukaisena listana [7, s. 351].

Jos valmiista kontrolleista ei löydy sopivaa, voi kontrolleja kehittää myös itse. WinJS:ssä kontrolli on käytännössä JavaScript-luokka, joka hallitsee sen HTML-elementin ulkoasua ja toiminnallisuutta, johon se on instantioitu [7, s. 286]. Listauksessa 3.1 on esitetty toteutus erittäin yksinkertaiselle WinJS-kontrollille, joka näyttää parametrina annetun tekstin elementissä, johon kontrolli liitetään. Kontrollin instantiointi HTML:stä on esitetty sivukontrolli-esimerkin yhteydessä listauksessa 3.3. Monimutkaisemman kontrollin toteutuksessa täytyy lisäksi pitää huolta kontrollin resurssien vapauttamisesta [7, s. 259]. Jos itse toteuttamassaan kontrollissa varaa minkäänlaisia resursseja tulee omaan kontrolliinsa myös toteuttaa *dispose*-metodi, joka vapauttaa nuo resurssit [7, s. 287]. Näihin resursseihin kuuluvat esimerkiksi tapahtumankuuntelijoiden poistaminen, kesken olevien asynkronisten operaatioiden pysäyttäminen ja olioviittausten vapauttaminen [7, s. 287].

HTML-merkkauksessa kontrolli liitetään elementtiin *data-win-control*-attribuutin avulla ja kontrollille voi välittää parametrejä *data-win-options*-attribuutin avulla. Kontrollit täytyy määritellä johonkin nimiavaruuteen, jotta niitä voi käyttää toisten kontrollien HTML-merkkauksesta. *WinJS.Namespace*-nimiavaruudesta löytyy *define*-funktio nimiavaruuksien määrittelyyn. Esimerkissä on määritelty *HelloControl*-kontrolli *AppControls*-nimiavaruuteen. Kontrolli on määritelty JavaScript-luokkana *WinJS.Class.define*-apufunktiolla, jonka ensimmäinen parametri on luokan rakentaja. [7, s. 289]

Kontrollien HTML-merkkauksen käsittelystä vastaavat *WinJS.UI*-nimiavaruuden funktiot *process* ja *processAll* parsivat *data-win-options*-attribuutin arvon, kutsuvat *data-win-control*-attribuutin mukaisen kontrollin rakentajaa, joka muokkaa parametrina annettua DOM-puuta haluamakseen. Lopuksi viittaus kontrolliin tallennetaan HTML-elementin *winControl*-kenttään. [7, s. 261-262]

```
1  WinJS.Namespace.define("AppControls", {
2       HelloControl: WinJS.Class.define(
3          // Kontrollin rakentaja
4           function (element, options) {
5              element.winControl = this;
6              this.element = element;
7              if (options.message) {
8                  element.textContent = options.message;
9              }
10         }
11     )
12 });
```

Ohjelma 3.1. *Yksinkertaisen WinJS-kontrollin JavaScript-koodi [7, s. 289]*

Sivukontrolli on erityinen kontrolli, joka kuvaa kokonaisen sivunäkymän HTML:n, JavaScriptin ja CSS:n avulla. Sivukontrollien määrittelyyn WinJS tarjoaa *WinJS.UI.Pages.define*-apufunktion, jonka ensimmäinen parametri on projektin suhteellinen URL sivukontrollin HTML-tiedostoon. Toinen parametri on olio, jolla voi määritellä sivukontrolliin koukkumetodeita, joita kutsutaan kontrollin elinkaaren eri vaiheissa. Kyseiset metodit on esitetty taulukossa 3.1. *PageControl*-oliot voi renderöidä haluamaansa elementtiin käyttäen

WinJS.UI.Pages-nimiavaruuden funktiota *render*, jota käytettäessä sivun juurielementille kutsutaan automaattisesti *WinJS.UI.processAll*-metodia [7, s. 482].

Taulukko 3.1. *WinJS:n PageControl-olion uudelleenmääriteltävät koukkumetodit. [7, s. 142]*

Sivukontrollin metodi	Milloin kutsutaan?
init	Kutsutaan ennen kuin sivukontrollin sisältämät elementit on luotu.
processed	Kutsutaan kun *WinJS.UI.processAll* on valmis, eli kun sivulla olevat kontrllit on instantioitu, mutta ennen kuin sivun sisältö on lisätty DOM-puuhun.
ready	Kutsutaan kun sivu on lisätty DOM-puuhun, ennen edellisen sivun unload-metodin kutsumista.
error	Kutsutaan jos sivun lataamisessa tai renderöinnissä tapahtuu virhe.
unload	Kutsutaan kun sivulta on navigoitu pois. Oletuksena WinJS vapauttaa sivun kontrollien resurssit.
updateLayout	Kutsutaan *window.onresize*-tapahtuman tullessa, eli ikkunan koon muuttuessa.

```
WinJS.UI.Pages.define("/pages/home/home.html", {
    // Tässä voi määritellä uudelleen haluamiaan sivukontrollin metodeja
});
```

Ohjelma 3.2. *Yksinkertaisen sivukontrollin JavaScript-koodi.*

```
<!DOCTYPE html>
<html>
    <head>
        <!-- Ohjelman laajuiset tyylit -->
        <link href="/css/default.css" rel="stylesheet">
        <!-- Sivukontrollin tyylit -->
        <link href="/pages/home/home.css" rel="stylesheet">
        <!-- Sivukontrollin JavaScript-koodi -->
        <script src="/pages/home/home.js"></script>
    </head>
    <body>
        <!-- Ladattavan sivun sisältö. -->
        <div class="fragment homepage">
            <h1>Sivun otsikko</h1>
            <!-- Aiemmin luodun kontrollin käyttö -->
            <div data-win-control="AppControls.HelloControl"
                 data-win-options="{ message: 'Heipä hei' }" >
            </div>
        </div>
    </body>
</html>
```

Ohjelma 3.3. *Yksinkertaisen sivukontrollin HTML-merkkaus.*

Reitityksen toteuttamiseksi WinJS:ssä tarvitaan sivukontrollien ja *WinJS.UI.Pages*-rajapinnan lisäksi kaksi muuta osaa: rajapinta sovelluksessa navigointiin ja tämän rajapinnan yhdistäminen sivukontrollien kanssa eli reitityslogiikan toteutus. Ensimmäisen osan roolin täyttää *WinJS.Navigation*-rajapinta, joka tarjoaa sovellusliittymät navigointia varten, kuten navigaatiohistorian ja funktiot tietylle sivulle sekä eteen- ja taaksepäin navigoin-

tiin. *WinJS.Navigation* on myös tapahtumalähde navigaatioon liittyville tapahtumille. [7, s. 137]

Toinen osa reititystä ei kuulu WinJS-kirjastoon, mutta on toteutettu useissa UWP-sovellusesimerkeissä *PageControlNavigator*-luokkana, jonka voi kopioida omaan sovellukseensa [7, s. 138]. Kyseinen luokka toteuttaa navigointitapahtumien kuuntelun ja lataa uuden sivun navigointitapahtumien tullessa.

Sidonta

Sidonnalla (binding) tarkoitetaan mallin ja näkymän yhdistämistä. Kun ainoastaan näkymä kuuntelee mallin muutoksia ja muuttaa omaa tilaansa niiden mukaisesti, puhutaan yksisuuntaisesta sitomisesta. Jos lisäksi malli kuuntelee näkymää, puhutaan kaksisuuntaisesta sitomisesta. Kaksisuuntaisessa sitomisessa täytyy pitää huolta siitä, ettei muutosten päivittäminen luo ikuista silmukkaa, vaan tila päivitetään ainoastaan kun jotakin on muuttunut. Käytännössä sitominen tarkoittaa tarkkailija-suunnittelumallin käyttämistä. [7, s. 301]

Yksisuuntaisen sitomisen voi WinJS:ssä toteuttaa *WinJS.Binding*-rajapinnan funktioita ja *data-win-bind*-attribuuttia käyttäen. HTML-attribuuttia *data-win-bind* käyttäen voi määrittää mitkä datakonteksti-olion kentistä sidotaan mihinkin elementin ominaisuuteen [7, s. 303]. Kaksisuuntaiseen sidontaan WinJS:ssä ei ole tarjolla asiaa helpottavia työkaluja [7, s. 309].

Samaan tapaan kuin kontrollit, myös sidontojen HTML täytyy prosessoida. Datakonteksti välitetään oliona *WinJS.Binding.process* tai *WinJS.Binding.processAll*-funktiolle, joka luo JavaScript-koodin sidonnan toteuttamiseksi [7, s. 304]. Tavallisia JavaScript-olioita ei voi sitoa, joten *WinJS.Binding*-rajapinnassa on funktiot *as* ja *define*, joiden avulla tarkkailtavia olioita voi luoda. Funktio *as* tekee olemassaolevasta JavaScript-oliosta tarkkailtavan lisäämällä siihen tarvittavat metodit ja funktiolla *define* voi määritellä rakentajan tarkkailtavien olioiden luomiseen [7, s. 307]. Näistä tavallisin on *List*-luokka, joka tekee JavaScript-taulukosta tarkkailtavan olion [7, s. 331]. *List*-luokkaa voi käyttää esimerkiksi WinJS:n *ListView*-kontrollin kanssa.

Listauksissa 3.4 ja 3.5 on yksinkertaisen sidonnan HTML-merkkaus ja JavaScript-lähdekoodi, jotka esittävät sisäänkirjautuneen käyttäjän nimen ja kuvan. Esimerkissä *loginData*-olion *name*-kenttä on sidottu *span*-elementin *textContent*-kenttään eli elementin tekstimuotoiseen sisältöön ja *photoURL*-kenttä on sidottu *img*-elementin *src*-attribuutin arvoon. Kun *loginData*-olion kentät muuttuvat, myös vastaavat arvot DOM-puussa muuttuvat.

```
<section id="loginInfo">
    <p>Olet kirjautuneena käyttäjänä
        <span id="loginName" data-win-bind="textContent: name;"></span>
    </p>
    <img id="photo" data-win-bind="src: photoURL"/>
</section>
```

Ohjelma 3.4. *Esimerkki sidonnasta WinJS:ssä (HTML-merkkaus).*

```
const loginData = WinJS.Binding.as({
    name: "Matti Meikäläinen",
    photoURL: "http://example.com/userphoto.jpg"
});

WinJS.Binding.processAll(document.getElementById("loginInfo"), loginData);
```

Ohjelma 3.5. *Esimerkki sidonnasta WinJS:ssä (JavaScript-koodi).*

Kieliversiot

WinJS tarjoaa työkalut sovelluksen kääntämiseen eri kielille *WinJS.Resources*-nimiavaruuden työkalujen avulla. Käännökset luodaan JSON-tiedostomuodossa. Projektihakemiston *strings*-alihakemistoon luodaan jokaiselle lokaalille oma alihakemistonsa, jossa *resources.resjson*-niminen tiedosto sisältää käännösmerkkijonot kyseiselle lokaalille [7, s. 1096]. Esimerkiksi sivun otsikon voisi määritellä käännöstiedostossa seuraavasti:

```
{
    "app_title": "Sivun otsikko"
}
```

Yksittäisen käännetyn merkkijonon saa noudettua *WinJS.Resources.getString*-funktiolla [7, s. 1090]. Käännettyjä merkkijonoja voi sijoittaa myös HTML:n avulla käyttämällä *data-win-res*-attribuuttia [7, s. 1093]. Kontrollien ja sidontojen tapaan HTML-merkkaus täytyy tällöin prosessoida *WinJS.Resources*-nimiavaruuden *process* tai processAll-funktiolla [7, s. 1094]. Prosessoitaessa käännetyt merkkijonot noudetaan JSON-tiedostosta ja sijoitetaan *data-win-res*-attribuutin määrittelyn mukaisesti DOM-puuhun [7, s.1095]. Edellä määritellyn käännöksen käyttäminen HTML-merkkauksesta näyttää seuraavalta:

```
<span class="pagetitle" data-win-res="{textContent : 'app_title'}"></span>
```

Sovelluksen kieliversio pyritään päättelemään ja valitsemaan automaattisesti käyttöjärjestelmän asetusten perusteella [7, s. 1097]. Jos sopivaa kieltä ei löydy, käytetään sovelluksen manifestitiedostossa määriteltyä oletuskieltä [7, s. 1091].

