Universitetet i Oslo
Institutt for informatikk

Konfigurering og evaluering av et "low budget" videoredigerings-system for distribuert sanntidsvideo

Hovedfagsoppgave IFI

Henning Kulander

Henning Kulander
<hennikul@ifi.uio.no>

6. august 2004

Sammendrag

I denne oppgaven vurderes videoredigeringssystemer, protokoller og videokodeker opp mot hverandre, og et system basert på åpen kildekode som tar video fra en maskin på nettet og redigerer videostrømmen i sanntid utvikles.

Med et system for redigering av sanntidsvideo over et nettverk menes her et system hvor sanntidsvideo sendes fra et kamera koblett til en maskin over et nettverk til en annen maskin som redigerer og viser video umiddelbart uten mellomlagring. Tenkte bruksområde er videoeffekter på en konsert, overvåkingskamera eller videokonferanse i visualiseringsrom.

Oppgaven legger vekt på problemer som introduseres når video skal overføres gjennom ett nettverk, som valg av protokoller for overføring, båndbredekrav og forsinkelse. Disse problemene løses ved å se på tilgjengelige systemer. Forskjellige kodeker er testet opp mot hverandre ved å bruke dem til komprimering av videostrømmer fra tjeneren til videoredigeringsprogrammet. Båndbreddebruk og forsinkelse er målt, kvalitet og ressursforbruk vurdert.

Graphics Environment for Multimedia (GEM) er et videoredigeringssystem for sanntidsvideo, basert på Pure Data (PD), som gir brukeren stor frihet til å kombinere forskjellige objekter til å lage kunstneriske oppsett eller annen behandling av video. GEM brukes i dag av flere kunstnere. Å kunne redigere video direkte fra nett i GEM åpner for nye og spennende muligheter.

QuickTime Broadcaster og QuickTime Streaming server er en kraftig kombinasjon som muliggjør enkelt oppsett av videotjener med ulike kodeker. Oppgaven har sett på en rekke innstillinger og funnet en kombinasjon som gir lav forsinkelse, særlig ved bruk av MJPEG som kodek i Broadcaster.

Gjennom arbeidet med oppgaven er det erfart at utvikling med systemer basert på åpen kildekode kan være vanskelig grunnet manglende dokumentasjon og manglende implementasjon. Vurdering av systemer basert på åpen kildekode med tanke på videre utvikling er vanskelig fordi det er vanskelig å bedømme kvaliteten på koden før den blir forsøkt brukt. Selv om selve systemet har mange brukere kan den ønskede funksjonaliteten være mangelfull dersom denne er lite brukt.

Det er også erfart at det finnes bra systemer basert på åpen kildekode, hvor koden er ryddig og velprøvd, og utviklerne er mer enn villige til å hjelpe dersom problemer oppstår, og som tar imot tilbakemeldinger og prøver å finne løsninger på problemene.

Innhold

Forord

Denne versjonen har med endringer basert på tilbakemeldinger fra veileder på kapittel 1-3, samt ny konklusjon og abstract.

I den endelige versjonen vil forordet snakke om hvem som kom med ideen til oppgaven, bruken i kunst osv.

1 Innledning

Gjennom 80- og 90-tallet foregikk det en revolusjon innen sanntidsredigering av lyd. Derimot innen video har ikke en tilsvarende revolusjon funnet sted. Det som nå finnes av verktøy kjøres på dyre, kraftige maskiner og bruker tilsvarende dyr programvare. Bruken begrenser seg derfor til fjernsynsproduksjon og store konserter. Lite arbeid har blitt gjort for å gjøre slike verktøy tilgjengelig for folk flest.

1.1 Bakgrunn

Rundt 1995 begynte det å bli vanlig med multimedia PCer, mye hjulpet av Microsofts Windows 95, som innførte en multimedia infrastruktur. Standard PCer fikk CD-ROM, Lydkort og et skjermkort som kunne vise 16-bit eller 24-bit farger. Det ble også laget standardkrav for multimedia maskiner, samtidig som det ble vanlig at hjemme- og kontor-PCer oppfylte disse kravene. PCen hadde nå fått samme muligheter til lyd- og bildebehandling som Commodore Amiga og Apple Macintosh hadde hatt flere år tidligere.

Etter hvert ble PCene kraftige nok til å brukes til avansert lydredigering med flere kanaler og musikkproduksjon. Med en vanlig hjemme-PC ble kvaliteten god nok for hjemmebruk, og ved å investere i et lydkort til 6-8000 kroner, ble studio-kvalitet en mulighet. Dette har ført til at det i dag er mulig for “Hvermansen” å produsere egen musikk og å distribuere denne enten ved live-opptredener med digitale effekter og lydsamples, eller ved digital distribusjon av den ferdige musikken.

Omtrent hvert tyvende år kommer en ny teknologi som revolusjonerer lyd og musikk industrien eller dens økonomi. I fortiden har disse teknologiene primært påvirket lydstudioene og platetrykkeriene, men den siste revolusjonen skjer i hjemmene til folk[1]. Tidligere var musikkproduksjon kun mulig i avanserte lydstudioer med utstyr flerkanals innspillingsutstyr med en pris fra 50000 dollar. I dag kan det samme gjøres med en datamaskin til under 2000 dollar. Også for kunstnere har dette åpnet mange spennende muligheter som sanntids lydinstallasjoner publikum kan interagere med.

Innen video har ikke en tilsvarende revolusjon funnet sted enda. Men digitale kamera kan i dag kjøpes for 5000 kroner og oppover, og de kommer med standardutstyr for å kobles til en datamaskin. Programvaren begynner å bli kraftig nok til å legge på avanserte effekter, klippe videoen fornuftig og legge på lyd. Nye PCer kommer med ferdig installert videoredigeringsprogramvare og mulighet for å koble på DV-kamera via FireWire.

Men redigeringen kan fremdeles ikke gjøres i sanntid. Det leverandørene av sanntids-redigeringsprogramvare til PC mener med sanntidsredigering, er programvare som klarer å spille av et ferdig redigert prosjekt med effekter i sanntid, uten å beregne effektene før avspilling. Overføring fra kamera til PC, redigering på PC og overføring tilbake tar fremdeles lang tid. *Denne oppgaven definerer sanntids videoredigering til video som redigeres slik at resultatet vises samtidig som opptaket gjøres, og hendelsen opptaket registrerer, finner sted.* Det er umulig å vise resultatet samtidig som opptaket gjøres, det vil alltid være en forsinkelse. Hvor stor forsinkelsen kan være avhenger av bruksområdet. I et system med interaktiv dialog må forsinkelsen være mindre enn 250 ms[2], dersom systemet krever leppesynkronisering mellom bilde og lyd må forsinkelsen være mindre enn 40 ms[3][4] i forhold til lyden. I en konsertsituasjon hvor det redigerte bildet vises bak artisten vil dette si at forsinkelsen ikke kan overstige 40 ms. dersom leppene er synlige, i situasjoner hvor interaktivitet er viktig men lyd ikke behandles må forsinkelsen være mindre enn 250 ms.

Profesjonelle miljøer bruker i dag ekstremt kraftige maskiner som Onyx og Tezro seriene fra SGI for å få til slik sanntids-videoredigering. Disse maskinene koster fra rundt 200.000,- og oppover, og benyttes derfor kun av TV-selskaper og store band til sanntidseffekter på sportssendinger, digitale tv-studioer og andre sanntidseffekter som brukes live på TV. De brukes ikke av “Hvermansen”.

En datamaskin til flere hundre tusen kroner er et hinder for de fleste amatører. De som ikke blir klarer å skaffe en maskin, blir gjerne stoppet når de skal skaffe programvare. Denne er ofte like dyr eller dyrere enn maskinen den kjøres på. Til TV-sendinger i dag brukes dyr programvare fra blant andre firmaene Descreet og Vizrt for sanntidsredigering av video. Programmer fra Descreet er veldig populære til både sanntidsredigering (Flint, Inferno), og postproduction redigering med produkter som Fire og Smoke. Vizrt er et firma med røtter i TV2 som leverer egen programvare for TV-sendinger i hele verden. Firmaet Snell&Wilcox leverer programvare som Magic Dave som kan mikse video live.

Spørsmålet er om dette kan gjøres dersom totalkostnaden for maskiner og programvare ikke skal overskride det “Hvermansen” kan klare å betale, et absolutt maks er 50000 kroner. Det må undersøkes hva som kan oppnåes dersom flere PCer kobles sammen i et nett og videoredigerings programvaren distribueres ut på flere maskiner. Forhåpningen er at sanntidsvideoredigering til VJ-bruk, konsertbruk og kanskje også redigering av TV-sendinger vil være mulig slik at det kan komme en videoredigerings revolusjon liknende den som har vært innen lydredigering[5]. Ideen er at kostnaden for redigering av video kan minimeres ved å bruke et nettverk av rimelige standard maskiner til redigeringen, i stedet for én dyr og kraftig spesialmaskin.

Det er gjort lite forskning innen sanntidsredigering av distribuert video med rimelige maskiner. Men trenden er at rimelige maskiner kan gjøre mer og mer ettersom regnekraften dobles hver attende måned. Redigeringssteknikker som tidligere trengte maskinvare assistanse kan nå gjøres med ren programvare [6]. Ved å koble flere maskiner sammen og distribuere arbeidsoppgavene i nettet kan det være mulig å ytterligere øke funksjonaliteten. Redigeringen kan gjøres mens innholdet er i nettet fordi datakraften i nye maskiner øker hele tiden[7].

En annen grunn til at lydredigering er populært er at det er enkelt. Programvaren er bygd opp av et enkelt, visuelt grensesnitt. Ved videoredigering er det helt essensielt å ha et visuelt brukergrensesnitt som stimulerer brukerens intuisjon [8]. Dette er en utfordring som også må løses før folk flest vil ta teknologien i bruk.

1.2 Teknologiske utfordringer

De største problemene med et distribuert system for sanntidsredigering av video er forsinkelse i nettet og i sendermaskinen samt begrenset tilgjengelig båndbredde. Dette er krav som grupperes under “Quality of Service”. Dersom lyd og bilde sendes og redigeres separat, kan forsinkelsen gjøre at lyden kommer raskere fram enn bildet. Hjernen er vant med at bildet kommer først siden lys beveger seg raskere enn lyd, noe som gjør at en situasjon hvor lyden kommer før bildet er slitsomt og forvirrende. Kombinasjoner av media må derfor behandles med et temporært og romlig forhold slik at leppesynkronisering mellom lyd og bilde og koordinering av forskjellige videokilder stemmer[9].

I en konsertsituasjon, hvor bildet redigeres samtidig som lyden spilles direkte til publikum med neglisjerbar forsinkelse, kan lite gjøres for å sikre synkronisering utover å få forsinkelsen ned til et minimum. Dersom lyden også redigeres i samme system, kan standardteknikker benyttes for å få bedre synkronisering av lyd og bilde, men det innebærer at lyden forsinkes like mye som bildet. Kravet om sanntidslevering gjør at det innføres en øvre grense for hvor mye forsinkelse hvert ledd innfører. Ved en videostrøm med 25 bilder i sekundet vil denne grensen være på 40 ms. (1 sekund / 25 bilder pr. sekund). Problemet er at enkel lydbehandling ofte ligger godt under denne grensen mens tung bildebehandling ligger tett opp mot den, dermed forsinkes bildene i forhold til lyden. Det vil dessuten være flere ledd i et distribuert system, minstekravet er et kamera, en tjener som sender ut video, et nettverk og en klientmaskin som tar mot video og redigerer. Alle disse leddene kan legge på opp til 40 ms. forsinkelse uten at kravet om sanntid brytes. I tillegg er det vanlig at disse komponentene igjen er delt opp i mindre logiske enheter, som videokort med buffer, komprimeringsoprogramvare, tjenerprogramvare, nettverksstack i operativsystemet, nettverkskort for eksempel.

Minimal forsinkelse i alle ledd er nødvendig for å dekke kravet til sanntid, en forsinkelse på 40 ms. er for stor til at leppesynkronisering er mulig.

En fullkvalitets PAL bildestrøm krever 44 megabyte (MB), eller 354 megabit (Mbit), i sekundet (s) ((768x576 punkter) x 32 bit x 25 bilder). Et standard lokalnett klarer i dag 100 eller 10 megabit i sekundet. Et 100Mbit/s nett gir under en fjerdedel av den båndbredden som kreves. Det er ikke lenger umulig å få tak i 1Gbit/s nett, det begynner allerede å bli standard på Apple-maskiner. Med 1Gbit/s nett mellom klient og tjener er båndbredden nok til å sende to fullkvalitetsstrømmer.

Komprimering er en teknikk som brukes for å sende mer informasjon enn båndbredden tilsier. Ett problem med komprimering er at det innfører et nytt ledd med forsinkelse, siden komprimering krever mye datakraft. Et annet problem er at de komprimeringsalgoritmene som gir høyest kompresjon, fører til tap av kvalitet, særlig når strømmen må dekomprimeres og komprimeres for hvert ledd i en kjede. Det finnes teknikker som tar vare på informasjon om komprimeringen i hvert ledd for å unngå tap av kvalitet i hver komprimering, men da er det begrenset hvor mye som kan gjøres med videoen uten at denne informasjonen blir foreldet.

Et sanntidsvideoredigeringssystem bør inneholde følgende funksjonalitet[5]:

- **Framebuffer** er et område i minnet som kan holde på ett bilde. Det brukes for å kombinere bilder fra forskjellige kilder slik at de blir synkronisert. Framebufferet bør ha funksjonalitet som keying og valgfri forsinkelse. Det bør også være mulig å få resultatet fra en redigering inn i et framebuffer.

- **Keyere: Luminans, chroma, ekstern matte** Keying er en effekt som brukes for å kombinere to videosignal. Keyeren ser etter en bestemt egenskap i det øverste bildet og fjerner alle punktene i bildet som har denne egenskapen slik at det underliggende bildet synes gjennom. Egenskapen kan være lysstyrke (luminans) eller farge (chroma) som med blåskjerm. Alternativt kan keyingen gjøres ved å bruke en egen kilde som matte, dette er oftest et bilde som er sort der bildet skal være gjennomsiktig og hvit ellers.

- **Colorization** brukes for å endre fargene i et videosignal.

- **Displacement mapping, warping og embossing**. Dette teknikker for å flytte punktene i et bilde slik at resultatet blir forvridd. Warping er generell flytting av referansepunkter i bildet for å forvri det, displacement mapping bruker et bilde for å beskrive hvordan bildet skal forvris og embossing bruker et bilde for å legge til følelse av dybde.