Muut rajapinnat

WinJS tarjoaa lisäksi useita pieniä web- ja UWP-sovellusten ohjelmointia helpottavia rajapintoja. WinJS käyttää lokien kirjoittamiseen *WinJS.log*-funktiota. Lokien kirjaaminen

käynnistetään *WinJS.Utilities.startLog*-funktiolla, joka oletuksena formatoi lokiviestit ja tulostaa ne *console.log*-funktiolla. Lokien kirjoittamisen voi kustomoida mieleisekseen *startLog*-funktiolle annettavien parametrien avulla. [7, s. 166]

WinJS.Utilities-nimiavaruus sisältää myös paljon muita apufunktioita ja oliota, joista suurin osa liittyy DOM-puun käsittelyyn [46]. Suurin yksittäinen kokonaisuus *Utilities*-rajapinnassa on *Scheduler*-olio, joka tarjoaa työkalut asynkronisten operaatioiden vuorontamiseen ja priorisointiin [7, s. 157]. *WinJS.Application*-rajapinta tarjoaa sovellustason toimintoja, kuten tietojen tallentamisen sovelluksen data-hakemistoihin ja sovelluksen elinkaaritapahtumien käsittelyn [45].

4 DISSECAM-KUVAUSJÄRJESTELMÄ

Tässä luvussa esitellään toteutettu DisseCam-kuvausjärjestelmä. Esikäsittelyvaiheen käyntiinpanoa kutsutaan usein myös dissektioksi eli ”paloihin leikkaamiseksi”, vaikka pienimpiä näytepaloja ei leikata. Kuvausjärjestelmän nimi ”DisseCam” on yhdistelmä sanoista ”dissection” ja ”camera”.

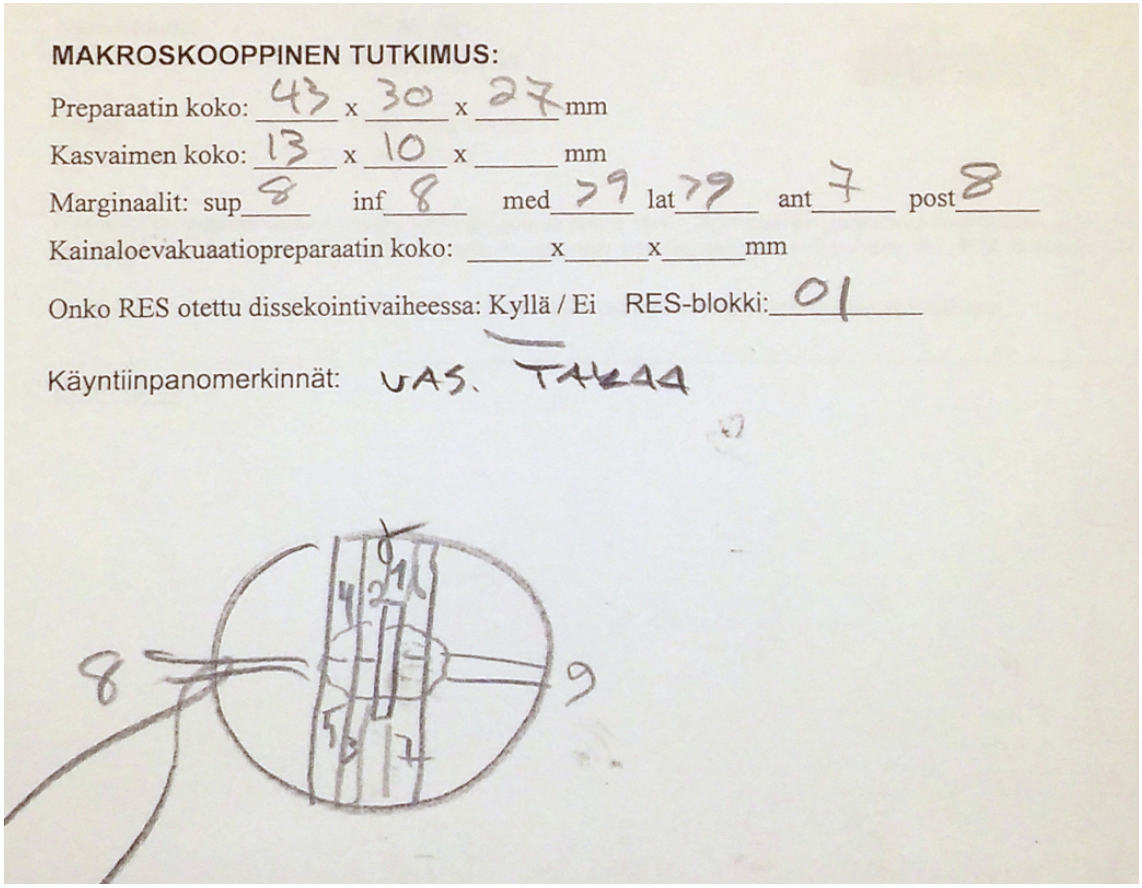
4.1 Taustaa

Jilab Oy kehittää digitaalipatologian kuvantamisratkaisuja ja erityisesti virtuaalimikroskopian sovelluksia suomalaisten asiakkaiden tarpeisiin. Idea makroskooppisen kuvausjärjestelmän kehittämiseen syntyi asiakkaan tarpeista ja halusta parantaa käytäntöjä patologian laboratoriossa. Ennen projektin aloittamista oli epäselvää, onko vaatimusten mukaisen järjestelmän toteuttaminen mahdollista resurssien puitteissa, koska riittävän hyvän etähallittavan kameran ja objektiivin löytyminen vaikutti epävarmalta. Jilab Oy on aiemmin kehittänyt ratkaisuja näytelasien digitointiin sekä digitoinnin tuloksena syntyvien virtuaalisten näytelasien tallentamiseen ja katseluun. Esikäsittelyvaiheen kuvausjärjestelmä täydentää Jilab Oy:n kuvantamisratkaisuja ja on siten looginen lisäys Jilab Oy:n tarjontaan.

Projektin alkaessa näytteen esikäsittelyn dokumentointi vaihtelee Suomessa eri laboratorioiden välillä, eikä kuvallista dokumentaatiota usein ole. Esikäsittelyvaiheessa täytetään yleensä lomake, johon merkitään olennaiset tiedot ja löydökset. Lomakkeeseen piirretään myös kaaviokuva, jolla tarkennetaan mistä kohtaa näytettä kasettikohtaiset näytepalat on leikattu. Kuvassa 4.1 on esimerkki erään suomalaisen laboratorion esikäsittelyvaiheen lomakkeen kaaviokuvasta. Vaikka tarkoitukseen tehtyjä kuvausjärjestelmiä on ollut olemassa useita vuosia, vasta viime vuosina suomalaisissa laboratorioissa on alettu siirtyä käyttämään kyseisiä järjestelmiä. Tämän työn tavoitteena on parantaa ja helpottaa esikäsittelyvaiheen dokumentointia kehittämällä kuvausjärjestelmä, joka on riittävän hyvä suomalaisten patologian laboratorioiden tarpeisiin.

4.2 Laitteiston esittely

DisseCam-kuvausjärjestelmän laitteisto koostuu Sonyn WiFi-yhteydellä varustetusta mikrojärjestelmäkamerasta, Microsoft Windows 10 Pro -päätelaitteesta sekä erilaisista oh-



Kuva 4.1. *Esimerkki asiakkaan esikäsittelyvaiheen dokumentaatiosta ennen kuvausjärjestelmän käyttöönottoa.*

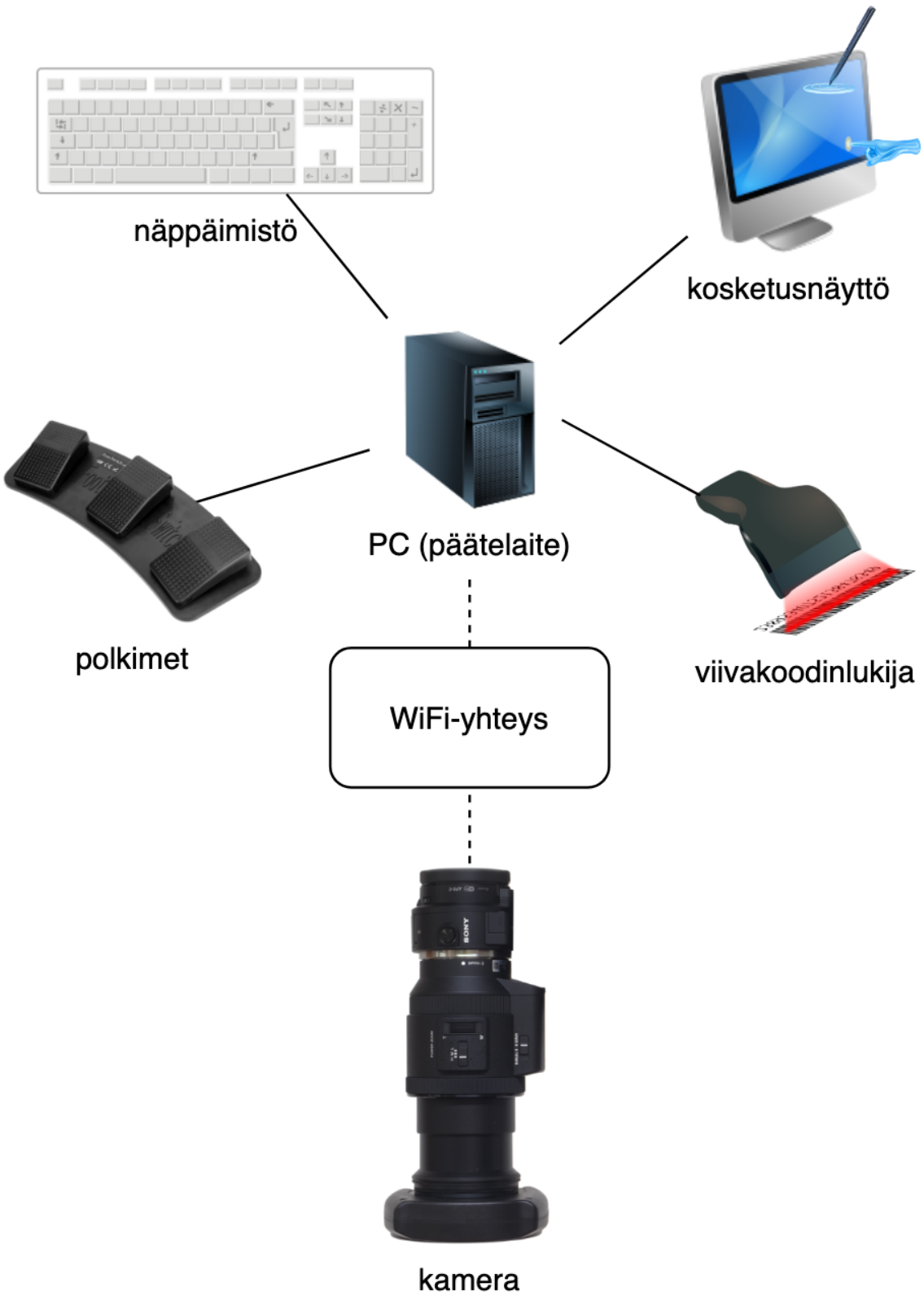
jainlaitteista. Päätelaite on yhteydessä kameraan langattomasti. Laitteistokokoonpanoa voidaan mukauttaa asiakkaan mieltymysten mukaiseksi esimerkiksi vaihtamalla päätelaite toiseen tai lisäämällä erilaisia ohjainlaitteita. Tässä luvussa kuvataan yleisellä tasolla kaikki järjestelmän pakolliset ja valinnaiset laitteet. Laitteisto on esitetty kuvassa 4.2.

4.2.1 Kamera ja objektiivi

Suurin haaste laitteiston rakentamisessa oli sopivan kameran löytäminen. Kameran ja objektiivin valinnassa kiinnitettiin huomiota seuraaviin ominaisuuksiin:

- Objektiivin ominaisuudet (tarkennusetäisyydet, motorisoitu zoom)
- Ohjelmointirajapinnat
- Riittävä kuvien resoluutio (vähintään 15 megapikseliä)
- Liitännät ja virransyöttö

Vaatimuksena kuvausjärjestelmälle oli myös, että kamera pitää pystyä asentamaan vetokaappiin siten, ettei se häiritse näyteen esikäsittelyä. Tämä tarkoittaa, että kamera täytyy asentaa vähintään 50 cm etäisyydelle kuvattavasta kohteesta. Lisäksi kuvattavaa aluetta



Kuva 4.2. *DisseCam-kuvausjärjestelmän laitteisto*

haluttiin rajata objektiivin zoomin avulla, joten objektiivin vähimmäistarkennusetäisyyden maksimi-zoomilla täytyi olla enintään 50 cm. Objektiivin piti myös suurentaa riittävästi, jotta kamera voitiin asentaa riittävän kauas kohteesta.

Kuvausjärjestelmän toteuttamiseksi oli välttämätöntä löytää kamera, joka tarjoaa ohjelmointirajapinnan kameran ohjaamiseen. Vähimmäisvaatimukset rajapinnan tarjoamille toiminnoille olivat kuvien ottaminen, zoomaus ja suoran videokuvan näyttö kameralta. Zoom-ominaisuuden hallinta rajapinnan kautta vaatii laitteiston tuen, eli käytännössä motorisoidun objektiivin. Tällaisia objektiiveja on saatavilla vähän ja niiden yhteensopivuus on usein rajattu vain tiettyihin kameramalleihin.

Käytännössä kaikista kuluttajakäyttöön tarkoitetuista digitaalikameroista löytyy nykyään USB-liitäntä, jonka kautta kameran muistikortille tallennettuja kuvia voi katsella, siirtää ja poistaa. Monia kameroita on myös mahdollista etäohjata USB-liitännän kautta esimerkiksi gPhoto-ohjelman avulla [16]. Valitettavasti gPhoto ei tue Sonyn kameroille kuvausjärjestelmän toteutuksessa vaadittuja live-kuvan välitystä ja zoom-tason hallintaa.

2010-luvulla useat kameravalmistajat ovat alkaneet valmistaa digitaalikameroita, jotka on varustettu langattomalla yhteydellä. Monien muiden valmistajien lähestymistavasta poiketen, elektroniikkavalmistaja Sony on kehittänyt langattoman yhteyden kautta käytettävän ohjelmointirajapinnan kameroidensa etähallintaan [17]. Rajapinta on melko nuori, mutta se tarjoaa edellä mainitut ominaisuudet, joiden avulla kuvausjärjestelmä on mahdollista toteuttaa. Kameran etähallintaa käsitellään tarkemmin luvussa 4.3.5.

Kameraksi valittiin Sonyn QX1, joka on peilitön mikrojärjestelmäkamera ilman näyttöä. Sonyn QX-sarjan kamerat käynnistävät automaattisesti kamerassa etähallintasovelluksen, jotta kameraa voi hallita WiFi-yhteyden kautta HTTP-rajapinnan yli. Muissa Sonyn kameroissa etähallinnan tajoava sovellus pitäisi käynnistää aina erikseen, mikä hankaloittaisi käyttöä.

4.2.2 Päätelaite

Järjestelmän päätelaitteen valinnassa tärkein asia oli laitteen käyttöjärjestelmä ja sen tarjoamat ohjelmointiympäristöt. Jo projektin alussa oli myös selvää, että järjestelmää halutaan usein hallita kosketusnäytöllä. Valittu kamera rajapintoineen ei rajoittanut päätelaitteen valintaa käytännössä millään tavalla. Varteenotettavia vaihtoehtoja laitteen käyttöjärjestelmäksi olivat Microsoftin Windows 10 ja Googlen Android. Lopulta valinnassa päädyttiin Windows 10:een sen monipuolisen laitteistotuen, erilaisten laitteiden saatavuuden ja mukauttamisen helppouden takia. Päätelaitteena voidaan käyttää mitä tahansa Windows 10 -käyttöjärjestelmällä varustettua laitetta poislukien älypuhelimet. Päätelaite voi olla tablet-tietokone, kannettava tietokone tai tavallinen PC.

4.2.3 Ohjainlaitteet

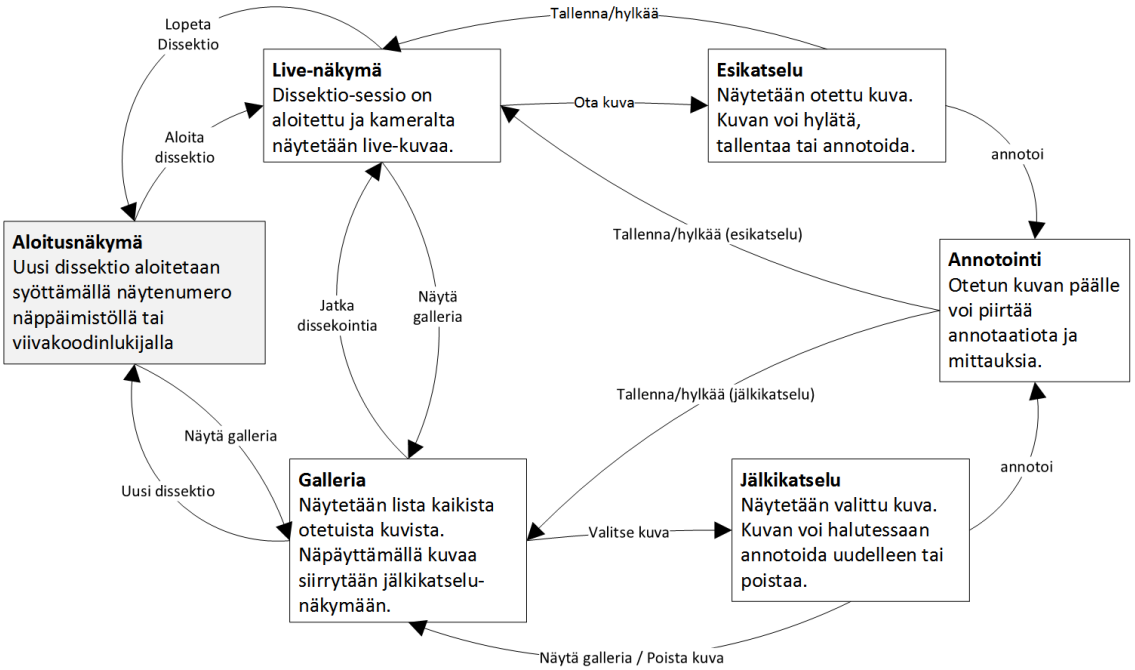
DisseCam-kuvausjärjestelmän ohjainlaitteista tärkein on graafisen käyttöliittymän ohjaamiseen käytetty osoitinlaite, eli yleensä kosketusnäyttö tai hiiri. Koska annotaatioiden piirtämisessä tarkkuus on tärkeää, valittiin osoitinlaitteeksi useiden kokeilujen jälkeen aktiivisella piirtokynällä varustettu kosketusnäyttö. Toinen pakollinen ohjainlaite on näppäimistö, joka voi olla joko USB-liitäntäinen tai langaton Bluetooth-yhteydellä toimiva. Yleisesti ottaen mikä tahansa hiiren tai näppäimistön tavoin toimiva USB-liitäntäinen tai Bluetooth-yhteydellä toimiva laite voidaan liittää päätelaitteeseen ja sitä voidaan käyttää kuvausjärjestelmän ohjaamiseen. Tästä yksi esimerkki on viivakoodinlukija, joka toimii järjestelmän kannalta näppäimistön tavoin, mutta helpottaa huomattavasti näytenumeron syöttämistä. Ohjainlaitteista näppäimistön tavoin toimivat myös USB-liitäntäiset polkimet, joiden avulla voidaan hallita kameraa ja tallentaa kuvia. Näytettä leikkaavan kädet ovat yleensä likaiset, joten polkimia käyttämällä vältetään päätelaitteen tai muiden ohjainlaitteiden likaantumista. Koska polkimien haluttu toiminta poikkeaa tavallisen näppäimistön toiminnasta, on polkimien toiminta toteutettu ohjelmaan erikseen, eikä mitä tahansa polkimia voi siis suoraan käyttää järjestelmässä.

4.3 Sovelluksen esittely

Tässä aliluvussa esitellään DisseCam-kuvaussovellus, joka koostuu päätelaitteella ajettavasta sovelluksesta ja palvelinkomponentista kuvien lähettämistä varten. DisseCam on UWP-hybridisovellus, jonka ominaisuuksiin kuuluvat räätälöidyt kuvien annotointityökalut, kuten etäisyyksien mittaustyökalu sekä ”blokkityökalu” näytekasetteihin leikattujen kudospalojen sijaintien merkitsemiseen. Kuvat otetaan aina käyttäjän antamaan näytenumeroon liittyvään kuvausistuntoon, jonka päättyessä kuvat lähetetään kuvapalvelimelle. Istunnon aikana käyttäjä voi ottaa näytenumeroon liittyen niin monta kuvaa kuin haluaa ja tehdä niihin annotaatioita. Kuvauspäätteelle tallennettuja kuvia pääsee halutessaan selaaman ja annotoimaan uudestaan tai poistamaan.

4.3.1 Käyttöliittymä

Päätesovelluksen käyttöliittymä pyrittiin pitämään rakenteeltaan yksinkertaisena. Navigaatiorakenne on yksitasoinen ja sovelluksessa on kuusi sivunäkymää, joiden välillä käyttäjä voi navigoida. Sovelluksen navigaatiokartta on esitetty kuvassa 4.3. Koska sovellus on tarkoitettu käytettäväksi tietyn kokoisella näytöllä, käyttöliittymää ei ole suunniteltu mukautumaan eri kokoisille tai pystyasentoisille näytöille.



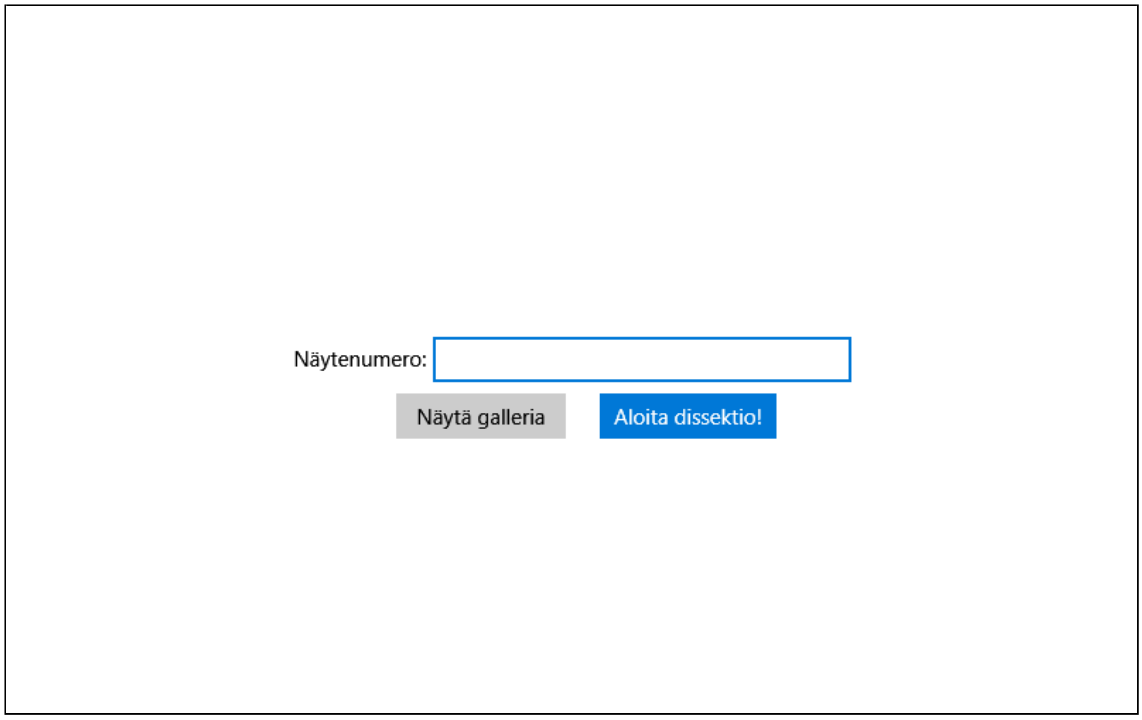
Kuva 4.3. *DisseCam-sovelluksen navigaatiokartta*

Aloitusnäkymä

Aloitusnäkymä on ohjelman yksinkertaisin sivunäkymä ja se näytetään ohjelman käynnistyttyä sekä aina silloin, kun käyttäjä haluaa aloittaa uuden esikäsittelyistunnon tai edellinen istunto on lopetettu. Näkymästä tarjotaan kaksi navigointipolkua: Käyttäjä voi aloittaa uuden istunnon syöttämällä näytenumeron näppäimistön tai viivakoodinlukija avulla ja painamalla ”Aloita dissektio”-painiketta. Jos näytenumero ei ole sallittua muotoa, tästä näytetään käyttäjälle virheilmoitus ja pyydetään syöttämään sääntöjen mukainen näytenumero. Jos kudosnäytteelle on jo olemassa kuvia, käyttäjälle näytetään tästä ilmoitus ja tarjotaan mahdollisuutta jatkaa kuvien ottamista kyseiselle näytteelle. Galleria-näkymään pääsee navigoimaan painikkeella ”Näytä galleria”. Aloitusnäkymä on esitetty kuvassa 4.4.