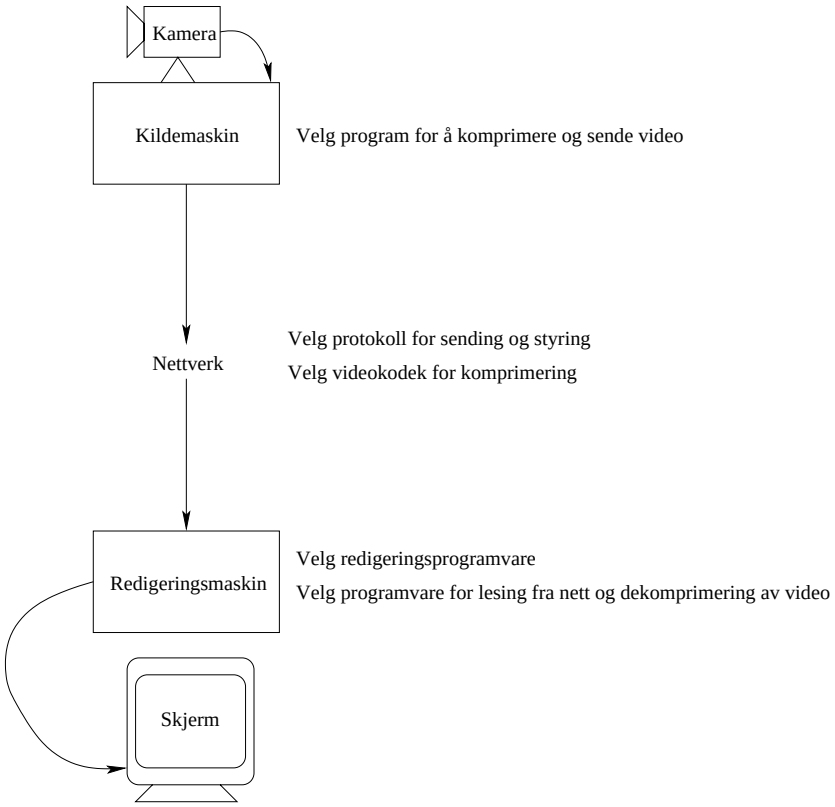
- **DVE Resize, rotation og position** brukes for å skalere, rotere og endre posisjon på video i forhold til fremvisningsmedia.
- **Noise generation** Legger til støy i et bilde. Dette kan enten være tilfeldig støy, eller kontrollert støy for eksempelvis å få videosekvensen til å se gammel og slitt ut.
- **Sequencer** brukes for å spille ferdiglagrede videosekvenser.
- **Mix/wipe** brukes for å lage en effekt i overgangen mellom to videokilder.
- **Outline** genererer en ramme rundt videobildet.
- **Equalize** balanserer fargene i bildet slik at det er like mange piksler for hver lysstyrke i histogrammet til bildet.
- **Representasjon av lyd** brukes for å lage et bilde som gjengir egenskapene til lyden. Det vanligste er oscilloskop som viser lydbølgen og fourieranalyse som viser frekvensene i lyden.
- **Autopaint** er effekter som genereres automatisk uten interaksjon fra brukeren.

Forsinkelsen fra kamera til fremvisning bør være så lav som mulig, men ikke overstige 250 ms. Båndbreddebruken bør være så lav at det er mulig å kjøre flere samtidige strømmer i et 100Mbit/s lokalnett, en øvre grense på 10Mbit/s vil muliggjøre 10 samtidige strømmer og brukes som øvre grense her.

1.3 Mål og metode

Hva skal oppnåes? Målet med denne hovedfagsoppgaven er å lage et system for sanntidsredigering av distribuert video. Det viktigste i denne oppgaven blir å velge kommunikasjonssystemet mellom de forskjellige delene av systemet, samt design, implementasjon og testing av dette systemet. Se figur 1 på neste side for en illustrasjon som viser delene av systemet, og hva som skal velges og implementeres.

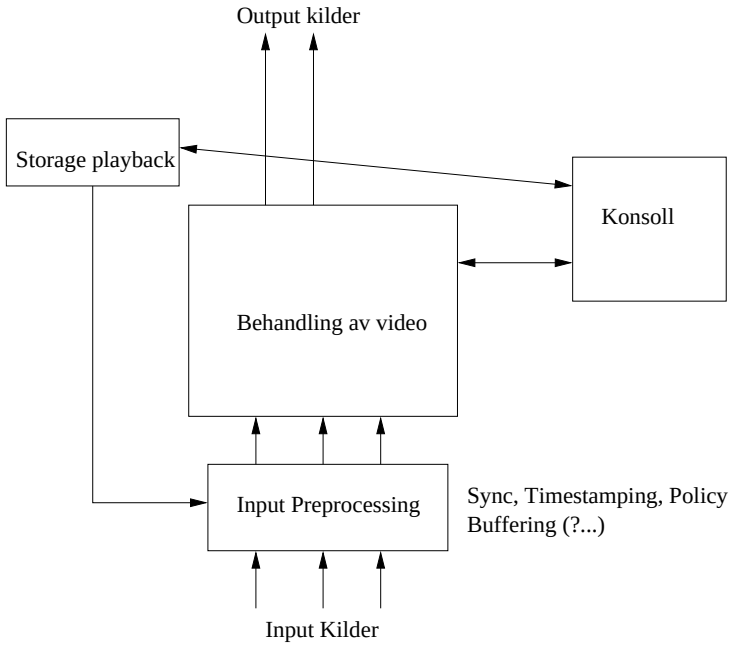
Et minimalt distribuert redigeringssystem kan settes opp ved at én eller flere kilder (tjenere) sender ut video over nett til en maskin som redigerer videostrømmene (klient) og genererer output. Et mer avansert system vil i tillegg ha klienter som også kan fungere som kilder ved å sende resultatet fra redigeringen ut på nett. I denne oppgaven blir et minimale redigeringssystemet utviklet, det mer avanserte eksempelet vil ta for lang tid. En slik kommunikasjon



Figur 1: Tegning av systemet som utvikles i denne oppgaven.

krever et kommunikasjonssystem mellom maskinene og et mellomformat for videostrømmene. Mellomformatet inneholder video samt nødvendig metadata som bildefrekvens, oppløsning, fargerom, bildeformat, protokoll enkapsulering og annet [10] som er nødvendig for å tolke videosignalet. Det endelige systemet vil bruke mellomformatet som output fra en redigering og videre som input til en ny redigering. Med et "distribuert redigeringssystem" mener denne oppgaven nettopp dette, et redigeringssystem hvor redigeringen er distribuert til flere maskiner. Se figur 2 på neste side for en tegning av en komponent i et slikt system. Systemet kan sammenlignes med systemet i [9].

Hensikten med denne oppgaven. Hensikten med denne oppgaven er å undersøke om det er mulig å lage et slikt system med rimelig og lett tilgjengelig programvare som kjører på rimelige datamaskiner slik at "Hvermansen" har muligheter til å skaffe det. Oppgaven undersøker hva som finnes av tilgjengelige verktøy for redigering av sanntidsvideo, hvilke protokoller og videokodeker som



Figur 2: Tegning av en enkel versjon av det komplette videosystemet. Input preprosessering og behandling av video blir hovedtemaene i denne oppgaven.

kan benyttes for å sette sammen et mellomformat og hva som kan brukes som tjener i systemet.

Den praktiske delen av oppgaven kobler forskjellige programvarekomponenter sammen slik at kommunikasjon mellom tjener og redigeringsmaskin er mulig. Systemet vil testes så for å verifisere at redigering faktisk er mulig. Deretter måles systemets ytelse med tanke på forsinkelse og båndbreddekrav som nevnt over.

Det legges vekt på bruk av åpen kildekode og åpne protokoller under kommunikasjonen for å lettest mulig kunne utvide systemet med ferdige, eller utvidbare, komponenter.

Avslutning av oppgaven. Når det er satt opp et redigerinssystem som tar video fra en maskin, redigerer den og legger på noen effekter avsluttes utviklingen. Når dette er gjort vil kvaliteten måles med hensyn til båndbreddekrav og forsinkelse og bildekvaliteten vurderes.

Metode. Metoden som benyttes er å utvikle et system som overfører video fra en maskin til en annen som redigerer i sanntid. Det måles hvor lang tid

det tar fra videoopptaket gjøres til resultatet er klart fra den andre maskinen. Båndbreddeforbruk og bildekvalitet måles også.

For å komme fram til det utviklede systemet gjennomføres det et survey hvor litteratur og andre informasjonskilder undersøkes for å finne ut hva som finnes av slike systemer i dag og ideer til hvordan ting kan gjøres. Hva som finnes av teknologi som protokoller og mellomformater for overføring av video over nett undersøkes. Det er også et mål å finne ferdig eller delvis ferdig programvare som kan benyttes i systemet slik at minst mulig må utvikles fra bunnen av.

Oppgavens oppbygning. Oppgaven begynner med å finne et redigeringsverktøy som kan brukes til dette prosjektet. Deretter ser oppgaven på hvilke protokoller som kan brukes for å overføre video over et nett. Så går oppgaven over til å se på hvilke styringsprotokoller som finnes for å styre videostrømmen og hvilken videokodek som skal brukes for komprimering av video. I hvert kapittel gjøres det et valg av hvilken protokoll som er best egnet til å benyttes i utviklingen av systemet. Når videoredigeringsprogramvare og protokoller er valgt blir det gjort forsøk på å finne delvise eller komplette implementasjoner av disse som kan bygges sammen til et komplett redigeringssystem.

Når alt er valgt, bygges det opp en demonstrasjonmaskin hvor systemet implementeres og testes. De ulike komponentene som nå er valgt, integreres med hverandre og en nettverksstrøm settes opp mellom tjener og klient.

Til slutt testes systemet for å måle den totale forsinkelsen og båndbreddeforbruket. Målet er at systemet skal kunne brukes i en sanntidssituasjon og med utstyr "Hvermansen" kan få tak i. Båndbreddeforbruket er viktig for å muliggjøre bruk av lett tilgjengelige eller eksisterende nett, forsinkelsen er viktig for å få til sanntid. Det konkluderes med om resultatet var bra nok og hva som må forbedres. Til slutt foreslåes noen oppgaver til videre forskning.

1.4 Liknende arbeid

Saarland universitetet i Saarbrücken, Tyskland, jobber med et system de kaller "Network-Integrated Multimedia Middleware for Linux" [11]. De har laget et system hvor multimediautstyr med nettverksfunksjonalitet kan kobles sammen med hverandre og brukes som ett produkt. De jobber også med kommersialisering av et av sluttproduktene deres.

I [12] definerer de fordeler med deres teknologi, som er basert på distribuerte multimediamaskiner og programvare komponenter som integreres i en felles flytgraf.

1.4 Liknende arbeid

Arkitekturen er åpen og baserer seg ikke på noen bestemt teknologi eller mellomvare, men åpner for fleksibel bruk av forskjellige nettverks- og mellomvareteknologier. Fordelene er hetrogenitet, mulighet for adskilt kommunikasjon til data og kontroll, optimaliserte kommunikasjonsstrategier og hendelsesbehandling.

Dette systemet er et nivå høyere enn det som er beskrevet i denne artikkelen. Det vil trolig la seg gjøre å la NMM benytte seg av teknologien utviklet her for å koble sammen enheter støttet av NMM og enheter støttet av et distribuert redigeringssystem.

2 Videoredigeringsprogramvare

Dette kapittelet omhandler valget av programvare som skal brukes til selve videoredigeringsbiten. Det første avsnittet omhandler noe av det som er tilgjengelig av videoredigeringsprogramvare for profesjonelt bruk. Neste avsnitt omhandler programvare som er mer aktuell for bruk i denne oppgaven, og hva som kan forventes av disse systemene.

Tilgjengelige gratis/rimelige systemer vurderes til slutt opp mot hverandre. Hovedkriteriene her er at systemet skal være rimelig og enkelt i bruk. Åpen kildekode ansees som en stor fordel siden systemet antakelig må utvides med ekstra funksjonalitet for å få inn video fra nett.

2.1 State of the Art innen videoredigering

Det finnes i dag flere forskjellige verktøy for å redigere videostrømmer. De mest populære i film- og TV-industrien koster flere hundre tusen kroner, og kjøres på hardware som er tilsvarende dyr. I Norge er firmaet Discreet en kjent leverandør av programvare som Flame og Smoke. De leverer også Frost, som er beregnet på redigering av live-sendinger[13]. Firmaet Avid leverer programmer som Avid|DS HD 4.0 med en pris på ca.$300,000[14]. For kringkasting leverer firmaet Global Streams programmet GlobalCaster Studio[15], dette er en løsning som består av både både hardware og software.

TV2 utviklet til å begynne med egen programvare for sanntidsredigering av video til bruk under direktesendinger. Dette muliggjorde et moderne preg på direktesendingene som den gang var unikt i Norge. Utviklingen ble senere lagt over til Pilot Broadcast Systems, et datterselskap av TV2. Etter en sammenslåing med Peak Software Technologies og oppkjøp av RT-SET heter de nå Vizrt[16]. De er i dag ledende innen kringkastings løsninger og brukes nå også av NRK, CNN og andre store kanaler.

Disse systemene er alt for dyre for “hvermansen”, som kan være en uavhengig kunstner, VJ eller hjemmebruker, med ønske om å drive med sanntids videoredigering. Her må rimeligere alternativer benyttes. Det kan ikke forventes at et gratis system skal ha samme funksjonalitet som et system til flere hundre tusen kroner, men denne oppgaven viser at det er mulig å finne systemer som kan brukes og utvides for å oppnå ønsket funksjonalitet.

2.2 Tilgjengelig gratis/rimelig programvare

Et søk på nettet resulterer i er del gratis og rimelig programvare, som enten kan brukes direkte, eller kan brukes som rammeverk for et videoredigeringssystem. Ingen av disse har funksjonalitet eller brukervennlighet som tilsvarer den produktene til f.eks. Discreet har, men spesielt for kunstnere vil ofte muligheten til å utvide være et sentralt behov, her kan programvare med åpen kildekode stille veldig sterkt siden hva som helst kan gjøres med det så lenge den tekniske kompetansen er tilstede.

Programvare behandles i dette kapittelet med henblikk på funksjonalitet, kostnad, brukergrensesnitt og plattform:

Funksjonalitet. Med funksjonalitet menes hvilke funksjoner programmet kan utføre og hva det kan brukes til. For eksempel er det her interessant å se hvilke videoredigeringseffekter som kan legges til en videostrøm, hvilke videoformater som kan benyttes og hvor godt systemet fungerer med sanntids video. Funksjonaliteten kan vurderes opp mot funksjonalitetskravet fra kravspesifikasjonen.

Kostnad. Kostnadene tilknyttet et datasystem er delt opp i innkjøpskostnadder og driftskostnadder, tilsammen utgjør dette “Total Cost of Ownership”. Det er enkelt å finne innkjøpskostnadene på systemet, men det er vanskelig å beregne hvilke ekstra kostnadder utvikling av ekstra funksjonalitet, opplæring og annet vil medføre.