Live-näkymä

DisseCam-kuvausjärjestelmän tärkein ominaisuus on kuvien ottaminen. Kuvia otetaan live-näkymässä, jossa käyttäjälle näytetään kameralta videokuvaa. Käyttäjä voi hallita kameran objektiivin zoom-ominaisuutta ja ottaa kuvia. Kuvien ottamisessa käytetään automaattitarkennusta ja muut kameran asetukset on asetettu etukäteen asetustiedoston kautta. Kuvan ottamisen jälkeen siirrytään esikatselunäkymään. Live-näkymä on esitetty kuvassa 4.5.

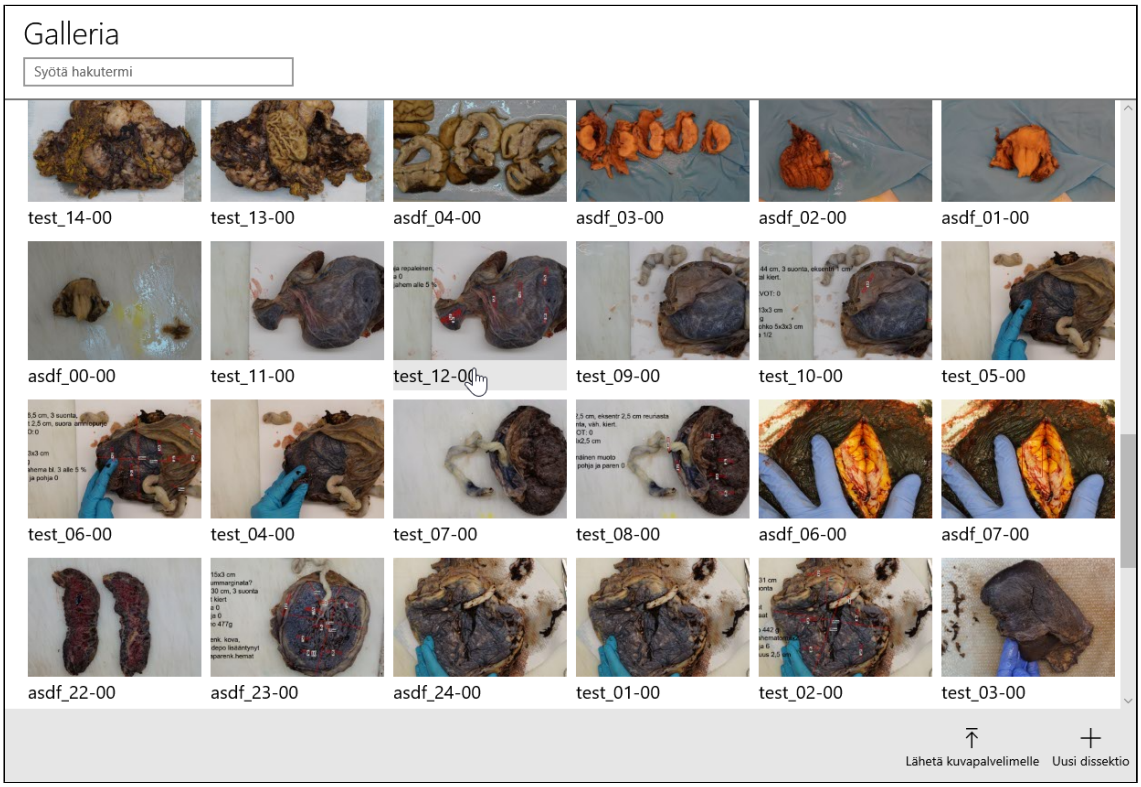


Kuva 4.4. *DisseCam-sovelluksen aloitusnäkymä*

Kuva 4.5. *DisseCam-sovelluksen live-näkymä*

Galleria-näkymä

Galleria-näkymässä käyttäjälle esitetään vieritettävänä listana kaikki päätelaitteen muistissa olevat makrokuvat. Jos kuvia on enemmän kuin ikkunaan mahtuu, näytetään oikeassa reunassa vierityspalkki. Näkymän yläreunassa on syötekenttä, jonka avulla listassa näkyviä kuvia voi suodattaa. Jos kuvaa ei olla vielä lähetetty palvelimelle, sen nimen jälkeen näytetään punainen huutomerkki-kuvake. Käyttäjäkokemuksen parantamiseksi kaikkia näytettäviä kuvia ei ladata muistiin kerralla, vaan kuvalistaa vieritettäessä kuvia ladataan lisää. Galleria-näkymä on esitetty kuvassa 4.6.

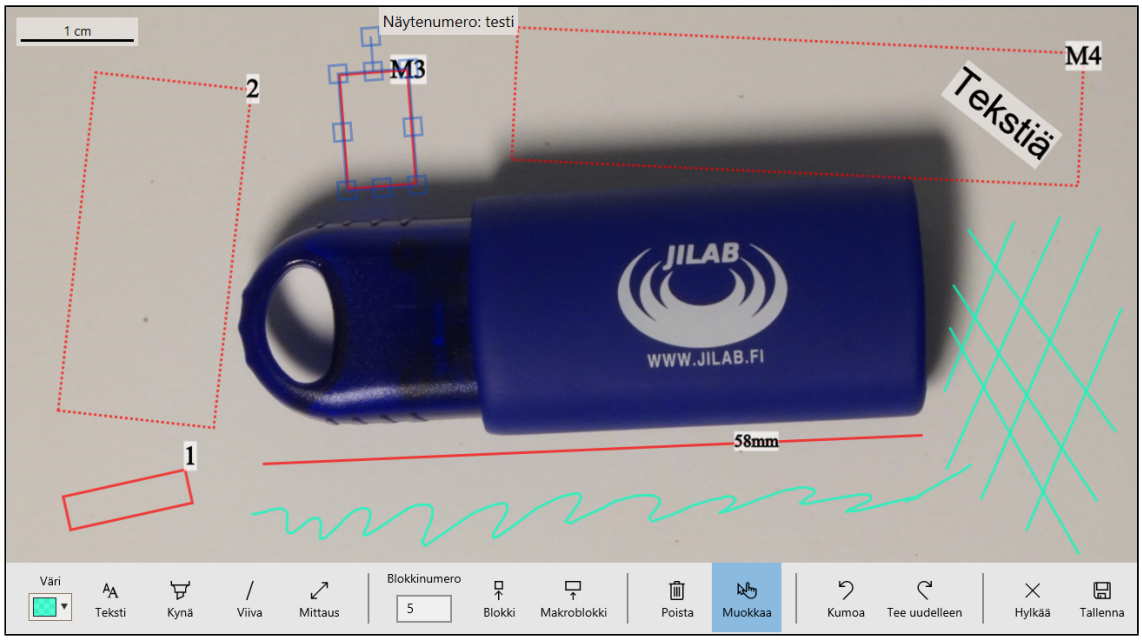


Kuva 4.6. *DisseCam-sovelluksen gallerianäkymä.*

Annotointinäkymä

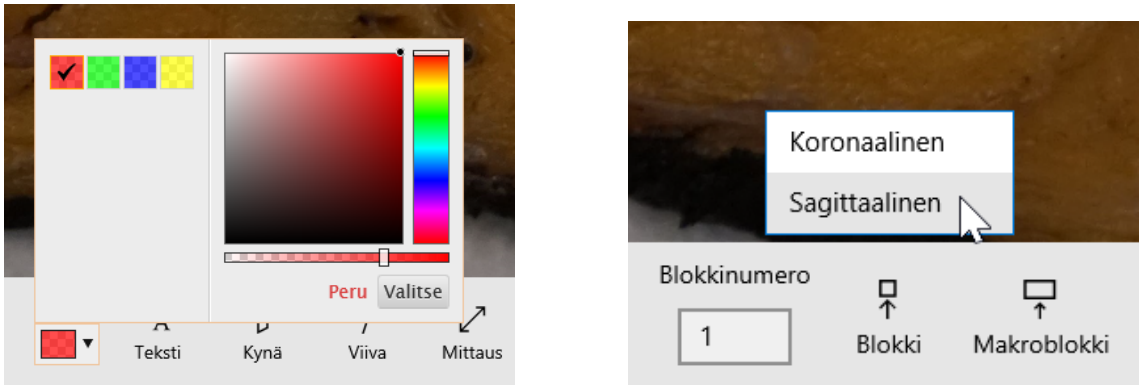
Pelkkien näytteistä otettujen kuvien tallentaminen ei ole esikäsittelyprosessin dokumentaation kannalta kovin hyödyllistä. Tämän vuoksi sovellukseen on kehitetty erilaisia annotointityökaluja, joilla käyttäjä voi piirtää ottamansa kuvan päälle merkintöjä. Annotointitoiminnallisuus on toteutettu käyttäen *Fabric.js*-kirjastoa [13]. Annotointinäkymä on esitetty kuvassa 4.7.

Annotointinäkymän vasemmassa yläreunassa on esitetty kuvan mittakaava yhden senttimetrin mittaisena viivana. Näkymän yläreunassa on esitetty myös näytenumero. Hallintapalkissa äärimmäisenä vasemmalla on värivalitsin, jonka avulla käyttäjä voi valita annotaatioiden värin. *Spectrum*-kirjastoa [34] käyttäen toteutettu värivalitsin on esitetty



Kuva 4.7. *DisseCam-sovelluksen annotointinäkymä*

kuvassa 4.8 vasemmalla.



Kuva 4.8. *Annotointinäkymän yksityiskohtia. Vasemmalla värivalitsin ja oikealla blokkityökalu.*

Käytettävissä olevat annotointityökalut hallintapalkin järjestyksessä vasemmalta oikealle ovat:

- *Teksti*, jolla voi lisätä kuvaan tekstikenttiä.
- *Kynä*, jolla voi piirtää viivoja vapaalla kädellä.
- *Viiva*, jolla voi piirtää suoria viivoja
- *Mittaus*, jolla voi mitata kuvasta janojen pituuksia millimetreissä.
- *Blokki*, jolla voi merkitä helposti näytekasetteihin leikattuja näytepaloja.

Blokkityökalu on näistä monimutkaisin. Sillä voi piirtää helposti juoksevalla numerolla merkittyjä tietyn levyisiä suorakaiteita. Blokkityökaluja on kaksi, tavallinen blokki ja makroblokki, jotka vastaavat laboratoriossa käytettäviä näytekasettien kokoja. Blokkeja piirretään samalla tavalla kuin viivoja ja blokin numero kasvaa automaattisesti jokaisen piirre-

tyn blokin jälkeen. Numerointia voi myös muuttaa työkalupalkissa olevan numerokentän avulla. Blokkityökalu on esitetty kuvassa 4.8 oikealla.

Blokkityökalun valitsemisen jälkeen käyttäjän täytyy vielä valita mistä suunnasta katsottuna haluaa blokin piirtää. Asiakkaan pyynnöstä suunnat on nimetty anatomisten tasojen mukaan termein ”sagittaalinen” ja ”koronaalinen”. Sagittaalitaso on mikä tahansa taso, joka jakaa ruumiin oikeaan ja vasempaan puoliskoon. Koronaalitaso on mikä tahansa taso, joka on kohtisuorassa sagittaalitasoon nähden, eli jakaa ruumiin etu- ja takaosaan. Blokkeja piirrettäessä ei jaeta kokonaista ruumista, mutta kameran näkökulmasta blokit piirretään vastaavasti. [3]

Fabric.js-kirjasto tarjoaa oletuksena käyttöliittymän annotaatioiden muokkaamiseen. Annotaatioita voi siirtää ja niiden kokoa voi muuttaa vetämällä annotaation ympärille ilmestyvästä kehikosta, kun annotaatio valitaan. Mittaus-tyyppisten annotaatioiden koon muuttaminen on kytketty pois päältä, koska koon muuttaminen vaatisi mittauksen pituuden laskemisen uudelleen.

Esi- ja jälkikatselunäkymä

Sovelluksen esi- ja jälkikatselunäkymät esittävät molemmat kuvausjärjestelmällä otetun kuvan. Näkymät eroavat hallintapalkin toimintojen osalta ja niitä käytetään eri tarkoituksiin.

Esikatselunäkymä näytetään kuvan ottamisen jälkeen. Sen tarkoitus on antaa käyttäjälle mahdollisuus arvioida otoksen onnistuneisuutta. Kuvaa ei vielä tallenneta esikatselunäkymään tultaessa, vaan käyttäjä voi valita hallintapalkista kuvan tallentamisen, hylkäämisen tai annotoimisen. Esikatselunäkymä on esitetty kuvassa 4.9. Annotoimisen valitseminen tallentaa alkuperäisen kuvan automaattisesti.