Brukergrensesnitt. Det er to viktige ting å se på når brukergrensesnitt vurderes: Hva som finnes og hva som kan lages.

Det beste er om det finnes et ferdig brukergrensesnitt som fyller brukerens behov. Da kan applikasjonen taes i bruk uten å utvikle noe selv. Det som da må vurderes er hvor avansert dette brukergrensesnittet er og hvor enkelt det er for en ny bruker å ta det i bruk.

Dersom det ikke finnes et ferdig brukergrensesnitt, eller dersom brukergrensesnittet ikke fyller behovene, må et eget utvikles. Det som da må vurderes er hva slags brukergrensesnitt det er mulig å lage til den applikasjonen som vurderes. Plattformen applikasjonen kjører på vil sette begrensninger for hva som kan lages av brukergrensesnitt, og hvor enkelt det vil være å lage det.

Plattform. Med plattform menes her kombinasjonen av operativsystem og maskinvare. For eksempel er Linux på ix86 en plattform, mens Linux på MIPS64 en annen. Plattformen er i stor grad med på å bestemme kostnadene til systemet. Både operativsystem og maskinvare koster penger og avhengigheter mellom operativsystem og maskinvare kan begrense valgmulighetene slik at systemet blir dyrere.

De første systemene denne oppgaven tar for seg er videoredigeringssystemer med brukergrensesnitt som er beregnet for sluttbrukere. Oppgaven nevner også noen rammeverk, eller APIer (Application Program Interface), for å utvikle videoredigeringssystemer og andre multimediaapplikasjoner. De inneholder funksjonalitet for blant annet videoredigering, men brukeren må selv utvikle brukergrensesnitt. I denne oppgaven ville det være for tidkrevende å utvikle et system fra bunnen, derfor er det kun gjennomført tester med de komplette videoredigeringssystemene.

Systemene vurderes opp mot hverandre og det systemet som dekker størst del av funksjonalitetskravet til sluttbrukeren, samtidig som det er rimelig nok for "Hvermansen" velges. Det er viktig at systemet er utvidbart slik at det kan legges inn støtte for lesing av video fra nettverk.

2.2.1 Max/NATO

Max[17] er i utgangspunktet et system for lydredigering hvor brukeren kan sette opp objekter med ulik funksjonalitet, og koblinger mellom disse, som samarbeider om å løse den endelige oppgaven. Den resulterende grafen bestemmer funksjonaliteten brukeren kan benytte seg av, og også hvilke brukergrensesnitt, som MIDI keyboard eller mus, som kan brukes. Ideen bak Max er at det skal være mulig å styre "alt med alt", det vil si at alle typer objekter kan brukes til å styre alle andre typer objekter. For eksempel kan et MIDI keyboard styre en laserdisk spiller eller en mus styre QuickTime avspilling.

Objektene løser forskjellige oppgaver i systemet. De defineres ved støttet inputformat, prosesseringsegenskaper og outputformat. Eksempler på objekter er tellerobjekter, forsinkelsesobjekter og mer avanserte objekter som kan legge en konvolusjonsmatrise på et bilde. Objektene kobles sammen slik at ut-strømmen fra ett objekt settes til inngangen på et annet objekt som støtter den type data. Figur 3 på side 15 viser et skjermbilde fra Max/NATO, boksene i bildet representerer objekter.

Max uten ekstramoduler er beregnet på å styre MIDI. Men med ekstramoduler utvides funksjonaliteten. Max/MSP gjør det mulig å redigere digital lyd og

Max/NATO muliggjør redigering av video.

NATO[18] er et tillegg til Max beregnet på redigering og generering av video. Det muliggjør videoredigering styrt av MIDI-utstyr eller lyd. NATO bygger på QuickTime rammeverket til Apple. Det er beregnet bruket sammen med Quicktime video og bruker QuickDraw og QuickTime-VR til å lage effekter. NATO tilfører 116 nye objekter til Max[19].

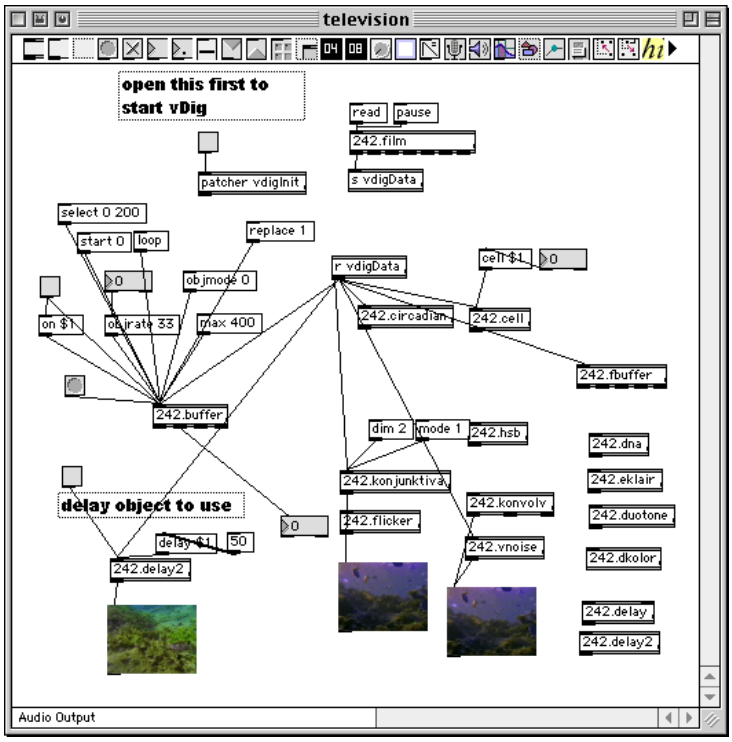
Funksjonalitet. Av filformater støtter NATO blant annet Quicktime, DV, AVI, MPEG og PNG. NATO kan lese filer både fra lokal disk og fra nett. Siden NATO bygger på QuickTime vil funksjonaliteten til NATO vokse i takt med funksjonaliteten i QuickTime.

NATO er et avansert verktøy til redigering. MIDI hendelser kan bindes opp mot hendelser i NATO for å styre redigeringen, og det finnes mange filtre. Av interessante filter nevnes komposisjon, som setter sammen flere videostrømmer, skalering, rotasjon, mapping på tredimensjonale objekter (QuickTime 3D eller OpenGL) og fargejusteringer.

Kostnad. NATO krever at brukeren har Max installert. Max koster $295. Pakken Max/MSP utvider Max til å kunne behandle audio i tillegg til MIDI, den koster $495. I tillegg til dette må NATO kjøpes til en pris av $549.54. Utover dette kommer kostnaden ved at det kjører på PowerPC/Mac OS plattformen. Kildekoden er ikke tilgjengelig, men det er mulig å lage egne moduler (eksterne objekter) til Max i C/C++ ved hjelp av et medfølgende bibliotek.

Brukergrensesnitt. Brukergrensesnittet til Max er et brukergrensesnitt for å lage et redigeringssystem. Dette gjøres ved å plassere objekter og koble disse sammenbaserer seg på å tegne streker mellom som vist i figur 3 på neste side. På denne måten bygges dataflyter mellom objektene i en konfigurasjon, hvert objekt utfører en enkel del av redigeringen. Max har også objekter som leser fra tastatur, MIDI-keyboard, mus osv. Ved å legge slike objekter i grafen vil sluttbrukeren kunne bruke redigeringssystemet på en intuitiv måte.

Plattform. Max kjører på Windows og Macintosh. NATO kjører kun på Macintosh med Mac OS.



Figur 3: Brukergrensesnittet til Max/NATO. Brukergrensesnittet bygger på prinsippet om sammenkoblede objekter. Objektene kobles sammen til trær, og informasjon sendes gjennom koblingene.

2.2.2 PD/GEM

PD[20] (PureData) er system med åpen kildekode som ligner på Max/MSP. Det baserer seg på samme prinsipp som i Max hvor objekter, tilsvarende objektene i Max, plasseres på et "lerret" og koblinger, som representerer dataflyt, trekkes mellom disse. Også her kan egne objekter utvikles i C[21]. Det finnes ulike typer objekter: kodemoduler som utfører en jobb og forskjellige typer parameter-objekter som styrer kodemodulene.

I utgangspunktet var PD også beregnet for lydredigering. GEM[22] (Graphics Environment for Multimedia) er en tilleggspakke til PD som er utviklet for å brukes til videoredigering. GEM baserer seg på OpenGL, og lar brukeren lage OpenGL objekter. OpenGL er et grafikk bibliotek utviklet av Silicon Graphics Inc (SGI) i 1992. OpenGL er mest kjent for 3D grafikk, og er den mest portable plattformen for 3D grafikk, men kan også brukes til 2D grafikk og video. OpenGL egner seg derfor som en portabel plattform for grafikk. Dette er grunnen til GEMs

valg av OpenGL og muliggjør at GEM virker både på Windows, Linux og IRIX. Det finnes egne objekter for å lese video og å legge video som teksturer på OpenGL objekter. Video kan styres på flere måter fra PD. For eksempel kan lyden styre video eller omvendt. Kildekoden er også tilgjengelig, funksjonalitet lar seg derfor utvides.

Med PD/GEM følger det med en rekke ferdiglagde eksempelgrafer som kan brukes for å lære hvordan PD/GEM fungerer. Eksemplene som omhandler video viser at GEM inneholder mange vanlige 3D objekter og lar brukeren definere egne. Brukeren kan styre egenskaper som lyssetting, oppløsning på objekter (antall triangler objekter er bygd opp av) og teksturer. Normal video fra videokortet legges på et objekt som en tekstur. Dersom vanlig 2D video ønskes, legges video som tekstur på et flatt objekt med normalen rett mot brukeren.

Funksjonalitet. Videoredigeringen i GEM gjøres ved å sette opp OpenGL-objekter, mye av funksjonaliteten er derfor basert på 3D-objekter og transformasjoner av disse.

Det finnes en rekke objekter laget for å redigere teksturer. Egendefinerte konvolusjonsmatriser gjør at veldig mange effekter kjent fra bildebehandling kan brukes. Teksturinformasjon kan leses på pikselnivå og kan brukes til ting som for eksempel farge på 3D objekter. Det er enkelt å sette opp masking ved enten å bruke multiplisering av to bilder eller ferdiglagde maskeobjekter som gjør objektet gjennomsiktig utenfor masken. Dette kan brukes til effekter som bluescreen. Fargen på teksturene kan justeres og tekst kan legges til.

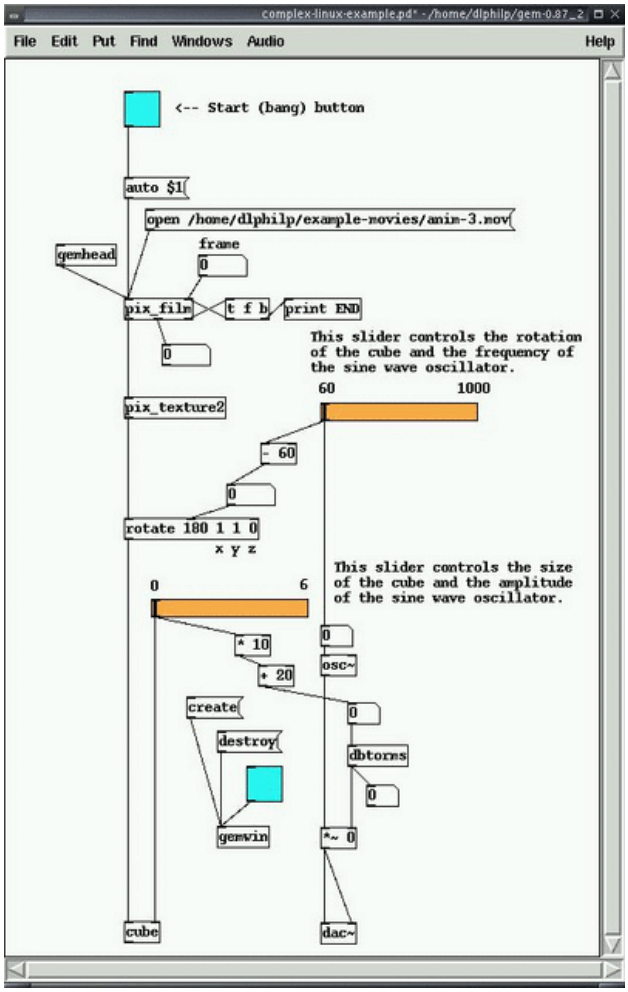
Kostnad. Både PD og GEM er gratis. Begge kan kjøre på både Windows og Linux, samt IRIX. Dersom PD/GEM kjøres på Windows eller Linux blir innkjøpskostnaden for programvare på under 1000 kroner, som er kostnaden for operativsystemet. Maskinen må være kraftig nok til å beregne de effektene som er ønsket og den må støtte OpenGL. OpenGL krever hardware-støtte for å gå raskt, dette innebærer at skjermkort-driverene må oversette OpenGL kall til kall på hardware rutiner i skjermkortet slik at de kan bli utført der [1]. Et skjermkort med OpenGL-drivere kan anskaffes til under 1500 kroner. Videoredigering vil være mulig på en maskin til under 10000 kroner.

Kildekoden til både PD og GEM er fritt tilgjengelig og kan endres.

Brukergrensesnitt. Brukergrensesnittet ligner veldig på brukergrensesnittet til Max (Se figur 4 på neste side for et eksempel). PD støtter også MIDI, så å styre

[1] Noen skjermkort støtter OpenGL direkte

2.2 Tilgjengelig gratis/rimelig programvare



Figur 4: Brukergrensesnittet til PD/GEM. I likhet med Max/NATO er PD/GEMs brukergrensesnitt bygget på sammenkoblede objekter. Dette eksemplet lager en roterende kube med videotekstur.

ting fra MIDI kan muliggjøre veldig brukervennlige systemer ved å bruke eksternt utstyr beregnet på oppgaven brukeren vil løse.

Plattform. GEM ble utviklet for OpenGL under SGI IRIX, men kjører nå også under Microsoft Windows 95/NT eller nyere og under Linux. Hardware støtte for OpenGL er viktig dersom ytelsen skal bli akseptabel, det kreves også et lydkort. Dersom IRIX benyttes som operativsystem må også SGI hardware benyttes da IRIX kun kjører på SGI maskiner. Windows kan kjøres på ix86 maskinvare, Linux kan kjøres på flere typer maskinvare.

2.2.3 FreeJ

FreeJ[23] er et prosjekt med åpen kildekode som går ut på å lage gratis VeeJay[2]-software, i første omgang under Linux. Meningen er at det skal være et digitalt instrument for video livesets med sanntidsvisning av flerlags video med kjedede effekter rett på en skjerm.