Jälkikatselunäkymään siirrytään galleria-näkymästä valitsemalla aiemmin tallennettu kuva. Käyttäjä voi annotoida kuvan uudelleen, poistaa kuvan tai palata takaisin galleria-näkymään. Jälkikatselunäkymä on esitetty kuvassa 4.10.

Yleiset käyttöliittymäkomponentit

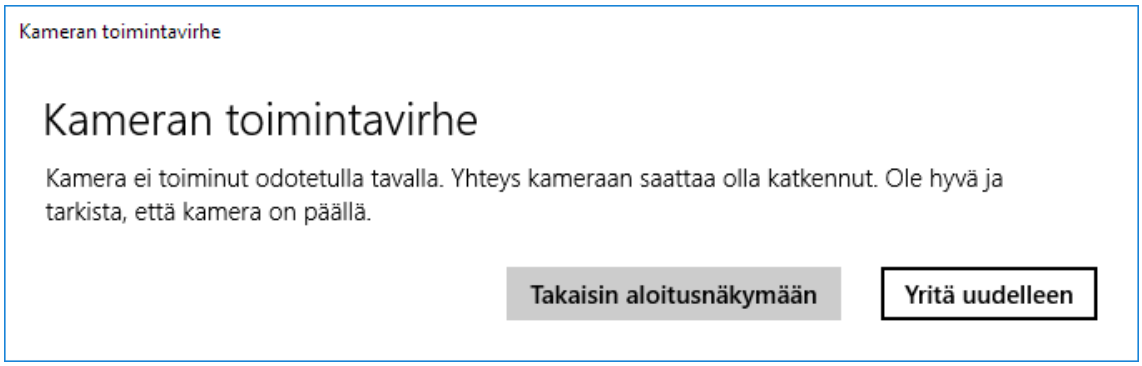
Sivunäkymien lisäksi sovelluksen käyttöliittymässä on komponentteja, joita käytetään useassa eri näkymissä. Tällaisia komponentteja ovat hallintapalkki, ponnahdusikkunat ja latausanimaatio.

Hallintapalkki on näkyvissä sivujen alareunassa ja sen kautta voi suorittaa sivun kontekstiin liittyviä toimintoja. Esimerkiksi annotointinäkymässä hallintapalkin kautta voi valita eri annotointityökaluja ja tallentaa kuvan. Aloitusnäkymä on ainut näkymä, jossa hallintapalkkia ei ole.

Ponnahdusikkunoilla käyttäjälle näytetään virheilmoituksia sekä esitetään kysymyksiä.

Kuva 4.9. *DisseCam-sovelluksen esikatselunäkymä*

Kuva 4.10. *DisseCam-sovelluksen jälkikatselunäkymä.*



Kuva 4.11. *Kameran toimintavirheestä ilmoittava ponnahdusikkuna.*

Esimerkiksi kuvan poistaminen vahvistetaan ponnahdusikkunalla ja kameran yhteysongelman tapauksessa näytetään virheilmoitus. Esimerkki ponnahdusikkunasta on kuvassa 4.11 esitetty ”Kameran toimintavirhe”-ponnahdusikkuna.

Latausanimaatio näytetään koko käyttöliittymän päällä, kun käynnissä on operaatio, jota käyttäjän täytyy odottaa. Tällaisia operaatioita ovat esimerkiksi kuvan ottaminen ja kuvalistauksen lataaminen.

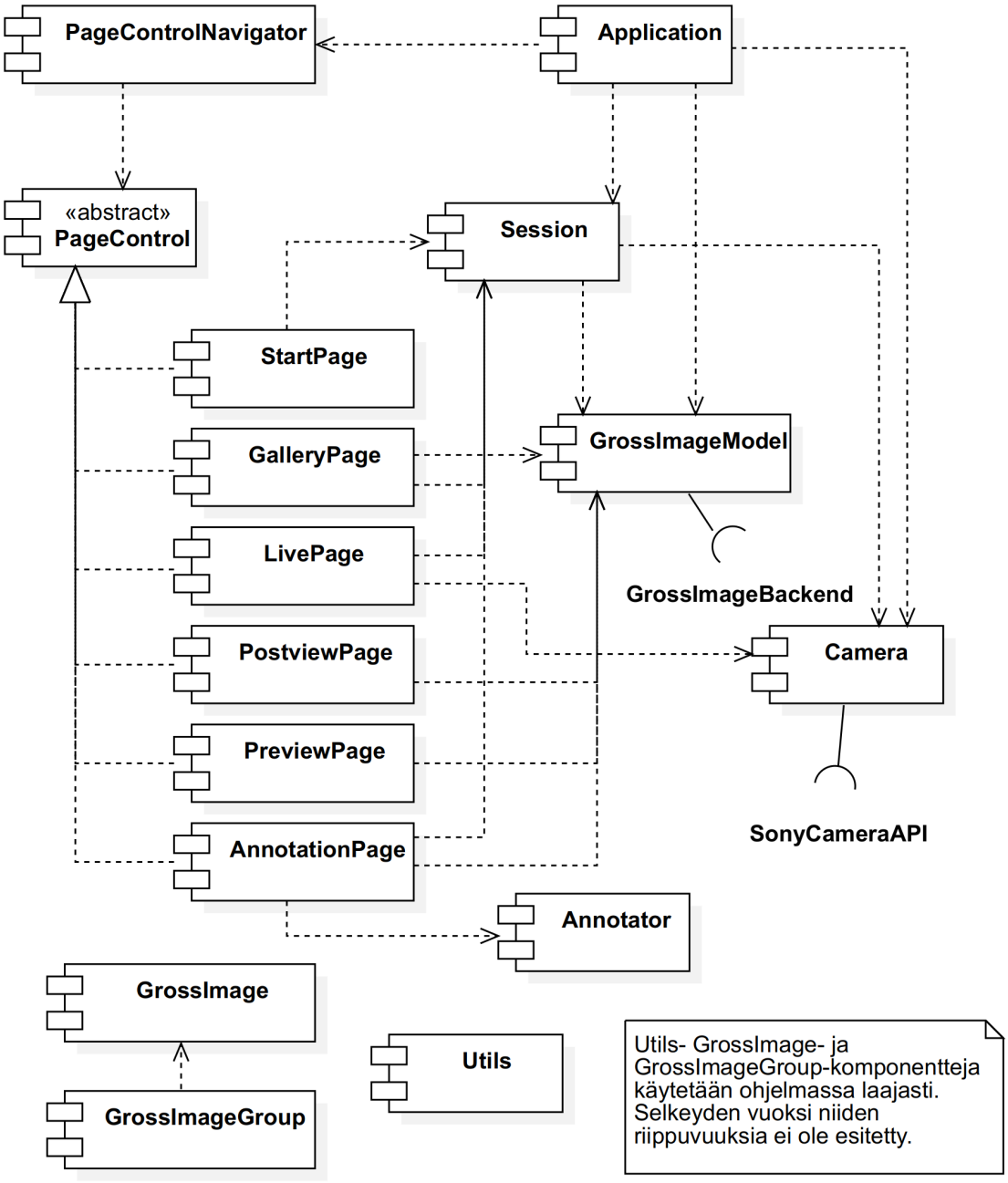
4.3.2 Arkkitehtuuri

DisseCam-sovelluksen arkkitehtuuri muistuttaa MVC-arkkitehtuurityyliä, mutta ohjaimen vastuut on siirretty sivukontrolleihin ja reitittimeen. Kuvassa 4.12 on esitetty DisseCam-sovelluksen komponenttikaavio. Kaikki sovelluksen sivunäkymät ovat omia sivukontrollejaan, jotka on määritelty *WinJS.UI.Pages.define*-funktiolla. Käyttäjän syötteet käsitellään sivukontrolleissa, jotka reagoivat niihin esityslogiikan mukaan. Sovelluksen sivunäkymiä ei ole ositettu pienempiin alinäkymiin, koska niiden uudelleenkäytölle on sovelluksessa hyvin vähän tarvetta. *PageControlNavigator*-on aliluvussa 3.3.3 mainittu reititinkomponentti, joka vastaa sovelluksen sivunäkymien ja kontekstin vaihtamisesta, kun *WinJS.Navigation*-rajapinnasta löytyviä navigointifunktioita kutsutaan.

Application-komponentti vastaa muiden komponenttien alustamisesta ja ohjelman elinkaaren hallinnasta. Se lataa sovelluksen asetustiedoston ja kuuntelee sovelluksen elinkaaritapahtumia *WinJS.Application*-rajapintaa käyttäen.

Galleria-näkymässä kuvalistaus on toteutettu *WinJS.UI.ListView*-kontrollina, joka on sidottu sivukontrollissa kuvalistaan *WinJS.Binding.List*-luokan avulla. Muutoksia kuvahakemistoihin ei kuunnella, vaan kuvalistaus noudetaan Galleria-näkymän sivukontrollin sisällä manuaalisesti, jolloin kuvalistaus galleria-näkymässä automaattisesti päivittyy.

Tärkein ohjelmassa tallennettava tietoyksikkö on makrokuva, joka on sovelluksessa abstrakti *GrossImage*-tietotyyppi. *GrossImage* toimii apukomponentteina kuvainformaation välittämiseen ohjelman sisällä yhdessä *GrossImageGroup*-luokan kanssa, joka kuvaa joukon *GrossImage*-olioita.



Kuva 4.12. *DisseCam-sovelluksen komponenttikaavio*

GrossImageModel on mallikomponentti, jonka avulla voidaan validoida näytenumero, pyytää seuraavan kuvan tunnisteet, tallentaa kuvia, poistaa kuvia, listata kuvia, lähettää kuvia palvelimelle ja ladata kuvia palvelimelta. *GrossImageModel* toimii myös tehtaana *GrossImage*-olioiden luomiseen. *GrossImageModel* käyttää *GrossImageBackend*-HTTP-rajapintaa kuvien lähettämiseen kuvapalvelimelle.

Session eli istunto on komponentti, joka vastaa tietyllä näytenumerolla aloitetun kuvausistunnon hallinnasta. Se tietää istunnon aikana otetut kuvat ja osaa suorittaa istunnon aloittamisessa ja lopettamisessa tarvittavat toimenpiteet kutsumalla *GrossImageModel*-komponenttia. Istunto-olio tarkkailee paikalliseen kuvavarastoon tehtyjä muutoksia *GrossIma-*

geModel-komponentin avulla ja päivittää omaa kuvalistaansa, jos uusia kuvia ilmestyy.

Camera-komponentti kapseloi kameran HTTP-rajapinnan ja täydentää sitä. Komponentin avulla kameraa voi hallita ja sen asetuksia muuttaa. Komponentti osaa myös piirtää kameralta tulevaa live-kuvaa HTML:n *canvas*-elementtiin. Komponentti käyttää Sonyn kameroiden tarjoamaa HTTP-rajapintaa, joka on kaaviossa esitetty nimellä *SonyCameraAPI*.

Annotator-komponenttia käytetään annotaatioiden piirtämiseen otetun kuvan päälle *AnnotationPage*-sivukontrollissa. Komponentti käyttää piirtämiseen *Fabric.js*-kirjastoa. *Utils*-moduuli sisältää pieniä apufunktioita muun muassa ponnahdusikkunoiden ja latausanimaation näyttämiseen ja piilottamiseen.

4.3.3 Kuvausistunto ja kuvien tallentaminen

Yksi monimutkaimpia asioita web-sovellusten arkkitehtuurin suunnittelussa on sovelluksen tilan hallinta. Sovelluksella on kahdenlaista tilaa: väliaikaista ja pysyvää. Väliaikainen tila on sovelluksen omaa tilaa, josta ei olla kiinnostuneita sovelluksen ulkopuolella. Esimerkki väliaikaisesta tilasta on esimerkiksi hakukentän sisältämä arvo. Pysyvä tila tarkoittaa sovelluksen tietomalliin kuuluvaa tietoa, joka halutaan tallentaa pysyvästi esimerkiksi tietokantaan. Esimerkki tällaisesta tiedosta on DisseCam-sovelluksen tapauksessa sovelluksella otetut kuvat. Haastavaksi tilan hallinnan tekee tiedon ja sen näkymien pitäminen keskenään ajan tasalla. Web-sovelluksen tapauksessa tieto on DOM-puussa, sovelluksen paikallisessa tietovarastossa, sekä mahdollisesti tallennettuna pysyvästi tietokantaan.