FreeJ benytter seg av et lag konsept ("layers"). Hvert videokilde sees på som et lag, hvert lag kan ha en kjede med effekter. En Videokilde kan være enten et video4linux-kort eller en "Video for Windows"-fil. I hvert lag settes det så opp en kjede med filter som behandler video på forskjellige måter.

Foreløpig finnes det ikke så mange effekter, men mange er planlagt. Generelt sett virker det som om hele prosjektet er i en ganske tidlig fase. En av planene er å integrere effektene til EffecTV[24]. Siden begge er prosjekter med åpen kildekode er ikke dette problematisk, og EffecTV har mange avanserte effekter ferdige.

Funksjonalitet. Funksjonaliteten er per dags dato veldig begrenset. Den baserer seg på videoinput fra "Video4Linux" drivere i kjernen, og støtter ikke streaming. Det finnes bare en håndfull effekter, og disse er ikke så utrolig spennende. Fordelen er at systemet er åpent slik effekter, streaming osv. kan legges til av brukeren.

I motsetning til GEM og NATO støtter ikke FreeJ redigering av lyd. Det er planlagt å lage visuelle effekter styrt av lyd i en fremtidig versjon.

Kostnad. Selve programmet er fritt tilgjengelig. Det kjøres under Linux, så det er ingen utgifter knyttet til innkjøp av programvare. Maskinkravet er det samme

[2] Se ordforklaringer bakerst.

som til PD/GEM med to unntak: Det er ikke krav til OpenGL-støtte, men det kreves at et videokort med video4linux-drivere er installert.

I tillegg kommer kostnaden med å utvikle ønsket funksjonalitet dersom det ikke er tid til å vente på naturlig utvikling. Kildekoden er fritt tilgjengelig og kan endres.

Figur 5: Brukergrensesnittet til FreeJ. FreeJ styres med tastetrykk. På skjermen kan brukeren valgfritt se statusinformasjon om hvilke filtre som kjøres.
Bildet er hentet fra `http://freej.dyne.org/`

Brukergrensesnitt. Grensesnittet er svært enkelt. Det baserer seg på overlay av tekst på videobildet for å forklare hva som skjer(Se figur 5 for et eksempel). Dette kan slåes av, og det som styrer redigeringen er tastetrykk. Tastetrykk velger effekter, setter opp kjeder av effekter osv.

Plattform. Plattformen her er x86 PC med MMX-støtte i prosessoren og Linux som operativsystem. Det er mulig systemet blir portet til andre plattformer senere.

2.2.4 EffecTV

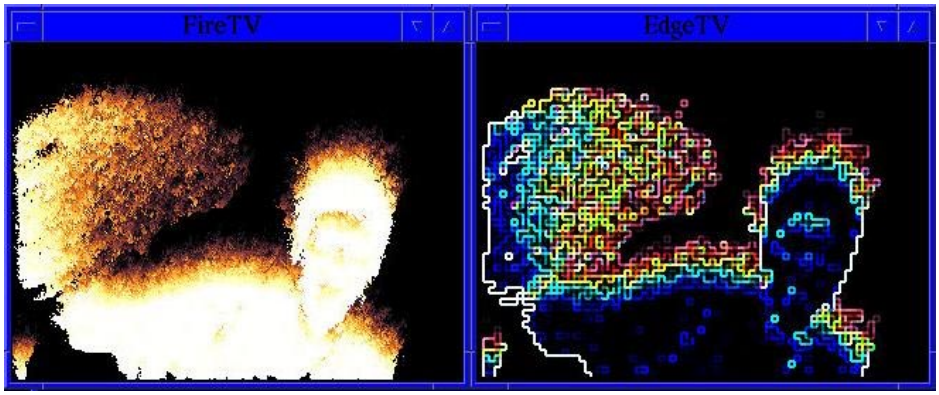
EffecTV[24] er et lite prosjekt med ganske mye utvikling. EffecTV er en sanntids video “effector”, det vil si et program legger på effekter i sanntid. EffecTV har innebygget en hel del effekter, og det kommer stadig nye.

Funksjonalitet. Det eneste som finnes av funksjonalitet i EffecTV er muligheten til å bytte mellom effekter og å justere noen parametere til disse. EffecTV kan kun legge til en effekt av gangen og er mer å regne som et leketøy for å legge på en effekt enn et videoredigeringssystem.

Det er mulig å kjede sammen effekter ved å bruke en video-loopback driver, en video4linux-driver som lar output fra ett program brukes som input i et annet program. Det vil si at video output fra en instans av EffecTV lager output som via loopback-driveren brukes om input i en annen kjøring av EffecTV. Dette er egentlig mer en funksjonalitet i operativsystemet enn i EffecTV siden det må kjøres en instans av EffecTV for hver effekt.

Kostnad. EffecTV er fritt tilgjengelig. Som med FreeJ er det ingen kostnader forbundet med programvare bortsett fra utvikling av funksjonalitet som ikke allerede finnes. Kostnad for maskinvare tilsvarer også FreeJ.

Kildekoden er fritt tilgjengelig og kan endres.



Figur 6: Brukergrensesnittet til EffecTV. I likhet med FreeJ styres EffecTV med tastetrykk. Den eneste tilbakemeldingen EffecTV gir er resultatet av filteret brukeren har valgt. Her brukes resultatet av det venstre bildet som input til filteret i det høyre bildet.
Bildet er hentet fra `http://effectv.sourceforge.net/`.

Brukergrensesnitt. Brukergrensesnittet er basert på at brukeren får se video som er kjørt gjennom ett filter. Ved å trykke kjente taster på tastaturet kan effekten som kjøres byttes til en annen og parametere til kjørende effekt kan endres. Utover det er det ikke noe synlig brukergrensesnitt. Figur 6 viser et eksempel på hvordan

output kan se ut. Her er output fra det venstre vinduet brukt som input til det høyre vinduet.

Plattform. EffecTV kjører på x86 Linux med MMX. I tillegg krever EffecTV et video4linux kompatibelt videokort i likhet med FreeJ. EffecTV kan også kjøres på PlayStation2 med Linux.

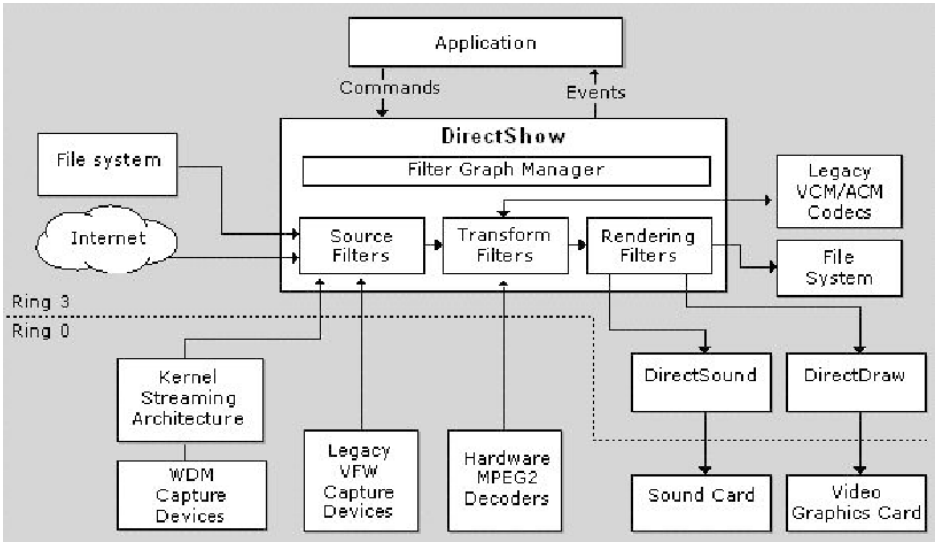
2.2.5 Microsoft DirectShow

Microsoft har bygget inn i Windows-plattformen et lagdelt mellomvaresystem for multimedia kalt DirectShow[25], beregnet for utvikling av applikasjoner for generering, redigering og visning av multimedia[26]. DirectShow er tett knyttet mot Microsofts Windows operativsystem.

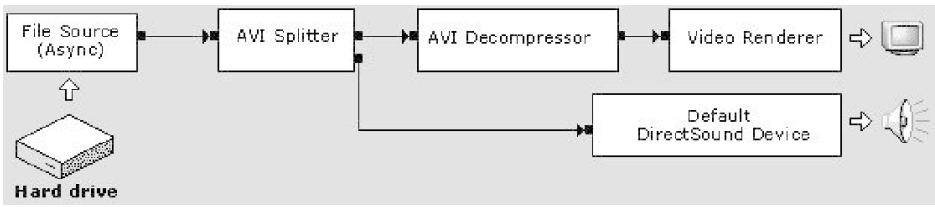
En DirectShow applikasjon består av en rekke logiske multimediakomponenter, også kalt filtre[27]. Filtrene kan ha pinner for input og output. Filtre med pinner kun for output kalles kilde filter (Source filter), filtre med pinner kun for input kalles visnings filtre (Rendering filter) og filtre med både output og input pinner kalles transformasjons filtre (Transform filter). Filtrene kobles sammen i en filtergraf som kan bygges manuelt eller automatisk av en GraphBuilder basert på input media. Figur 7 på neste side illustrerer oppbygningen av en DirectShow applikasjon.

Filtrene prosesserer data som kommer på input pinnene og sender resultatet til output pinnene. For at filtrene skal kunne kobles sammen med hverandre må pinnene være kompatible, det vil si at de støtter samme format (for eksempel kan en MPEG demultiplekser kobles sammen med en MPEG dekoder fordi demultiplekseren sender en MPEG strøm ut, mens en MPEG dekoder vil ha en MPEG strøm inn). Når filtrene er koblet sammen benyttes et push og pull grensesnitt for transport.

En stor fordel med å bruke filtre, som er så sterkt knyttet til operativsystemet som i DirectShow, er at de lett kan benytte seg av spesialisert maskinvare til å gjøre mye av jobben. Dagens datamaskiner har vanligvis mulighet for å gjøre ting som invers cosinus transformasjon, konvertering mellom fargeformat som YUV og RGB og skalering i skjermkortet. Dette avlaster prosessoren slik at den har mer ledig kapasitet til redigering og annet, noe som gjør at sluttbrukeren kan klare seg med en mindre prosessor. Filtrene gjør det også enkelt å endre en applikasjon til å lagre resultatet til disk i stedet for å vise på skjerm. Det som må gjøres er å bytte ut et filter. Se figur 8 på neste side for en eksempelapplikasjon som spiller .avi filer og viser på skjerm og sender lyd til høyttalere.



Figur 7: Filtertyper i DirectShow. Bildet viser flyten i et DirectShow system, og illustrerer hvilke filtertyper som finnes. Legg merke til hvordan DirectShow kan dra nytte av forskjellig maskinvare.
Bildet er hentet fra [26].



Figur 8: Eksempel på DirectShow applikasjon. Denne applikasjonen leser en avi (Video for Windows) fil fra disk, deler den opp i lyd og bilde som dekodes og vises på skjerm og spilles ut gjennom lydkortet i maskinen.
Bildet er hentet fra [26].

Et viktig begrep i DirectShow er MediaSample. Dette er et beholderobjekt eller buffer for media. Her lagres data sammen med tidsstempling, media type, lengde og annen informasjon som trengs innad i grafen og muliggjør synkronisert visning. Egne filtre kan lages som COM objekter.

DirectShow har vært i bruk siden Windows95, og er derfor godt testet. Det brukes i dag av veldig mange video- og lydapplikasjoner i Windows.

Funksjonalitet. Funksjonaliteten til DirectShow er bundet opp mot hvilke filtre som er tilgjenglige. Funksjonaliteten kan således bygges ut ved å legge til filtre enten ved å kjøpe spesiallagde filtre, eller ved å lage egne. DirectShow Editing Services er et tillegg til kjernen i DirectShow som muliggjør ikkelineær redigering av video. Her ligger det innebygget sceneskifte effekter ("whipeouts"), keying funksjonalitet og annet. Video Mixer er også innebygget i siste versjon, som kun finnes for Windows XP.

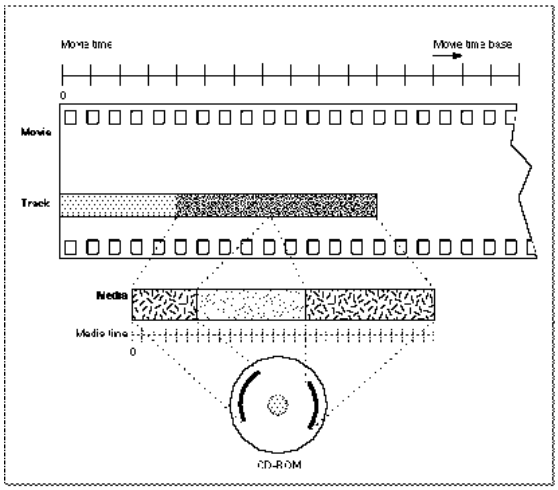
Kostnad. DirectShow følger med DirectX som igjen følger med Windows og kan lastes ned gratis fra Microsofts hjemmeside. For å utvikle egne applikasjoner som bruker DirectShow trengs DirectX SDK. Denne kan også hentes gratis ned fra Microsofts hjemmesider. I tillegg trengs en C++ kompilator, Microsoft anbefaler sin egen Visual C++ kompilator. Siden DirectShow kun fungerer i Windows, er Windows operativsystemet også et krav. Men den største kostnaden ligger i utvikling av selve redigeringssystemet siden DirectShow kun er et rammeverk og ikke et ferdig program.

Brukergrensesnitt. DirectShow har ikke noe brukergrensesnitt siden det kun er et rammeverk for å utvikle applikasjoner. Det følger med et program ved navn GraphEdit, som kan brukes for å generere en filtergraf, og teste denne. Grafen laget i GraphEdit er ikke ment å brukes i en ferdig applikasjon. Brukergrensesnittet brukeren presenteres med må lages av applikasjonsutvikleren. Siden applikasjonen må lages i Windows er brukergrensesnittet begrenset til hva som er mulig å lage der.

Plattform. DirectShow er bundet til Windows plattformen. Det er mulig å benytte noe av DirectShow-funksjonaliteten i Linux på ix86 hardware gjennom Wine (DivX playere i Linux gjorde dette tidligere), men å lage et fullstendig redigeringssystem kan kun gjøres i Windows.

2.2.6 Apple QuickTime

Apples QuickTime[28] er standard multimediaplattform på Macintosh plattformen. Det finnes også til Windows. QuickTime er et mer komplekst system enn DirectShow fordi det ikke bare definerer en komponentmodell, men også definerer et filformat og en datastruktur. Det de fleste kjenner QuickTime for er nok at trailere til de fleste filmer blir lagt ut som QuickTime filer.

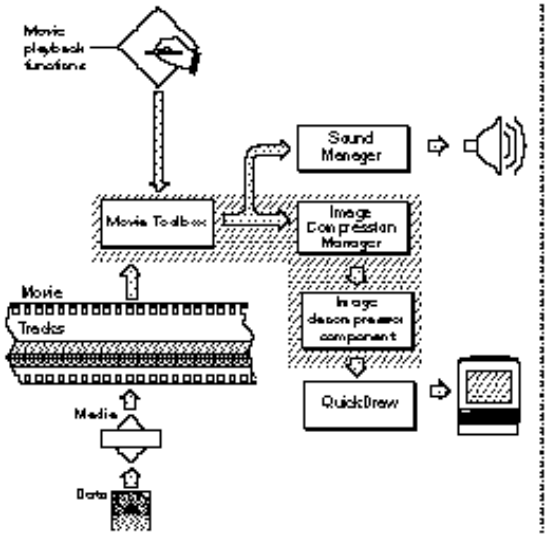


Figur 9: QuickTime's datastruktur. Film i bunnen med flere spor lagt ut i tids dimensjonen. Sporene holder referanser til media som for eksempel ligger på disk. Illustrasjonen er hentet fra `http://developer.apple.com/`.

QuickTimes datastruktur er en tredelt hierarkisk struktur. I bunnen ligger Film (Movie). Dette er en beholder som inneholder parallelle datastrømmer kalt Spor (Tracks). Film definerer også tiden. Sporene holder kontroll på de forskjellige lyd og video sporene. Sporene har en gitt lengde, rate, de har referanser til andre spor og referanse til media. Media ligger inni sporene og inneholder de ekte dataene. Her defineres mediatype, datatype, kvalitet, språk og så videre. Figur 9 illustrerer datastrukturen.

QuickTime støtter en rekke datatyper. Video, bilder, vektor grafikk, sprites, samplet lyd, MIDI, 3D objekter, panorama (QuickTime VR) og tekst. QuickTime filformatet er et beholderbasert filformat med en hierarkisk filstruktur bygget på datastrukturen forklart over.

QuickTime-systemet er bygget opp av komponenter som illustrert i figur 10 på neste side. Komponentene styres av en komponentbehandler (Component Manager) og løser forskjellige oppgaver i systemet. En film-verktøykasse (Movie Tool-



Figur 10: QuickTime's komponentstruktur. QuickTime leser video fra media, en film-verktøykasse finner ut hva som må gjøres med filmen og aktiverer andre komponenter.
Illustrasjonen er hentet fra `http://developer.apple.com/`.

box) gir høynivå tjenester for filmer. I tillegg har QuickTime en kompresjonsbehandler (Compression Manager) som støtter mange kompresjonsalgoritmer som plugins. Andre komponenter gjør ting som å komprimere, ta opp lyd og video, behandle media, styre klokka, utveksle data osv. I QuickTime versjon 4 og nyere er det også mulighet for å spille av videostrømmer fra nettet over RTP protokollen (se 3.3.3 på side 47).

QuickTime definerer et tidskoordinatsystem som er sentralt i QuickTimes virkemåte[29]. Dette tidskoordinatsystemet er et system hvor avstanden mellom tidspunktene defineres til å være $\frac{1}{n}$ sekunder, hvor *n* ofte er bestemt av bilderaten til video, eller samplingsraten til lyden. Denne informasjonen brukes for å synkronisere forskjellige kilder som lyd og bilde.

QuickTime er, i likhet med DirectShow, et system utviklet av et stort firma og har vært i bruk i lang tid. Apple plattformen er veldig populær til videoredigering, og mange av applikasjonene til videoredigering bygger på QuickTime rammeverket.

Funksjonalitet. QuickTime er et rammeverk for utvikling av multimediaapplikasjoner. Det er et stort rammeverk, med veldig mange innebygde funksjoner. Web-sidene til Apple lister opp mange "whipeout" effekter, keying effekt-

er, egendefinerte konvolusjons matriser, alphablending osv. Det er også støtte for streaming av video. QuickTime kan bygges ut videre med tredjeparts komponenter som kan øke funksjonaliteten.

Kostnad. Utviklingsgrensesnittet er tilgjengelig fra Apples hjemmeside for Macintosh og Windows plattformen. I tillegg trengs kompilatorer som fungerer mot dette grensesnittet, og et operativsystem. Videre kommer kostnadene med å utvikle en applikasjon for sluttbrukeren. Som med DirectShow er det utviklingskostnadene som er den store kostnaden.

Brukergrensesnitt. Brukergrensesnitt må lages selv. Siden QuickTime kjører både under Windows og Macintosh er det valgfritt hvilken vindus API applikasjonen utvikles for. Men de er begge ganske like hverandre og har stor funksjonalitet.

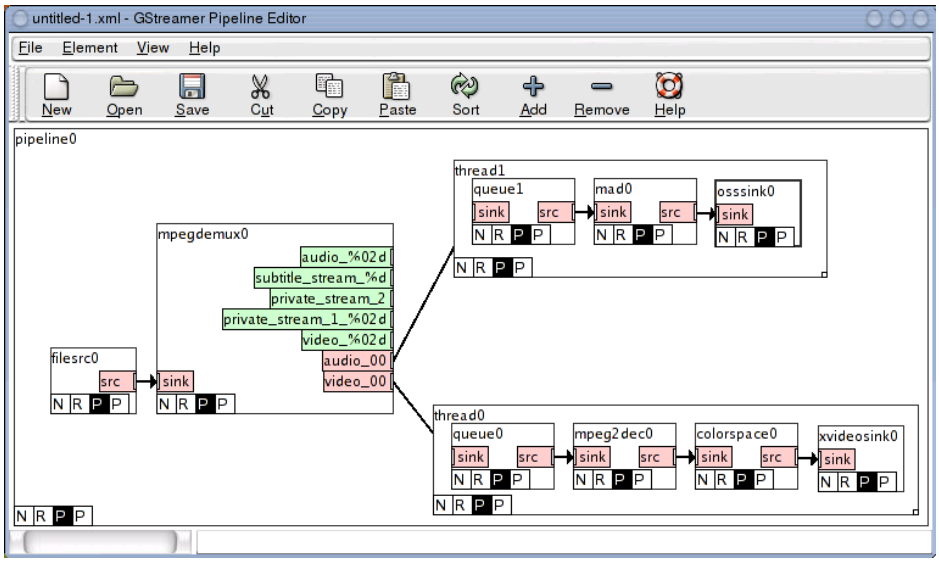
Plattform. QuickTime er utviklet av Apple, som også har utviklet Macintosh plattformen. Den er innebygget i de seneste versjonene av Mac OS som det interne multimediasystemet der, mye på samme måte som DirectShow systemet er i Windows. QuickTime systemet finnes også under windows, og brukes der som DLL[3]er. Maskinvarekravet er enten en PC med Windows, eller en Apple Macintosh med Mac OS.

2.2.7 Gstreamer

Gstreamer[30] er en multimediaplattform med åpen kildekode for Gnome, et skrivebordsmiljø til Linux. Ideen bak Gstreamer er å lage et rammeverk som skal gjøre det enkelt for utviklere å lage multimediaapplikasjoner, altså samme ide som DirectShow og QuickTime. Forskjellen er at Gstreamer er et helt åpent og fritt system. Gstreamer benytter grafer, i likhet med PD. Utvikleren setter opp en graf med nødvendige komponenter for å utvikle den ønskede applikasjonen. Grafene kan også genereres automatisk, basert på input og output.

Systemet er bruker plugins, også kalt komponenter, til å løse konkrete oppgaver. Det finnes kodeker til de fleste kjente formater (MPEG, MP3, OGG, AVI osv.), filtere for noen effekter og output plugins til de viktigste output systemene i Linux. Pluginsystemet er designet slik at enhver plugin registrerer seg i systemet med informasjon inputformat og outputformat. Systemet kan selv bygge en graf med nødvendige komponenter for å få data fra ett ønsket inputformat til et ønsket output format. Skal det f.eks. spilles en .mp3 fil på et standard Linux-system

[3]Dynamisk innlastede biblioteker



Figur 11: gst-editor er grafeditoren til Gstreamer. Her er en eksempelgraf som dekoder en MPEG2 strøm og sender output til skjerm og høytalere.
Bildet er hentet fra: `http://www.gstreamer.net/`.

med OSS lyd-drivere trengs en mp3-parser, en mp3-dekoder og en oss-sink. En plugin har enten en "sink", en "src" eller begge deler tilknyttet seg. En "sink" er et koblingspunkt hvor pluginen leser inn data fra en annen plugin, en "src" er et koblingspunkt hvor pluginen sender ut data til andre plugins.

Kildekoden er fritt tilgjengelig under GNU LGPL lisensen. Et firma som heter RidgeRun sponser en del av utviklingen for å bruke rammeverket i utviklingen av multimediasystemer på deres embeded-linux versjon.

Gstreamer er fremdeles noe uferdig. Det er mulig å spille av lyd, og en del videofiler, men det finnes ikke noen særlig støtte for redigering av video. Det er planlagt en del støtte ved å bruke deler fra f.eks. The Gimp til å redigere video.

Funksjonalitet. Funksjonaliteten i rammeverket muliggjør automatisk generering av grafer for å spille en rekke forskjellige formater. I følge hjemmesiden til Gstreamer, fungerer Gstreamer i dag bra til lydapplikasjoner, men mangler fremdeles noe funksjonalitet på video siden.

På mange måter kan den planlagte funksjonaliteten til gstreamer minne om PD og Max. Grafer av objekter benyttes til å kontrollere strømmen. Gstreamer vil også ha innebygget en grafeditor som lar brukeren kontrollere hva som brukes av

komponenter.

Kostnad. Det er ingen kostnader tilknyttet anskaffelse av programvare for å bruke dette systemet. Kostnadene ligger i innkjøp av maskinvare og utvikling av applikasjon til sluttbrukeren.

Kildekoden er fritt tilgjengelig og kan endres.

Brukergrensesnitt. Gstreamer i seg selv har ikke noe brukergrensesnitt. Det er et rammeverk ment for å lage multimediaapplikasjoner i Gnome. Brukergrensesnittet til det som finnes av applikasjoner basert på gstreamer er basert på Gnome og XML. Gstreamer vil inneholde komponenter som muliggjør redigering av XML-grafer for oppsett av video- og lydstrømmer. Dette programmet heter gst-editor, se figur 11 på forrige side for en eksempelgraf. Utover dette kan det lages brukergrensesnittet til sluttbrukerapplikasjonen i KDE eller Gnome.

Plattform. Gstreamer er per i dag bundet til UNIX plattformen, dvs. Linux, BSD, Solaris eller andre som støtter Gnome. Støtte for Windows er også påbegynt, men det er ikke prioritert. Mesteparten av utviklingen gjøres på Linux med ix86 maskiner, men Gstreamer er også designet for å kjøres på mindre kraftige maskiner.

2.3 Praktisk testing av programvare

For å kunne velge programvare sluttbrukeren kan bruke til videoredigering er et viktig kriterium hvordan programvaren virker i praksis. Den eneste måten å finne ut dette på er å teste programvaren i praksis.

Blant programvaren som er listet over måtte det velges hva som skulle testes. Å utvikle en videoredigeringsapplikasjon fra bunnen er for tidkrevende for en hovedfagsoppgave, derfor ble det valgt å bruke en ferdig applikasjon i stedet for å bruke DirectShow, QuickTime eller Gstreamer til å bygge en egen applikasjon.

Prisen på innkjøp av MAX/NATO, kombinert med at den ikke har åpen kildekode, ble utslagsgivende for å ikke prøve denne pakken. Det ville være enklere å knytte nettverkstøtte inn i en applikasjon hvor kildekoden er tilgjengelig enn i et lukket program. Valget falt på å teste PD/GEM og FreeJ siden EffecTV manglet fleksibilitet og grafisk brukergrensesnitt.

2.3 Praktisk testing av programvare

2.3.1 Krav til maskinvare

Følgende krav til maskinvare er funnet ved å lese tilgjengelig dokumentasjon og å teste programmene. Kravspesifikasjonen her er satt opp etter hva som er nødvendig for å muliggjøre enkel redigering av video i sanntid med disse applikasjonene. Enkelte spesielt avanserte effekter eller funksjoner vil kunne medføre høyere krav til maskinvare.

FreeJ krever et video4linux device for å starte. I Linux er video4linux standard driver-API for lesing fra videokort og kamera. Både FreeJ og GEM støtter video4linux, men kun FreeJ krever det. GEM trenger det kun ved redigering av live video fra tilkoblet kamera.

Video4linux drivere ligger i linux-kjernen, hvilke kort som er støttet er begrenset av tilgjengelige drivere der. Kravet er at kortet må være støttet, i testene her er det valgt et standard BT848 kompatibelt kort da dette er et veldig utbredt kort med gode drivere.

GEM bruker OpenGL til å vise video, så for å få bra ytelse i GEM trengs et skjermkort med bra OpenGL drivere. NVidia skjermkort er kjent for å ha veldig bra OpenGL drivere i Linux og Windows, særlig til spill bruk, men også til profesjonell bruk.

PD krever også et lydkort. Her er det viktigste at det har bra støtte i Linux. Et standard Creative SoundBlaster Live er tilstrekkelig, og koster under 300 kroner.

Resten av maskinen må velges ut fra sluttbrukerens krav til funksjonalitet. Minimumskravet er en Intel-kompatibel CPU med 1 GHz eller mer for å kunne bearbeide bildedata for nok. AMD Athlon/XP CPUer er kjent for å fungere bra i Linux, de gir også mye ytelse for pengene. Detbør være minimum 256 MB RAM i maskinen for å hindre at minnet skal gå fult. Minnet bør være av typen DDR-SDRAM eller RAMBUS, da video krever høy båndbredde.

Tabell 2.3.1 viser komponentene i maskinen som ble brukt i testene her. Tilsvarende komponenter fra andre leverandører kunne også vært brukt.

Prosessor:	AMD 1.4 GHz eller bedre
Minne:	256 MB DDR-SDRAM eller mer
Lydkort:	Creative SoundBlaster Live
Grafikk:	NVIDIA GeForce2 MX 400
Operativsystem:	Linux (Debian) med 2.4 kjerne
Videokort:	BT848 basert

Tabell 1: Spesifikasjon av maskin til test.

2.3.2 Sammenligning av PD/GEM og FreeJ

FreeJ og PD/GEM ble testet opp mot en kravspesifikasjon. Resultatet er systematisert i tabell 2.