DisseCam-sovelluksessa otetut kuvat tallennetaan ensin päätelaitteen kiintolevylle. Kuvat tallennetaan JPEG-tiedostoina, joiden nimi on muotoa

```
123ABC_XX_YY.jpg
```

missä ”123ABC” on näytenumero, ”XX” on kuvanumero ja ”YY” on annotaationumero. Näytenumero on laboratoriossa näytteelle annettu kirjaimia ja numeroita sisältävä vaihtelevan mittainen tunniste, joka yleensä luetaan näytteen säiliön viivakoodista. Kuva- ja annotaationumerot ovat nollalla alkavia kaksinumeroisia tunnisteita. Annotointinumero ”00” tarkoittaa aina annotoimatonta eli alkuperäistä kuvaa.

Kun käyttäjä syöttää näytenumeron aloitusnäkymässä, sovelluksessa käynnistetään kuvausistunto. Kuvausistunnon aloittamisen yhteydessä palvelimelta pyydetään käyttäjän antaman näytenumeron kuvat, jos sellaisia on. Jos samalle näytteelle löytyy kuvia, ladataan ne ensin palvelimelta päätelaitteelle, jotta aiempaa istuntoa voidaan jatkaa. Istunto aloitetaan riippumatta siitä, onko yhteyttä palvelimeen vai ei. Tämä on välttämätöntä demokäytössä, jolloin kuvapalvelinta ei ole. Istunnossa otetut kuvat tallennetaan aina ensin päätelaitteen kiintolevylle JPEG-muodossa. Otetut kuvat lähetetään kuvapalvelimelle vasta, kun istunto lopetetaan. Koska kuvien lopullinen sijoituspaikka on palvelimella, se

päättää lopulliset tunnisteet kuvalle. Jos kuvia nimetään uudelleen palvelimella, käyttäjälle ilmoitetaan kuvien uudelleennimeämisestä lähetyksen jälkeen ponnahdusikkunalla.

Kuvat lähetetään palvelimelle HTTP-rajapinnan kautta. Asiakkaan verkkoympäristössä voi olla useita DisseCam-kuvausjärjestelmiä käytössä yhtä aikaa, joten rinnakkaisuusongelmat tulee ottaa huomioon. Vaikka kuvausjärjestelmillä ei kuvattaisi yhtä aikaa saman näytenumeron kuvia, palvelinyhteys voi vikatilanteessa olla poikki, jolloin kuvat joudutaan lähettämään palvelimelle myöhemmin. On myös mahdollista, vaikkakin erittäin epätodennäköistä, että joskus kahdelta DisseCam-päätteeltä lähetetään yhtä aikaa samaan näytteeseen liittyviä kuvia. Konflikti havaitaan palvelimella ja kuvien lähetys keskeytetään. Päätelaitteella pidetään kirjaa siitä, mitä kuvia ei olla vielä lähetetty palvelimelle. Käytännössä tämä on toteutettu siirtämällä kuvatiedostot toiseen kuvahakemiston alihakemistoon, kun kuvan lähetys on onnistunut. Tämä kuvien välimuistina toimiva hakemisto siivotaan tietyin asetustiedostossa säädettävin aikavälein.

Kuvapalvelimella kuvien tallentamiseen käytetään RethinkDB-tietokantaa. Kuvan tunnisteet tallennetaan tietokannassa samassa muodossa kuin tiedostonimessä. Kuvadata tallennetaan palvelimella edelleen JPEG-muodossa ja tietokantaan tallennetaan viittaus kuvatiedostoon.

Kuvien tallentamisen palvelinrajapinta ja -toteutus muuttuivat projektin aikana. Ensimmäisessä järjestelmän palvelintoteutuksessa kuvat tallennettiin samoilla tunnisteilla kuin millä ne oli lähetetty, ja konfliktitilanteissa vanhat kuvat ylikirjoitettiin. Tässäkään toteutustavassa ei havaittu suuria ongelmia koekäytössä, mutta riski kuvien katoamiseen oli olemassa. Käytännössä konfliktitilanne voi syntyä, jos samaa näytettä on kuvattu ainakin kahdella DisseCam-järjestelmällä. Uudessa toteutuksessa kuvapalvelin päättää kuvien lopulliset tunnisteet, vaikka sovellus luo tunnisteet jo kuvien tallennusvaiheessa.

4.3.4 Annotaatioiden tallentaminen

Annotaatiot päätettiin myös tallentaa suoraan kuviin, jotta niiden siirrettävyys olisi mahdollisimman hyvä. Tallennetut kuvat ovat tavallisia JPEG-kuvatiedostoja, joita voi halutessaan katsella millä tahansa kuvankatseluohjelmalla. Annotaatioiden tallennus pelkkinä kuvina on yksinkertaisempaa ja vaatii vähemmän suunnittelutyötä kuin esimerkiksi vektorigrafiikan tai oman tallennusformaatin käyttö. Annotaatioiden tallentaminen kuvina aiheuttaa myös sen, ettei annotaatioiden muokkaaminen jälkikäteen ole mahdollista. Alkuperäinen kuva kuitenkin tallennetaan aina, joten sen voi tarvittaessa annotoida uudelleen.

Yhdestä otetusta kuvasta tallennetaan tyypillisessä tapauksessa yksi annotoitu versio, jolloin tallennustilaa kuluu yleensä noin kaksinkertaisesti. Tietomäärät ovat kuitenkin pieniä: enimmillään muutamia megatavuja per kuva. Asiakkaat tallentavat käytännössä aina palvelimelle myös virtuaalisia näytelaseja, joiden koko on tyypillisesti muutamia satoja megatavuja per kuva. Tällaisissa datamäärissä esimerkiksi yhden näytteen makrokuvien

vaatiman kapasiteetin pienentäminen kymmenestä megatavusta viiteen megatavuun on suhteellisen merkityksetön optimointi, etenkin kun yhdestä näytteestä valmistetaan tyypillisesti kymmeniä näytelaseja.

4.3.5 Kameran etähallinta

Järjestelmässä kameraa hallitaan HTTP-protokollalla JSON-RPC-rajapinnan kautta, eli kameralle lähetetään JSON-muotoisia etäproseduurikutsuja [4]. Näin kameran hallinta onnistuu helposti asynkronisten funktiokutsujen avulla. Tärkeimmät kameralle lähetettävät komennot liittyvät objektiivin zoomin hallintaan ja kuvan ottamiseen. Kuva-asetuksille on kamera-moduulissa järkevät oletusasetukset, mutta kaikki asetukset voi asettaa haluamikseen asetustiedostossa, joka luetaan sovelluksen käynnistyessä. Esimerkit JSON-RPC-etäkutsusta ja vastauksesta on esitetty listauksissa 4.6 ja 4.7.

```
{
    "method": "actTakePicture",
    "params": [],
    "id": 1,
    "version": "1.0"
}
```

Ohjelma 4.6. *Esimerkki kameran actTakePicture-etäkutsun hyötykuormasta.*

```
{
    "result": [
        ["http://ip:port/postview/postview.jpg"]
    ],
    "id": 1
}
```

Ohjelma 4.7. *Esimerkki kameran actTakePicture-etäkutsun vastauksen hyötykuormasta.*

4.3.6 Mittausten mittakaavan määrittäminen

Kuvien ottamisen yhteydessä JPEG-tiedoston metatietoihin tallennetaan kuvan mittakaava JSON-muodossa. Mittakaavan tallentamiseen käytetään EXIF-metatietojen *MakerNote*-kenttää. Sovelluksen annotaatiotyökalut mahdollistavat janojen mittaamisen millimetreissä otetuista kuvista. Mittausten pituuden määrittäminen millimeterissä vaatii mittakaavan laskemisen kuvan ottamisen yhteydessä. Kamera asennetaan kiinteälle etäisyydelle kuvattavasta tasosta, joten järjestelmän mittakaava pitää myös kalibroida asennuksen yhteydessä.

Kun kameralla otetaan kuva, tallennetaan JPEG-kuvan EXIF-metatietoihin muun muassa objektiivin polttoväli ja digitaalisen zoomin taso. Digitaalinen zoom-kerroin on suhdeluku, joka kertoo kuvan suurennoskertoimen verrattuna alkuperäiseen. Optinen suurennosker-

roin täytyy laskea polttovälin avulla. Teoriassa optinen suurennoskerroin pitäisi pystyä laskemaan lineaarisesti objektiivin polttovälialueen ja zoom-tasojen vaihteluvälin avulla. Käytännössä tämä ei kuitenkaan käyttämämme objektiivin kanssa näytä toimivan, mikä johtuu testiemme perusteella siitä, että tarkentaminen lähelle pienentää havaittua polttoväliä. Kyseessä on ilmiö nimeltä ”focus breathing”, joka tarkoittaa polttovälin muuttumista tarkennuksen seurauksena [6]. Optinen zoom-kerroin päädyttiin laskemaan muokatulla yhtälöllä, jossa maksimisuurennos mitattiin manuaalisesti kuvaamalla tietty fyysinen etäisyys pikseleinä minimi- ja maksimisuurennoksilla ja laskemalla niiden suhde.

4.4 Käyttöjärjestelmän asetukset

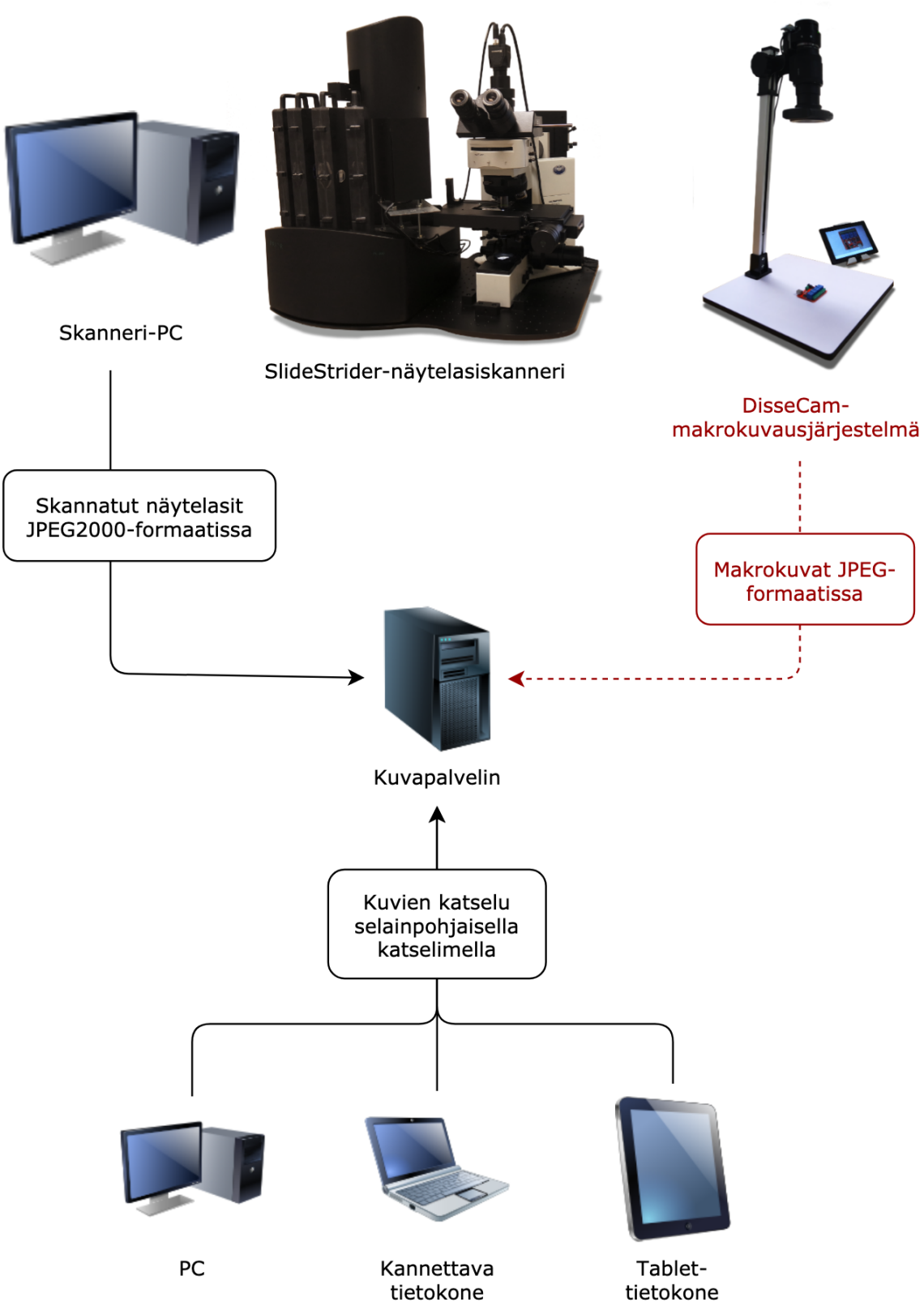
Kuvausjärjestelmän käyttöön vaikuttivat myös jotkin Windows-käyttöjärjestelmän asetukset. Virransäästöasetukset asetettiin minimiin, jotta sovellus toimii mahdollisimman sulavasti. Kuvausjärjestelmää ajettiin Windowsin ”määritetty käyttö”-tilassa, joka tunnetaan paremmin kioski-tilana. Tällöin sovelluksessa on ajossa yksi ohjelma, eikä Windowsin työpöytään tai muihin ohjelmiin pääse käsiksi. Kuvausjärjestelmä haluttiin yhdistää yhtä aikaa kameraan WiFi-yhteydellä ja säilyttää ethernet-yhteys sisäverkkoon. Windows-käyttöjärjestelmässä oletusastuksilla langaton yhteys katkaistaan, jos langallinen yhteys on kytketty. Tämä ominaisuus täytyy kytkeä pois päältä Windowsin rekisteriä muokkaamalla.