Ofte har ikke programmene direkte støtte for funksjonaliteten i kravspesifikasjonen, men i PD/GEM kan brukeren kombinere komponenter til å tilnærme den. I tabellen er det foreslått en måte å gjøre dette.

FreeJ. Under testen av FreeJ viste det seg at FreeJ ikke inneholder funksjonaliteten denne oppgaven krever. Det følger med få filtre, og de er meget begrenset i funksjonalitet slik at det mangler bredde i hva filtrene kan gjøre. Fleksibiliteten er også mangelfull. Måten FreeJ er bygd opp gjør at det er enkelt å bruke, så om det kommer flere filter kan dette være et interessant program.

Den største styrken til FreeJ er måten det styres fra tastaturet ved å sette opp filterkjeder. Brukeren kan ved ulike tastetrykk styre hvilke filtre som skal brukes i hvilket lag, og endre parametre til disse.

PD/GEM. PD/GEM virket først noe ustabilt, men dette viste seg å være et problem med skjermkortdriveren som nå er fikset.

Funksjonaliteten som er angitt i tabell 2 er her en funksjonalitet som er mulig ved å kombinere funksjonalitet fra flere objekter. Det er muligheten til å kombinere objektene fleksibelt som er PD/GEMs største styrke. Kombinasjonene i tabellen er ikke alltid utprøvd, da det vil ta for mye tid å teste alle kombinasjonene.

Tabell 2: Tabell over funksjonalitet i PD/GEM og FreeJ

Funksjonalitet	**GEM**	**FreeJ**
Frame buffer med key inngang	Ingen støtte.	Ingen støtte.
Colorization	pix_color kan brukes til å forandre fargen på video.	Blitt-effekter og colorcycling kan brukes til å forandre farge. Noen av filtrene har også støtte for dette.
fortsetter på neste side		

fortsatt fra forrige side

Funksjonalitet	**GEM**	**FreeJ**
Keyere: Luminans, chroma, ekstern matte, difference	Muligheten for dette er tilstede. Ekstern matte kan oppnås med pix_mask og difference med pix_substract. Den eksterne matten kan enten være et statisk bilde, eller en videokilde. Chroma kan gjøres med pix_chroma_key, luminanskeying med pix_compare.	Ingen støtte.
Displacement mapping	Siden alt i GEM er OpenGL objekter kan displacement mapping oppnås ved at videoen legges oppå et forvridd objekt.	Ingen støtte.
DVE Resize	Den kan gjøres ved å skalere objektet videoen er tekstur til. Selve teksturen kan også skaleres.	Zoom-filter kan brukes til skalering.
DVE Rotation	Dette kan gjøres ved å rotere objektet videoen er tekstur til.	Rotazoom-filteret roterer og zoomer videoen, men kontrollen er veldig begrenset.
Warp	Ved å forvri 3D-objektet video er tekstur på vil resultatet bli forvridd video. Avansert warping kan være noe vanskelig å gjøre i praksis, men det er mulig.	Ingen støtte.
Noise generation	Dette kan gjøres ved å generere tilfeldige tall, eller ved å bruker partikkelgenerator for å få systematisk “støy”.	Kan gjøres ved hjelp av blitting.
Emboss	Dette kan gjøres ved å bruke en konvolusjonsmatrise i pix_convolve.	Ingen støtte.
Sequencer	Ingen støtte.	Ingen støtte.

fortsetter på neste side

fortsatt fra forrige side

Funksjonalitet	**GEM**	**FreeJ**
Mix/Wipe	Dette kan gjøres ved å styre alfakanalen til en videokilde. Wipe er vanskeligere å få til, men det kan gjøres ved å lage en animasjon som brukes som alfakanal.	Ingen støtte.
Outline	Kan gjøres ved å lage et større objekt som ligger bak video-objektet.	Kan ha et objekt bak video-laget.
Equalize	Det nærmeste er pix_normalize.	Ingen støtte.
Delay	Kan gjøres med pix_delay.	Ingen støtte.
Feedback buffer	pix_snap resulterer i et objekt med ferdigrenderet data som kan brukes senere.	Ingen støtte.
Representasjon av lyd	I nyeste versjon av GEM er dette mulig. Men denne funksjonaliteten er noe begrenset.	Ingen støtte.
Autopaint	Det finnes mange muligheter her. For eksempel kan baner for objekter eller partikler settes opp. Ved kombinasjon av objekter kan mye oppnås.	Ingen støtte.

2.4 Valg av programvare til denne oppgaven

Valget falt på PD/GEM. PD/GEM hadde mer funksjonalitet enn FreeJ, og den funksjonaliteten sluttbrukeren trenger er lettere tilgjengelig i PD/GEM enn i FreeJ.

PD/GEM fungerer som et grafisk programmeringsspråk. Det er derfor fleksibelt, og muliggjør avansert funksjonalitet ved å kombinere forskjellige komponenter. Det meste vil også være mulig i FreeJ dersom en er villig til å programmere inn funksjonaliteten selv, men brukeren må da kompilere om programmet hver gang ny funksjonalitet legges til. I PD/GEM er det nok å endre grafen i et grafisk brukergrensesnitt, noe som kan gjøres mens systemet er i bruk.

Et annet argument for å velge PD/GEM var at brukermiljøet på Internett rundt PD/GEM er mye større enn til FreeJ. Dette gjør det lettere å få hjelp under utviklingen og bruken av systemet. PD/GEM har egne diskusjonslister som er tilgjengelige for alle, og det er stor aktivitet der. FreeJ blir utviklet av to personer, og det er vanskelig å finne noe brukermiljø.

3 Overføring av video over nettverk

For å få overført video fra en maskin til en annen trengs et fysisk nett, og en form for protokollstruktur. Det fysiske nettet må ha egenskaper som gjør det egnet for overføring av data med stor båndbredde og krav til lav forsinkelse, protokollstrukturen bør være mest mulig fleksibel, samtidig som funksjonalitetskravene til systemet dekkes.

Siden rå PAL-kvalitet video med 32 bit per piksel tar ca. 44MB/s (768x576 piksler (kvadratiske piksler), 25 bilder i sekundet, se 1.2 på side 5) må videostrømmen komprimeres før det sendes ut på nettet for at båndbreddekravet skal oppfylles.

3.1 Krav til nettet

Båndbredde. Nettverket mellom maskinene begrenser hvor mye data som kan sendes mellom dem i et gitt tidsintervall. Det vanlige er 10Mbit/s eller 100Mbit/s nettverk. 1Gbit/s nettverk begynner også å bli mer vanlig, men for “Hvermansen” er 100Mbit/s 802.3 LAN det mest aktuelle.

Dersom det benyttes et 100Mbit/s nett vil det kunne overføres rundt 13MB/s dersom båndbredden i nettet utnyttes hele tiden. Det vil si at 4:1 komprimering er tilstrekkelig til å overføre video i sanntid. En halvering av oppløsningen vil gi en 4:1 kompresjon. Videostrømmen vil da få en oppløsning på 384x288, og båndbreddekravet blir redusert til 11MB/s. Med 84,4% utnyttelse av båndbredden i nettet vil dette være mulig.

Dersom andre maskiner og applikasjoner også vil benytte nettet vil tilgjengelig båndbredde bli mindre. Dette vil gjøre det nødvendig med enda høyere kompresjon. Det anbefales å sette opp et dedikert nettverk for at videooverføringen ikke skal forstyrres av andre applikasjoner. Men dersom båndbredden utnyttes maksimalt for å overføre én strøm vil det ikke være mulig å utvide systemet til å sende strømmen videre til en annen maskin, det vil heller ikke være mulig å redigere mer enn en strøm av gangen. For å beholde fleksibilitet kreves det at hver enkeltstrøm ikke bruker mer enn 10Mbit/s, det er da mulig med maksimalt ti samtidige strømmer i nettet. Derfor er det nødvendig å komprimere ytterligere.

Dersom det benyttes et 10Mb/s må kompresjonen være minst 40:1. Med en oppløsning på 96x72, vil en 64:1 kompresjon oppnås. Skarpheten i bildet vil nå være borte slik at detaljer ikke lenger er synlige. Båndbreddekravet er nå 691KB/s, noe som er overkommelig dersom det ikke er mye annen trafikk i nettet. Se tabell 3 på neste side for en oppsummering av hvilke skaleringer som må gjøres for å kunne overføre video med forskjellige nettverkshastigheter.

Nettverk	Krevd komp.	Oppløsning	Komp.	Båndbreddebruk
1000 Mbit/s	1:1	768x576	1:1	354 Mbit/s
100 Mbit/s	4:1	384x288	4:1	88 Mbit/s
10 Mbit/s	40:1	96x72	64:1	6 Mbit/s

Tabell 3: Komprimering ved skalering

For å beholde mest mulig informasjonen i bildet bør forskjellige komprimeringsalgoritmer som beholder mesteparten av kvaliteten samtidig som det oppnås et lavt båndbreddekrav, vurderes. Se 3.3.1 på side 39 for en gjennomgang av slike komprimeringsalgoritmer.

Pakketap. Det som er av pakketap i lokalnett er kollisjoner i nettet, noe som ordnes av det fysiske laget ved at nettverkskortet venter til nettet er ledig igjen og sender på nytt. Systemet må likevel ta høyde for at pakker kan bli borte. Når pakker blir borte må det bestemmes om pakkene skal sendes på nytt, eller om de bare skal droppes. Dersom en pakke blir borte flere ganger må det finnes en strategi for hvor lenge systemet skal vente på en bestemt pakke.

Hvilken strategi som velges avhenger av hva slags trafikk som sendes. Ved sending av video er det vanligvis viktigere at informasjonen som mottas er ny enn at all informasjon kommer frem.

I et internett er det flere grunner til pakketap. Vanligste årsak er at pakker blir kastet av rutere som resultat av metning i nettet. Dette rettes ikke opp av nettverket, og må derfor rettes opp av transmisjonslaget i klientene. Dersom data i en sanntidsstrøm sendes på nytt i en slik situasjon, vil det føre til økt metning og vil forverre situasjonen. Alternativt kan data sendes saktere til metningen er over slik TCP gjør, dette vil føre til økt forsinkelse eller tap av kvalitet.

Forsinkelse og forsinkelsevarians. Den begrensningen som går mest ut over ytelsen på en multimediaapplikasjon er forsinkelse[31]. Forsinkelsen er den tidendet tar fra en pakke sendes fra en applikasjon til mottakerapplikasjonen mottar den. Det er flere årsaker til forsinkelse, en årsak er hastigheten pakkene kan sendes med over det fysiske mediet, andre årsaker er at pakkene ofte må gjennom flere lag av operativsystemet. Pakkene blir generert i applikasjonen, men når de skal sendes ut på nettet går de først ned til transportlaget i kjernen, så til nettverkslaget og til slutt til driveren til nettverkskortet. Låsing av delte ressurser gjør at det kan bli forsinkelse her. I [32] er forsinkelsen i et videokonferansesystem målt. Tallene viser at nettverket er den minst signifikante kilden til forsinkelse.

Forsinkelsevarians (Eng. Jitter) er et mål for hvor mye forsinkelsen varierer. Variasjonen oppstår når pakker må gjennom rutere på veien fra senderen til mottakeren og ruterne er i ferd med å bli mettet[31]. Pakken blir da liggende i en buffer til den kan sendes gjennom ruteren. Tiden pakken ligger i bufferen varierer slik at forsinkelsen på pakken også varierer. Dette gjør at det er vanskelig for mottakeren å vite hvor gamle pakkene er.

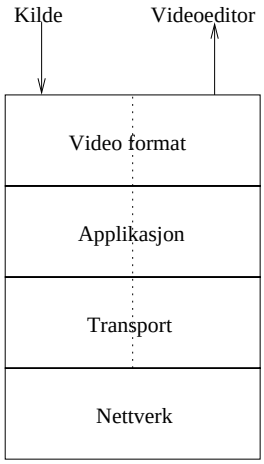
I denne oppgaven vil det benyttes et lokalnet uten rutere. I et slikt nett kan forsinkelsesvarians produseres av det som på engelsk kalles "capture effect" [33]. Dette er et fenomen som oppstår når en maskin tilsynelatende "holder" sendemediet i en periode tilsvarende flere pakkesendinger og lager forsinkelse for andre sendere. Årsaken til denne effekten er "Exponential Backoff". Når en kollisjon forekommer vil første sender som klarer å ta mediet få høyere prioritet siden den vil vente kortere enn de andre senderene ved neste kollisjon. Når nettverket er lite belastet vil dette problemet være mindre. Når kun en sender er aktiv vil båndbreddeproblemet være mye større.

Løsningen på problemet med forsinkelsesvarians er vanligvis å ha en buffer hos mottakeren. Bufferen brukes til å gi videodekoderen en jevn strøm med data selv om strømmen inn er ujevn. Problemet her er at bufferen innfører forsinkelse tilsvarende maksimal forsinkelsesvarians. Dette er lite ønskelig i en situasjon hvor resultatet skal vises fram i sanntid.

Oppsett og valg av fysisk nett. Applikasjonen i denne oppgaven vil bli kjørt over et 100Mbit/s 802.3 dedikert ethernett, som ikke kjører andre applikasjoner samtidig. Dette gjør at problemet med forsinkelse blir minimalt. Pakketapet i nettet bør heller ikke forekomme, da eneste kilde til trafikk er applikasjonen som utvikles her. Båndbreddekravet på 100Mbit/s er valgt både med tanke på kostnad for sluttbrukeren, og oppnåelig kvalitet med komprimering. Tregere nett vil føre til redusert kvalitet, raskere nett vil medføre høyere kostnader.

3.2 Lagdeling

Det er vanlig å dele opp nettverksystemer i lag. *Open Systems Interconnection* (OSI) var en av de første organisasjonene som formelt definerte en måte å koble sammen datamaskiner. Deres modell, OSI-modellen[34], brukes som en referanse modell for andre protokollgrafer. OSI-modellen bruker 7 lag, den mer vanlige Internett-modellen bruker kun 4. En lagdeling er fornuftig av mange grunner, særlig at den gjør systemet enklere å forstå og også det at det lettere lar seg implementere.



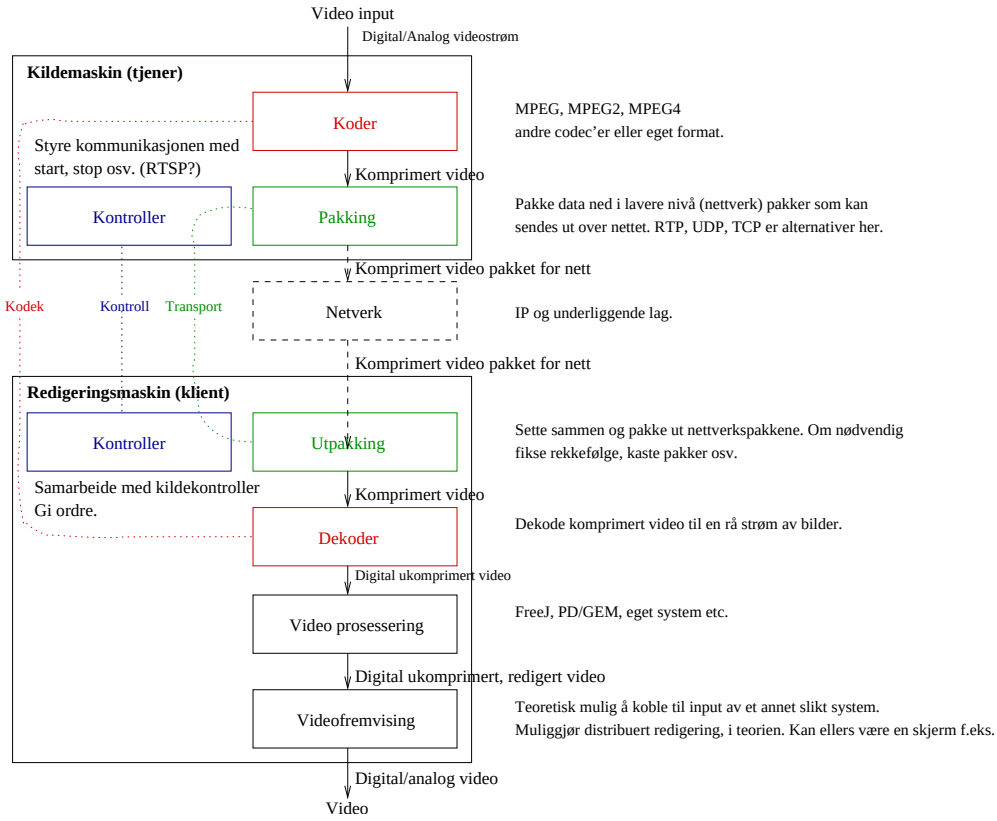
Figur 12: Lagdeling av et overføringssystem for video.

I en slik lagdeling er målet for hvert lag å abstrahere vekk lagene under, slik at lagene kan behandles ett og ett med antagelsen om at de underliggende lagene allerede fungerer. Det antas her at leseren er kjent med oppbygningen av Internett- og OSI-modellen, for å raskt oppsummere er Internett-modellen bygd opp av et fysisk lag (eks. IEEE 802.3) i bunnen, et nettverkslag over dette (eks. IPv4), et transportlag (eks. TCP) og et applikasjonslag på toppen (eks. HTTP).

I OSI-modellen vil lag 6, presentasjonslaget[34], omhandle HTML, XML og de andre beskrivelsesspråkene. Dette laget har ikke noe med selve overføringen å gjøre, og er heller ikke et eget lag i IP-modellen. Men det er likevel det viktig å vite at det er der, siden dette nivået for eksempel muliggjør at nettsider ser like ut på forskjellige maskiner med forskjellige nettlesere.

Også et videooverføringssystem bør ha en slik lagdeling. Et nettverkslag i bunnen, IP er ønskelig for å muliggjøre overføring gjennom eksisterende nettverk. Et transport lag som UDP eller TCP som tar seg av overføringen av datastrømmen. Et lag som forteller hvordan overføringen kontrolleres (hente et bilde, motta et bilde osv.) og til slutt et lag på toppen som sier hvordan videostrømmen er kodet. Se figur 12 for en tegning av hvordan dette kan se ut. Husk at under nettverkslaget er det også et fysisk lag, men dette er ikke interessant i denne sammenhengen.

Den stiplede linjen i midten av figur 12 skiller mellom sender og mottaker. Strømmen går nedover lagene i kildemaskinen, ut på nettverket før den går oppover lagene i mottakermaskinen. Det nederste laget tar seg av overføringen, i denne oppgaven benyttes IP.



Figur 13: Lagene i videooverføringssystemet brettet ut slik at hva som gjøres i hvert ledd kan sees. Flyten starter fra toppen og går gjennom hvert lag før den kommer ut ferdig behandlet i bunnen.

3.3 Moduler i et lagdelt system

Figur 13 viser hvordan figur 12 vil se ut dersom den brettes ut slik at de ulike modulene som er involvert kommer til syne. Med moduler menes her logisk adskilte enheter av koden som tar seg av fast definerte oppgaver. Tilsammen løser modulene en større oppgave enn det modulene hver for seg klarer. En slik oppdeling gjør det lettere å se detaljene i hva som kreves av systemet.

Den første modulen er en modul som tar video inn, legger ett og ett bilde i en framebuffer og sender disse ned til en koder.

Koderen komprimerer video til et bestemt format som dekoderen vet om. Til sammen utgjør disse det en kodek. Det finnes mange videoformat som dekker behovet her, de mest utbredte er MPEG-1 og MPEG-2, hvor MPEG-1 brukes i systemer som VideoCD, mens MPEG-2 brukes til digitale TV-sendinger, DVD og

SVCD. MPEG-4 er en standard for overføring av video på internett og mobile nett, kodeken derfra er i utstrakt bruk. Andre formater som H.261, H.263 og MJPEG er også mulige å bruke her, mer om kodeker i 3.3.1 på neste side. Det som kommer ut av denne modulen er en videostrøm som ikke overstiger et bestemt båndbreddekrav. Koderen må derfor konfigureres til å respektere dette kravet. Med videostrømmen må det ligge informasjon som gjør det mulig for dekoderen å dekode strømmen. Det viktigste her er informasjon om hvilken kodek som ble brukt.

I kontrollermodulen styres datastrømmen. Her trengs et sett med kommandoer applikasjonene i hver ende kan bruke til å styre flyten av data. “Start”, “Stop”, “Pause” osv. er kommandoer som kan implementeres her. Det vil normalt sett være videooutput delen som styrer dataflyten i et videoredigeringssystem, siden det er her brukeren er. Kontrolleren forteller videokilden når sendingen skal begynne, og når den skal slutte. Kontrolleren i denne enden (kilden) blir da en slags tjener som venter på en kommando som sier at den skal begynne å sende data ut på nettet.

Strømmen fra koderen går så videre til transportlaget. Her deles strømmen opp og pakkes inn i nettverkspakker som sendes ned til nettverksnivået og ut over nettet. I transportlaget er det en transportprotokoll. Her finnes det flere mulige protokoller. De mest brukte i dag er TCP og UDP. Retransmisjon fører til forsinkelse og gjør TCP mindre egnet til overføring av sanntidsvideo enn UDP siden pakker ikke sendes på nytt med UDP. En protokoll kalt RTP (Realtime Transport Protocol) er laget for å overføre sanntidsinformasjon som video og lyd. RTP bruker TCP eller UDP i bunnen for å overføre data, men tilfører ekstra informasjon som tidsstempling, sekvensiering, mulighet for metningskontroll med RTCP og kodekspesifik informasjon. Mer om dette i 3.3.3.

Fra Transportlaget går nettverkspakkene gjennom nettet til maskinen i den andre enden. Det antas her at Nettverket er basert på internett protokollen (IPv4). Grunnen til dette er at Internett er det mest utbredte nettet i dag, og derfor er det nettet “Hvermansen” kan ventes å ha tilgjengelig.

I andre enden sørger transportlaget for å ta imot pakkene, sette dem i riktig rekkefølge og pakke dem ut til en datastrøm som sendes videre opp i systemet. Transportlaget må i denne enden ta en del flere beslutninger. Disse avhenger av hvilken protokoll som er valgt. Det kan velges å vente på pakker som ikke er mottatt, eller å glemme dem og på den måten ofre bildekvalitet til fordel for bedre sanntidsflyt. I en sanntidssituasjon er det ofte bedre å få data med noe feil enn gammel data. Særlig hvis tap skjer sjelden eller avstanden mellom (forsinkelse) maskinene er stor. Det må beregnes en forsinkelse på tre enveis turer for å få retransmitert en pakke (to for å sende pakken og en for å gi senderen beskjed om

at første ikke kom frem) [35]. I denne oppgaven er avstanden liten, retransmisjon vil derfor ikke føre til så stor ekstra forsinkelse.

Kontrolleren i mottakeren styrer kontrolleren i senderen og sier fra om den vil ta imot video eller ikke. Den bør også lytte etter kommandoer fra den andre kontrolleren. Det er mulig den andre kontrolleren vil avslutte strømmen tidligere, for eksempel hvis kameraet slås av.

Videostrømmen sendes til dekoderen, som dekoder video ifølge kodeken og videoformatet som ble brukt i koderen. Resultatet blir en strøm av bilder som sendes til videoredigeringsprogramvaren.

Tre valg må gjøres i dette kapitlet:

- Hvilket format skal benyttes på videostrømmen?
  Dette valget vil styre kvaliteten på videostrømmen, samt båndbreddekravet til nettverket. Det vil også påvirke forsinkelsen og kravet til datakraft i tjeneren.
- Hvordan skal protokollen i applikasjonslaget se ut?
  Dette valget vil påvirke fleksibiliteten til styringen av strømmen.
- Hvilken protokoll skal benyttes i transportlaget?
  Dette valget vil påvirke forsinkelse, og også hvordan applikasjonen vil tåle pakketap.

3.3.1 Kodeklaget

Som nevnt i 3.1 på side 33 må videostrømmen komprimeres for å redusere båndbreddeforbruket. De mest kjente, og lettest tilgjengelige kodekene for videokomprimering blir vurdert her.

Det største problemet med å komprimere video i en sanntidssituasjon, er at det innfører enda ny kilde til forsinkelse, siden mye data skal behandles og en del av teknikkene som brukes er veldig tidkrevende. I verste fall kan komprimeringen ta så lang tid at det ikke kan gjøres i sanntid. Tradisjonelt har komprimering basert seg på at en kraftig maskin benyttes til å komprimere mens en mindre kraftig benyttes til å dekomprimere. I profesjonelle miljø benyttes gjerne spesialmaskinvare til komprimeringen, et MPEG-kodekort er et rimelig alternativ for “Hvermansen”.

“Billig komprimering”. En enkel metode for komprimering som allerede er nevnt, er å redusere oppløsningen. Dette kan gjøres over flere dimensjoner. Den mest opplagte er å gjøre det i bredde og høyde. Dette vil føre til at båndbreddekravet reduseres i takt med arealet av bildet. Halveres høyden og bredden vil båndbreddekravet bli en fjerdedel av det kravet var fra før. Oppløsningen på fargene kan også reduseres. Vanligvis brukes 24bit fargedybde, som gir 8 bit per piksel. Ved å gå ned til 16bit eller 8bit fargedybde vil det oppnås henholdsvis 3:2 og 3:1 kompresjon, men dette innebærer at det må beregnes et optimalt fargekart som legges ved bildet for at utvalget av tilgjengelige farger ikke skal bli for begrenset. En fordel med 8bit fargedybde er det blir enklere å bruke tradisjonelle tapsfrie komprimeringsteknikker som huffmann og runlength koding da flere punkter i bildet nå blir like. Den siste dimensjonen oppløsningen kan reduseres over er tid. Vanlig PAL video har 25 bilder i sekundet. Ved å redusere dette til 5 bilder vil båndbreddekravet reduseres til en femtedel. Dessverre blir opplevelsen for brukeren bli hakkete og lite flytende. Å redusere bildefrekvensen til en frekvens som ikke kan dele den opprinnelige kan føre til ujevne bevegelser siden det da ikke lenger er konstant tid mellom hvert bilde.

Ved å utnytte svakheter i synsoppfattelsen til mennesket kan kompresjon som ikke går så mye ut over den objektive kvaliteten til bildet oppnås. En av de mest brukte svakhetene er at menneskets øye er mer sensitiv ovenfor lysstyrke enn farger[36]. Dette utnyttes i en prosess som kalles “subsampling”. Her gjøres først fargene i bildet om fra RGB (Rødt, Grønt og Blått) til YUV (eller tilsvarende hvor Y er lysstyrke og resten er fargeinformasjon). Etter at bildet er gjort om til YUV reduseres oppløsningen til U og V komponentene. Bildet ser fremdeles skarpt ut, men den får noe dårligere farger.

En siste kompresjonsteknikk er å bruke en generell komprimeringsalgoritme som huffman eller runlength. Problemet med disse er at de de baserer seg på informasjon som gjentas over en dimensjon, noe som ofte forekommer i tekst. I video er det vanligvis litt støy og det er hyppige variasjoner når signalet behandles over kun en dimensjon. Derfor brukes denne teknikken stort sett sammen med andre teknikker som redusert fargedybde (som i GIF) eller som en del av de mer avanserte komprimeringsalgoritmene.

HUFF-YUV. HUFF-YUV[37] er en tapsfri komprimeringsteknikk som ligner på tapsfri JPEG. Den predikerer hver pikselverdi og Huffman-komprimerer differansen mellom predikert verdi og virkelig verdi. Det finnes forskjellige funksjoner som kan brukes for å predikere verdiene. “Left” predikerer at verdien er lik forrige verdi i samme kanal (En kanal for Y, U og V eller R, G og B). “Gradient” predikerer at verdien er verdien til venstre + verdien over -

verdien skrått opp til venstre. “Median” predikerer at verdien blir medianen av venstre verdi, verdien over og en verdien “gradient” funksjonen gir. Kanalene komprimeres hver for seg.

Dette er en enkel komprimeringsalgoritme. Den er derfor rask, sammenlignet med de mer kompliserte algoritmene, og har også den fordelen at kvalitet ikke går tapt. Ulempen er at det ikke oppnås veldig høy kompresjon. Ifølge web-siden[37] til en Windows-implementasjon av denne algoritmen oppnåes det rundt 2.5:1 komprimering avhengig av hvilken funksjon en bruker til å forutse verdiene. En av grunnene til at det ikke oppnås høyere kompresjon er at algoritmen kun komprimerer i romlige dimensjoner, ikke over tid.

MJPEG. MJPEG (Motion JPEG) er en veldig løst definert standard hvor hovedpoenget er at hvert bilde JPEG komprimeres for seg selv og legges etter hverandre.

JPEG er en vanlig komprimeringsalgoritme for bilder definert under standard “ISO/IEC IS 10918-1”. JPEG fungerer ved at bildet deles opp i små regioner som kjøres gjennom en DCT (Diskret Cosinus Transformasjon). Etter denne transformasjonen fjernes noe av informasjonen (den som har minst betydning for kvaliteten på bildet) og resultatet komprimeres med run-length og huffman. I tillegg brukes subsampling teknikken, forklart over, til å fjerne fargeinformasjon øyet ikke kan se[38].