4.5 Integraatio olemassaoleviin järjestelmiin

DisseCam-kuvausjärjestelmän toteutukseen liittyy läheisesti integraatio Jilab Oy:n kehittämään SlideVantage-kuvapalvelinjärjestelmään, joka on esitetty kuvassa 4.13. Kuvassa ylävasemmalla on Jilab Oy:n kehittämä SlideStrider-näytelasiskanneri, jolla on mahdollista digitoida virtuaalisia näytelaseja JPEG2000-tiedostomuodossa. SlideVantage on selainpohjainen kuvien katseluympäristö, jossa kuvat on tallennettu kuvapalvelimelle. Kuvapalvelinkomponentti pilkkoo kuvat pienempiin osiin ja muuntaa ne selainten ymmärtämään JPEG- tai PNG-muotoon. Tässä työssä toteutettava DisseCam-makrokuvausjärjestelmä on esitetty kuvassa korostettuna yläoikealla. Kuvat lähetetään DisseCam-päätteeltä SlideVantage-kuvapalvelimelle, josta niitä voi katsella selainpohjaisella katselimella samaan tapaan kuin virtuaalisia näytelaseja. Käytännössä integraatio on toteutettu siten, että samaan näytteeseen liittyvien makrokuvien ja virtuaalisten näytelasien katseleminen samaan aikaan on mahdollista selainpohjaisella katselimella.

4.6 Variaatio obduktion dokumentointiin

Projektin aikana havaittiin, että järjestelmä sopii pienin muokkauksin ruumiinavausten eli obduktioiden dokumentointiin. Kamera korvataan tavallisella digitaalikameralla ja sovel-



Kuva 4.13. *Jilab Oy:n SlideVantage-kuvapalvelinjärjestelmä ja siihen liittyvät laitteet*

luksesta vaihdetaan live-näkymä yksinkertaisemmaksi kuvien lähetysnäkymäksi, jossa sovellus tarkkailee tietokoneeseen liitettäviä uusia USB-muistilaitteita, jollainen kamera myös on. Kun kamera liitetään USB-liitännällä järjestelmään, käyttäjältä kysytään ponnahdusikkunassa, haluaako hän lähettää kamerassa olevat kuvat palvelimelle. Vahvistuksen saatuaan sovellus lähettää kamerassa olevat kuva kuvapalvelimelle näytenumeron mukaan uudelleennimettynä. Onnistuneen lähetyksen jälkeen kamerassa olevat kuvat poistetaan. Annotointityökalujen käyttö ei ruumiinavauksessa ole erityisen hyödyllistä, joten niiden käyttöä ei ole integroitu työnkulkuun. Kameran ollessa käyttäjän käsissä myöskään annotaatiomittausten tekeminen ei ole mahdollista.

5 ARVIOINTI

Tässä luvussa arvioidaan projektin onnistumista ja tuloksena kehitetyn sovelluksen käyttökelpoisuutta. Kokonaisuutena projektia ei voida katsoa täysin onnistuneeksi, koska käytön sujuvuutta ei saatu halutulle tasolle.

5.1 Järjestelmän käyttökelpoisuus

Kuvausjärjestelmä annettiin asiakkaalle pilottikokeiluun tammikuussa 2016. Järjestelmää kehitettiin palautteen mukaan edelleen korjaamalla havaittuja puutteita ja vikoja. Syksyllä 2016 asiakas ilmaisi halunsa ostaa kaksi kuvausjärjestelmä laboratorioonsa. Vuoden 2019 tammikuun lopussa näillä kahdella DisseCam-järjestelmällä otettuja kuvia annotoidut kuvat mukaan lukien oli tallennettu asiakkaan kuvapalvelimelle hieman alle 26000 kappaletta. Käyttäjäpalautteen perusteella annotointityökaluja ja kuvien laatua on pidetty hyvinä, mutta pääasiassa kamerasta johtuvat ongelmat aiheuttavat käyttäjille harmia ja käyttömukavuus kärsii.

5.2 UWP-kehitysympäristön käytettävyys

UWP-kehitysympäristö yhdistettynä web-tekniikoiden kanssa osoittautui ongelmalliseksi valinnaksi. Visual Studio ohjelmointityökaluna on monipuolinen, mutta toisinaan epävakaa, eikä kaikilta osin sovellu hyvin web-tekniikoilla kehittämiseen. Esimerkiksi moduuliniputtimien käyttämistä Visual Studiossa ei ole huomioitu millään tavalla, mikä tekee niiden käyttämisestä haasteellista. Niputtimen kutsumiseksi ennen sovelluksen kokoonpanoa täytyy sen komento lisätä projektitiedostoon BeforeBuild-koukkuun. UWP-sovellukset on tarkoitettu kehitettäväksi Visual Studio -kehitysympäristöllä ja niitä ajetaan kehityksen aikana Visual Studion debuggerilla, joka ei projektin aikana osannut muuntaa moduuliniputtimen generoiman koodin rivinumeroita vastaamaan lähdekoodia. Windowsin ja UWP-ympäristön kanssa vastaan tulleita ongelmia on kuvattu lisää aliluvussa 5.3.3.

5.3 Haasteita ja ongelmia

Projektin aikana ongelmia ilmeni UWP-kehitysympäristön lisäksi erityisesti kameran toiminnassa. Tässä luvussa esitellään projektin aikana esille tulleita haasteita ja ongelmia.

5.3.1 Kameran toiminta

Sonyn kamera-rajapinta oli projektin aikana beta-asteella, eikä kameran etähallinta toiminut luotettavasti. Kameran viiveet vastauksissa etäkutsuihin vaihtelivat. Rajapintadokumentaatiosta oli myös hankala saada selville, millä tavalla rajapintaa tarkalleen tulisi käyttää. Kameran automaattista sammutusta ei voinut kytkeä pois päältä. Toisaalta kamera pysyy päällä niin kauan, kuin siihen on langaton yhteys ja päätelaitteen voi asettaa yhdistämään kameraan automaattisesti. Kameralle lähetettävät pyynnöt epäonnistuvat toisinaan, mutta kaikkiin virhetilanteisiin ei haluta reagoida. Esimerkiksi päivämäärän asettaminen kameraan onnistuu kameran käynnistymisen jälkeen kerran, jonka jälkeen se aina epäonnistuu.

Kehityksen yhteydessä havaittiin kameran ilmoittaman polttovälin virhemarginaali harmillisen suureksi. Käytännössä tämä huomattiin siinä, että otetun kuvan EXIF-metatiedoissa ilmoitettu polttoväli pysyi samana, vaikka kuvassa havaittu suurennos oli selvästi muuttunut. Valitettavasti parempaa tapaa mittakaavan laskemiseen ei projektin aikana löydetty. Todennäköisesti polttoväliä ei ole tarkoitettu käytettäväksi näin tarkoissa laskutoimituksissa, koska käytetyn objektiivin tiedoista löydy virhemarginaaleja otettujen kuvien metatiedoille.

5.3.2 Laitteiden saatavuus

Koska kuvausjärjestelmässä käytetyt laitteet ovat kuluttajaelektroniikkaa, niiden saatavuus riippuu valmistajasta. Etenkin Sonyn QX1-kameran saatavuudessa oli haasteita projektin aikana, eivätkä muut Sonyn kameramallit soveltuneet projektin tarpeisiin yhtä hyvin. Saatavuusongelmat olivat kuitenkin väliaikaisia, eivätkä aiheuttaneet suuria ongelmia.

5.3.3 Windows ja UWP

Projektin aikana törmättiin muutamiin UWP-ympäristön ja Windows-käyttöjärjestelmän vikoihin, jotka hankaloittivat järjestelmän kehittämistä. Kerran ohjelman virheellinen toiminta paljastui UWP-sovellusten käyttämän Chakra-moottorin ohjelmointivirheeksi [12]. Windows 10:n isommat päivitykset aiheuttivat ohjelman ajoympäristön muuttumisen, minkä seurauksena ohjelma ei enää toiminut oikein, jos ohjelmapaketti oli luotu eri käyttöjärjes-

telmän versiolla kuin mihin se asennettiin. Windows-käyttöjärjestelmän ”Creators’s update”-päivitys rikkoi WinJS:n *ListView*-komponentin [41]. Kyseessä oli ilmeisesti myös Chakra-moottorin ohjelmointivirhe, joka vaikuttaa olevan sittemmin korjattu. Microsoft myös lopetti WinJS-kirjaston kehityksen projektin aikana, eikä uusissa UWP-ohjelmointiesimerkeissä enää käytetä WinJS:ää. Vaikka kirjasto on avointa lähdekoodia, avoimia ongelmaraportteja on kymmeniä ja kehitys näyttää pysähtyneen, joten kirjaston käyttöä ei voi enää suositella. Aliluvussa 4.4 esitellyt Windows-käyttöjärjestelmän rekisteriä muokkaamalla tehdyt muutokset katosivat päivitysten yhteydessä, joten ne täytyi tehdä uudelleen.

5.4 Jatkokehitysideoita

Kuvausjärjestelmän kehityksen aikana tuli esille useita jatkokehitysideoita, joiden toteuttamiseen resurssit eivät riittäneet. Tässä luvussa on kuvattu olennaisimmat jatkokehitysideat lyhyesti.

5.4.1 Purkkinumero

Patologian laboratoriossa samaan näytteeseen liittyvät näytepalat on joskus jaettu useampaan purkkiin. Purkin numeron voi toki lisätä itse näytenumeron perään, mutta olisi hyvä pitää se näytenumerosta erillisenä. Istunnon aloituksessa olisi tästä syystä hyvä olla purkkinumeron valinta näytenumeron lisäksi.

5.4.2 Annotaatioiden tallentaminen erillään kuvista

Annotaatioiden tallentaminen erillään kuvista on ollut esillä useasti projektin aikana, mutta ominaisuutta ei ole toteutettu sen vaatiman työmäärän vuoksi. Ominaisuus voidaan toteuttaa esimerkiksi tallentamalla annotaatiot erillisinä osittain läpinäkyvinä PNG-kuvina tai itse rakenteellisessa formaatissa kuten JSON:ssa. Annotaatioiden eriyttäminen pienentäisi annotaatioiden tallentamiseen tarvittavaa tallennuskapasiteettia ja rakenteellisessa formaatissa tallennettuja annotaatioita voisi muokata jälkikäteen.

5.4.3 Kameran hakkerointi

Kamerassa ajettavan ohjelmiston voisi mahdollisesti myös korvata omallaan. Sonyn nykyisissä kameroissa käyttöjärjestelmänä on Sonyn muokkaama versio Googlen Android-käyttöjärjestelmästä, johon voi asentaa sovelluksia Sonyn omasta *Sony PlayMemories Camera App Store* -sovelluskaupasta. Hakkerit ovat onnistuneet takaisinmallintamaan sovellusten asentamisen Sonyn digitaalikameroihin ja purkamaan laiteohjelmapakettien salauksen. Tällä tavoin omien sovellusten asentaminen kameraan on mahdollista. [32]

5.4.4 Langallinen yhteys kameraan

Suurimpia ongelmia WiFi-yhteydellä toimivan kameran kanssa ovat live-kuvan laatu ja ajoittainen hitaus. Monista kameroista, myös Sonyn QX1-kamerasta, videokuvaa on mahdollista siirtää käyttäen HDMI-liitäntää, joka parantaisi ainakin live-kuvan laatua, mutta kameran hallinta täytyisi silti hoitaa muilla tavoin. Jos zoom-ominaisuuden hallinta tehtäisiin käsin, WiFi-yhteyden käytöstä voitaisiin tällä lähestymistavalla päästä kokonaan eroon.

Kameran vaihtaminen langalliseen olisi toinen mahdollisuus. Tarkoitukseen soveltuvia USB3-kameroita on saatavilla, mutta ongelma on sopivan objektiivin löytäminen. Sonyn digitaalikamerat käyttävät objektiivien hallintaan omaa protokollaansa, jonka takaisinmallintamalla voisi objektiivia ohjata erillisen ohjaimen avulla. Tämän jälkeen objektiivin käyttäminen minkä tahansa kameran kanssa olisi ainoastaan fyysisen ja optisen yhteensovittamisen ongelma.

5.4.5 Kuvaussovelluksen siirtäminen toiselle alustalle

UWP-ympäristön ja Windows-käyttöjärjestelmän ongelmista johtuen sovelluksen siirtäminen toiselle käyttöjärjestelmälle voi olla järkevää. Linux-käyttöjärjestelmän asetuksia voitaisiin hallita helpommin. Myös käyttöjärjestelmän ilmaisuus on etu, koska kioskimoodi on käytettävissä vain Windows 10 Pro -käyttöjärjestelmässä, jota ei kaikissa muuten tarkoitukseen sopivissa tietokoneissa ole. Eri alustalle siirtäminen tarkoittaa useiden ohjelman osien uudelleentoteuttamista, mutta sovelluksen suhteellisen pienestä koosta johtuen työmäärä eri olisi kovin suuri. Erityisen mielenkiintoinen vaihtoehto UWP-ympäristölle on Electron-sovelluskehys, joka perustuu Chromium-selaimeen ja Node.js-suoritusympäristöön [10].

6 YHTEENVETO

Työn tuloksena syntyi makrokuvausjärjestelmä, joka on mahdollista käyttää näytteen esikäsittelyvaiheen dokumentointiin. Järjestelmän hyviä puolia ovat kuvien laatu ja annotointityökalut. Suurimmat ongelmat liittyvät järjestelmän luotettavuuteen ja kaytettävyyteen, joiden pääasialliset syyt ovat kameran toiminnan epävarmuus ja hitaus. Nämä ongelmat voidaan korjata vain vaihtamalla kamera toiseen tai muokkaamalla kameran laitteistoa itse. Ongelmat kyseenalaistavat järjestelmän käyttökelpoisuuden, joten tältä osin projektia ei voida pitää täysin onnistuneena.

Yksi työn tavoitteista oli UWP-ympäristön arviointi työpöytäohjelmoinnissa. UWP-ympäristö web-tekniikoiden kanssa todettiin haastavaksi ympäristöksi työpöytäsovellusten toteuttamiseen. Päänvaivaa aiheuttivat UWP-alustan ja Windows 10 -käyttöjärjestelmän viat sekä kehitysympäristön epävakaus. WinJS-alustan tarjoamat käyttöliittymäkomponentit auttoivat sovelluksen käyttöliittymän luomisessa, mutta kirjaston kehityksen pysähtyminen ei innosta jatkamaan sen käyttöä. Microsoft vaikuttaa jättävän web-tekniikat UWP-sovellusten kehityksessä taka-alalle. Monet UWP-ympäristön käytössä tässä projektissa kohdatuista haasteista todennäköisesti rajoittuvat web-tekniikoihin, koska esimerkiksi C#-kieltä käytettäessä UWP-sovellusten kehitys ja suoritusympäristö ovat hyvin erilaisia.

Projektin aikana hybridisovellusten kehittämiseen löydettiin toinen vaihtoehto: Electron-sovelluskehys, jota käytettiin toisen projektin toteutuksessa. Jos DisseCam-sovellus tulevaisuudessa päätetään toteuttaa uudelleen Electron-sovelluskehystä käyttäen, ohjelmointikoodista osan voi uudelleenkäyttää pienin muutoksin, koska kameran ja kuvien hallintaan käytetyt moduulit sisältävät vain vähän UWP-kohtaista koodia. Lisäksi allekirjoittaneelle on projektin aikana kertynyt hyödyllisiä tietoja ja taitoja web-tekniikoiden käytöstä sovellusohjelmoinnissa, joita voi soveltaa tulevissa projekteissa.

Projektin sivutuotteena järjestelmästä toteutettiin variaatio ruumiinavausten dokumentointiin. Kuvausjärjestelmän mukauttaminen onnistui yllättävän pienillä muutoksilla ja kameraongelmat poistuivat, koska kameraa ei käytetty WiFi-yhteyden kautta. Valitettavasti DisseCam-järjestelmän jatkokehitys näyttää epävarmalta, koska patologian laboratoriot Suomessa siirtyvät käyttämään suurten toimijoiden digitaalipatologian kokonaisratkaisuja, eikä DisseCam-järjestelmälle siten näytä ainakaan Suomessa olevan kovin suurta kiinnostusta.

LÄHDELUETTELO

[1] *About npm*. URL: https://docs.npmjs.com/about-npm/ (viitattu 11. 01. 2019).

[2] *Adobe - Flash Player*. URL: https://get.adobe.com/flashplayer/about/ (viitattu 11. 01. 2019).

[3] *Anatomia – Wikipedia*. URL: https://fi.wikipedia.org/wiki/Anatomia (viitattu 07. 12. 2018).

[4] *API references for Camera Remote API beta*. saatavissa: https://developer.sony.com/file/download/sony-camera-remote-api-beta-sdk-2/. Sony. 2017.

[5] *Asynchronous Programming - UWP App Developer | Microsoft Docs*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/uwp/threading-async/asynchronous-programming-universal-windows-platform-apps (viitattu 07. 11. 2018).

[6] B. Atkins. *Fabric.js Javascript Canvas Library*. URL: http://www.bobatkins.com/photography/technical/focus_breathing_focal_length_changes.html (viitattu 16. 02. 2019).

[7] K. Brockschmidt. *Programming Windows Store Apps with HTML, CSS and JavaScript*. 2. painos. Microsoft Press, 2014.

[8] *Controls and patterns for UWP apps*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/uwp/design/controls-and-patterns/ (viitattu 14. 02. 2019).

[9] *Document Object Model (DOM)*. W3C. 2005. URL: https://www.w3.org/DOM/ (viitattu 27. 11. 2018).

[10] *Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS*. URL: https://electronjs.org/ (viitattu 13. 11. 2018).

[11] E. Elliott. *Programming JavaScript Applications*. 1. painos. O'Reilly, 2014.

[12] *ES6 constructor returns class instead of object · Issue #1496 · Microsoft/ChakraCore*. Microsoft. URL: https://github.com/Microsoft/ChakraCore/issues/1496 (viitattu 19. 10. 2017).

[13] *Fabric.js Javascript Canvas Library*. URL: http://fabricjs.com/ (viitattu 13. 11. 2018).

[14] J. Garrett. *Ajax: A New Approach to Web Applications*. 2005. URL: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/ (viitattu 10. 07. 2017).

[15] *Glossary of Terms*. Digital Pathology Association. URL: https://digitalpathologyassociation.org/glossary-of-terms_1 (viitattu 22. 11. 2017).

[16] *gPhoto - Doc :: Remote controlling cameras*. URL: http://gphoto.org/doc/remote/ (viitattu 13. 11. 2018).

[17] *How to use the Camera Remote API beta*. saatavissa: https://developer.sony.com/file/download/sony-camera-remote-api-beta-sdk-2/. Sony. 2017.

[18] *HTML5, Hybrid or Native Mobile App Development*. IBM. 2012. URL: ftp://ftp.software.ibm.com/software/fr/pdf/whitepapers/Worklight-HTML5_Hybrid_Native_Mobile_App_Development.pdf (viitattu 15. 11. 2018).

[19] *Hybrid | Definition of Hybrid by Merriam-Webster*. URL: https://www.merriam-webster.com/dictionary/hybrid (viitattu 14. 01. 2019).

[20] *Hybridisovellukset helpottavat mobiiliapplikaatioiden kehittämistä*. URL: https://www.itewiki.fi/blog/2015/08/hybridisovellukset-helpottavat-mobiiliapplikaatioiden-kehittamista/ (viitattu 13. 11. 2018).

[21] *Java Software | Oracle*. URL: https://www.oracle.com/java/ (viitattu 11. 01. 2019).

[22] F.-M. Leong ja A. Leong. Digital Photography in anatomical pathology. *Journal of Postgraduate Medicine* 50.1 (2004), 62–69.

[23] M. Mäkinen, O. Carpén, V.-M. Kosma, V.-P. Lehto, T. Paavonen ja F. Stenbäck. *Patologia*. 1. painos. Duodecim, 2012.

[24] *Microsoft Dropping 'Metro' Name in Windows 8*. URL: http://uk.pcmag.com/windows-8-preview-release-date-news-features/61523/news/microsoft-dropping-metro-name-in-windows-8 (viitattu 18. 11. 2018).

[25] *Microsoft Silverlight*. URL: https://www.microsoft.com/silverlight/ (viitattu 11. 01. 2019).

[26] *Node.js - Wikipedia*. URL: https://en.wikipedia.org/wiki/Node.js (viitattu 12. 01. 2019).

[27] *Promise/A+ - An open standard for sound, interoperable JavaScript promises*. URL: https://promisesaplus.com/ (viitattu 29. 07. 2018).

[28] *Promise.then method*. Microsoft. URL: https://docs.microsoft.com/en-us/previous-versions/windows/apps/br229728(v=win.10) (viitattu 22. 09. 2018).

[29] A. Rauschmeyer. *Exploring ES6*. URL: http://exploringjs.com/es6 (viitattu 20. 03. 2019).

[30] T. Reenskaug. *Models-Views-Controllers*. 1979. URL: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (viitattu 20. 02. 2019).

[31] T. Reenskaug. *MVC XEROX PARC 1978-79*. URL: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html (viitattu 21. 02. 2019).

[32] *Reverse engineering Sony PlayMemories Camera Apps*. URL: https://github.com/ma1co/Sony-PMCA-RE (viitattu 10. 12. 2018).

[33] M. Slaoui ja L. Fiette. Histopathology Procedures: From Tissue Sampling to Histopathological Evaluation. *Methods in molecular biology (Clifton, N.J.)* 691 (2011), 69–82. DOI: 10.1007/978-1-60761-849-2_4.

[34] *Spectrum - The No Hassle jQuery Colorpicker*. URL: https://bgrins.github.io/spectrum/ (viitattu 22. 11. 2018).

[35] Y. Sucaet ja W. Waelput. *Digital Pathology*. 2. painos. Springer, 2014.

[36] S. Suvarna, C. Layton ja J. Bancroft. *Bancroft's Theory and Practise of Histological Techniques*. 7. painos. Churchill Livingstone Elsevier, 2013.

[37] B. Tetu, D. Wilbur, L. Pantanowitz ja A. Parwani. Teleconsultation. *Digital Pathology: Historical Perspectives, Current Concepts & Future Applications*. Toim. K. Kaplan ja L. Rao. Vol. 1. Springer International Publishing, 2016, 55–70.

[38] V. Tuominen. Virtual Microscopy: Design and Implementation of Novel Software Applications for Diagnostic Pathology. Tohtorinväitöskirja. University of Tampere, 2012.

[39] *Universal Windows Platform apps*. Microsoft. URL: https://en.wikipedia.org/wiki/Universal_Windows_Platform_apps (viitattu 11. 11. 2018).

[40] *Using Chakra for Scripting Applications across Windows 10*. Microsoft. 2015. URL: https://blogs.windows.com/msedgedev/2015/05/18/using-chakra-for-scripting-applications-across-windows-10/ (viitattu 28. 11. 2018).

[41] *UWP, after Windows 10 Creators Update (15063.138) Exception in ListView Item, removeEventListener · Issue #1665 · winjs/winjs*. URL: https://github.com/winjs/winjs/issues/1665 (viitattu 29. 11. 2018).

[42] W. Westra, R. Hruban, T. Phelps ja C. Isacson. *Surgical Pathology Dissection: An Illustrated Guide*. 2. painos. Springer, 2003.

[43] *What's a Universal Windows Platform (UWP) app?* Microsoft. URL: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide (viitattu 07. 11. 2018).

[44] *Windows 10 UWP App lifecycle - Windows UWP applications*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/uwp/launch-resume/app-lifecycle (viitattu 02. 01. 2019).

[45] *WinJS.Application Namespace*. Microsoft. URL: https://docs.microsoft.com/en-us/previous-versions/windows/apps/br229774(v=win.10) (viitattu 29. 04. 2019).

[46] *WinJS.Utilities Namespace*. Microsoft. URL: https://docs.microsoft.com/en-us/previous-versions/windows/apps/br229783(v=win.10) (viitattu 29. 04. 2019).