MJPEG er ikke tapsfri, men den oppnår en 10:1 kompresjon eller høyere. Det som tar mest tid her er DCT transformasjonen ved kodingen av bildet, men dette er en transformasjon de aller fleste videokomprimeringsalgoritmer benytter seg av. MJPEG utnytter ikke likheter i etterfølgende bilder. Dette gjør at det kan bli mye redundant informasjon lagres, men det har den fordelen at bildene ikke er avhengige av hverandre. Dette, samt høy kvalitet, er grunnen til at MJPEG ofte brukes til videoredigering.

En definert standard for MJPEG er definert i RFC 2435[39]. Her er det definert hvordan bildene skal lagres i RTP-pakker, hvordan de skal deles opp og settes sammen og så videre. Dette gjør det lett å koble sammen utstyr som jobber med JPEG over RTP, selv om dette ofte er vanskelig med annet utstyr som bruker MJPEG.

MPEG. MPEG er en forkortelse for Moving Picture Expert Group, som er navnet på gruppen som definerer MPEG standardene. Det finnes i dag flere MPEG standarder, de er MPEG-1 (ISO/IEC 11172), MPEG-2 (ISO/IEC 13818), MPEG-4 (ISO/IEC 14496), MPEG-7 (ISO/IEC 15938) og MPEG-21 (ISO/IEC 18034)[40].

MPEG-1 er en standard for å kode et kombinert audio- og videosignal med en bitrate opp til 1,5 Mbit/s, noe som gir en kvalitet sammenlignbar med VHS (standard videokassetter). Standarden består av 5 deler. Del en beskriver en syntaks for hvordan pakker med lyd og bilde kan overføres og lagres, og hvordan lyd og bilde kan synkroniseres. Del to beskriver hvordan video kan komprimeres, del tre hvordan lyd kan komprimeres. Del fire beskriver hvordan om konformitet testes, del fem viser eksempler på software enkoder og dekoder.

MPEG-1 videokomprimering baserer seg på DCT komprimering som i MJPEG og JPEG. MPEG-1 ser i tillegg på likheter i etterfølgende bilder[41]. To etterfølgende bilder i en video sekvens vil inneholde nesten identisk informasjon dersom det ikke er mye bevegelse. Selv om det er bevegelse er det mye likheter i bildene dersom objektene i bevegelse ikke endrer seg, men bare forandrer posisjon. MPEG-1 utnytter at disse likhetene er redundant informasjon.

En MPEG-1 videokoder tar en sekvens med bilder som input og komprimerer disse til tre typer rammer (Engelsk: frames), kalt I rammer (Intra-frame), P rammer (Predicted-frame) og B rammer (Bidirectionally predicted frame).

I-rammer kan tenkes på som en referanseramme. Her ligger hele bildet uten referanse til noen andre bilder og er kun komprimert med DCT og andre teknikker fra JPEG og MJPEG.

P og B rammer er avhengige av andre rammer, de spesifiserer relative forskjeller fra en eller to referanserammer. P-rammer spesifiserer forskjellen fra forrige I-ramme mens B-rammer gir en interpolasjon mellom forrige og neste I- eller P-ramme.

Bildene deles opp i makroblokker. I B- og P-rammene lagres det bevegelsesvektorer som forteller hvor mye og i hvilken retning en makroblokk har beveget seg siden referanserammen, samt hvor mye verdiene til pikslene har forandret seg.

MPEG-1 oppnår vanligvis mellom 30:1 og 50:1 komprimering. Problemet med MPEG-1 er at komprimeringsalgoritmen krever mye datakraft. Dekomprimering kan gjøres på en 200Mhz prosessor eller bedre i software, men komprimeringen er mye mer komplisert fordi algoritmen må prøve å beregne hvor makroblokkene har flyttet seg fra ett bilde til et annet. Det finnes ikke noen bestemt måte å finne ut dette på, det er opp til den som implementerer å velge en algoritme som får dette til. Det enkleste er å alltid anta at blokkene ikke har flyttet seg, det resulterer i lite behov av datakraft, men kvaliteten blir mye dårligere enn ellers. Dette gjør at sanntids MPEG-1 komprimering i dag stort sett gjøres på maskiner med spesiell hardware for dette.

MPEGs oppdeling i I-, B- og P-rammer fører til ekstra forsinkelse når MPEG skal brukes til liveoverføring. B-rammene er avhengige av neste P- eller I-ramme for

å kunne dekodes, noe som fører til forsinkelse da dekodingen av B-rammene må forsinkes til I- eller P-rammen er dekodet. Det er mulig å bestemme hvor mange B-rammer som brukes og på denne måten holde forsinkelsen på et minimum.

MPEG-2 bygger på MPEG-1 og spesialiserer seg på video med høy kvalitet. Målet er Digital-TV og DVD. Den største forskjellen her er høyere båndbredde, nye lydformater og noen nye protokoller som DSM-CC (se 4 på side 45).

MPEG-4 beskrives under. MPEG-7 er såpass nytt, og omhandler blanding av lyd og bilde med andre medium og er derfor ikke særlig interessant i denne oppgaven.

H.263. H.263[42] ble opprinnelig laget for å overføre video over linjer med båndbredde på rundt 20kbit/s. Men det viste seg etter hvert at H.263 er godt egnet også til høyere bitrater. H.263 bruker mange av de samme teknikkene som MPEG, men med en bitrate på rundt 1Mbit/s er H.263 ca. 30% mer effektiv.

H.263 benytter seg av DCT og bevegelsesvektorer på samme måte som MPEG, men er mer avansert enn MPEG fordi den inneholder en rekke utvidelsesmodus som ikke er påkrevet i en minimal implementasjon. H.263 ble brukt som modell når videokodeken i MPEG-4 skulle designes.

Siden teknikkene i H.263 i stor grad er de samme som i MPEG er det grunn til å tro at kravet til datakraft er omtrent det samme.

MPEG-4. MPEG-4[43] er en MPEG standard beregnet for multimedia på faste og mobile nett. Dette er en veldig stor spesifikasjon delt opp i forskjellige deler, i likhet med de andre MPEG-standardene. Ikke alle delene er standardisert enda, men videokodeken ble standardisert i 2000.

Målet for MPEG-4 var til å begynne med å kunne overføre video til små enheter med veldig lav båndbredde. Det ble innført en rekke nye teknikker for å få til dette. En av de viktigste er at bildet kan deles opp i objekter som kan kodes separat. For eksempel kan bakgrunnen kodes som ett objekt og forgrunnsobjekter, for eksempel en nyhetsoppleser, som et annet. Dette kan medføre stor båndbredde besparing siden bakgrunnen sjelden eller aldri endrer seg. Dette fungerer selvsagt best i situasjoner hvor det er forhåndsbestemt at dette er tilfellet, så innholdet bør her være designet for å ta i bruk denne teknikken.

For å få komprimert bildet bedre har MPEG-4 også muligheten til å bytte ut DCT-med Wavelett-transformer. Dette fungerer ved å se på romlige avhengigheter i stedet for å se på frekvenser innenfor små blokker og skal gi bedre komprimering. MPEG-4 kan brukes til komprimering av video med høy kvalitet. Med høy kvalitet er datakraft kravet til komprimering av MPEG-4 enda større enn MPEG-1.

3.3.2 Kontrolllaget

Kravet til kontrolllaget i dette systemet vil være at klienten må kunne ta kontakt med tjeneren og be den starte og stoppe sending av data.

HTTP - HyperText Transfer Protocol. HTTP er en protokoll som er utviklet for å overføre Hypertekst, og den er kontrollert av w3c[4] under RFC-2616[44]. Protokollen setter ikke noen begrensning for hva den faktisk kan overføre. HTTP protokollen er i utgangspunktet en enkel protokoll. Kommandoene "GET" og "PUT" er de viktigste, og brukes til henholdsvis å hente en fil fra tjeneren og å sende en fil til den. Som parameter til disse kallene sendes en URI (Uniform Resource Identifier), tradisjonelt har disse pekt på en fil på en bestemt maskin, men en slik URI kan også peke på en sanntids datastrøm. En sanntids datastrøm kan derfor startes ved å bruke et vanlig HTTP GET kall med en beskrivelse av strømmen.

Denne bruksmåten begrenser hvordan strømmen kan styres. HTTP kan bare brukes til å starte strømmen, for å stoppe den må forbindelsen brytes (For eksempel ved å bryte en forbindelse mot tjeneren på transportlaget). Hovedgrunnen til at det er slik er at HTTP er en protokoll hvor all tilstandsinformasjon ligger i klienten. Dette innebærer at HTTP-tjenerenen glemmer oppkoblingen med klienten med en gang en fil er overført.

SIP - Session Initiation Protocol. SIP er en signaleringsprotokoll for å sette opp, endre og terminere sesjoner med én eller flere deltakere. Disse sesjonene inkluderer multimedia konferanser over Internett, Internett telefonsamtaler og multimedia distribusjon. Sesjonene bruker enten multicast- eller et unicast-oppkoblinger.

SIP er standardisert gjennom RFC-2543[45] og har en funksjonalitet som kan minne litt om vanlig telefon. En bruker kontakter en eller flere andre brukere. Brukerne blir enige om et felles sett kompatible medietyper og bestemmer seg for å bruke disse. Det er klargjort for infrastrukturer nødvendig for mobilitet (gjennom proxy eller viderekobling) og nedkobling.

Problemet med å bruke SIP i denne oppgaven er at det er en protokoll mer egnet for konferanser. For å sette opp en kobling må først programvaren finne en SIP tjener, spørre denne hvor en bruker identifisert med brukernavn@maskin er, få denne brukeren invitert til samme sesjon som programvaren er i og så bli enig om medietype. Dette er en ganske komplisert prosedyre for å sette opp en kobling mellom to maskiner og vil gjøre programvaren mer komplisert enn nødvendig.

[4] World Wide Web Consortium

DSM-CC - Digital Storage Medium Command and Control. MPEG definerte i MPEG-2 standarden en protokoll med et sett funksjoner for å kontrollere MPEG bit-strømmer. Denne protokollen kalles DSM-CC[46].

DSM-CC er laget for å være uavhengig av underliggende transportprotokoll og bruker CORBA (Common Object Request Broker Architecture) for å implementere interaksjon mellom objekter i et distribuert system. DSM-CC har funksjonalitet kjent fra vanlige fjernkontroller til TV-apparater som “play”, “pause”, “resume” og også mer avanserte funksjoner som “seek”.

DSM-CC er laget spesielt for MPEG data. Det er i tillegg beregnet for å brukes til digital-TV. Det er ønskelig å kunne teste forskjellige videokodeker, dette gjør DSM-CC lite egnet som applikasjonslagprotokoll for denne oppgaven.

RTSP - Real-Time Streaming Protocol. RTSP er en protokoll definert i RFC 2326[47] for å styre sanntids multimediastrømmer utviklet av RealNetworks, Netscape Communications og Columbia University. Den bygger på erfaringer RealNetworks hadde med RealAudio og Netscape hadde med LiveMedia. RTSP er en klient-tjener multimediapresentasjonsprotokoll som muliggjør kontrollert leveranse av multimedia strømmer over IP nettverk. Den har funksjonalitet som avspilling, pause, spoling og posisjonering. Datakilden kan enten være sanntidsstrømmer eller lagrede klipp. RTSP er i dag støttet av blant andre Netscape, Apple, IBM, SGI, VXtreme og SUN.

RTSP er basert på HTTP 1.1 for å dra nytte av utviklingen av HTTP protokollen. Men RTSP skiller seg fra HTTP på flere måter. RTSP opprettholder tilstander for hver sesjon, for å kunne koble fremtidige forespørsler til en eksisterende strøm. RTSP symmetrisk, det vil si at tjeneren kan sende forespørsler til klienten.

I en enkel bruk av RTSP-protokollen begynner klienten med å sende et “DESCRIBE” kall med en multimediastrøm som parameter til tjeneren. Tjeneren svarer med å sende en beskrivelse av multimediastrømmen i SDP[5]-format. Klienten bruker denne informasjonen til å sette opp kodeker og sender så et “SETUP”-kall til tjeneren med parametre som beskriver hva slags overføring som er ønsket. Tjeneren svarer med hvilken overføring den støtter og setter opp en oppkobling for overføring av multimediadata, vanligvis med RTP. Når klienten sender et “PLAY”-kall til tjeneren begynner den å sende strømmen til klienten. Oppkoblingen brytes ved å sende en “TEARDOWN” kommando.

Ved å bruke RTSP som applikasjonsprotokoll kan klienten kommunisere med andre eksisterende RTSP-tjenere. En informasjonsside[48] er satt opp med linker til dokumenter om RTSP og implementasjoner av det.

[5] Session Description Protocol

Sammenligning. Tabell 4 sammenligner de forskjellige applikasjonslagsprotokollene ved å sette opp kriterier som er viktige for denne oppgaven.

	HTTP	SIP	DSM-CC	RTSP
Beregnet for Multimedia	Nei	Ja	Ja	Ja
To veis protokoll	Nei	Ja	Ja	Ja
Kodek støtte	Alle	Alle	MPEG-2 og MPEG-4	Alle
I utstrakt bruk	Ja	Nei	Ja, i DVB	Ja
Enkel virkemåte	Ja	Nei	Nei	Ja

Tabell 4: Sammenligning av applikasjonslagsprotokoller

HTTP er best til henting av filer, slik den brukes på web. SIP er riktig protokoll å bruke i et videokonferansesystem, eller til andre konferansesystem og systemer som prøver å etterligne telefoni. DSM-CC er systemet som bør brukes til DigitalTV.

RTSP er valgt som protokoll i denne oppgaven fordi dette er en protokoll beregnet for styring av sanntidsstrømmer og det er lett å finne ferdige implementasjoner og applikasjoner som benytter seg av denne til videooverføring. Muligheten til å fleksibelt kunne bytte kodek i protokollen har også telt positivt for RTSP.

3.3.3 Transportlaget

Det er i dag en rekke forskjellige transportprotokoller i bruk på Internett, men alle har forskjellige bruksområder. Det er derfor viktig å se nøye på hvilke bruksområder de forskjellige protokollene har før det velges hvilken protokoll skal benyttes.

TCP - Transmission Control Protocol. TCP er den mest brukte transportprotokollen på Internett i dag fordi den garanterer at data kommer fram i riktig form og rekkefølge. TCP gir en pålitelig, koblingsorientert bytestrømstjeneste som har blitt forbedret til å takle og forhindre metning i nettet.

TCP baserer seg på at mottakeren bekrefter at pakker er mottatt, dersom bekreftelsen uteblir sendes pakken på nytt. Ved overføring av multimedia er det ofte viktigere at data kommer fort fram enn at alt kommer fram. Dersom mottaker må vente på gjentatte retransmisjoner vil fremvisingen etterhvert bli forsinket, dette har ført til oppfatningen om at TCP er uegnet til sanntidsoverføring av multimedia, særlig live multimedia[31], hvor det ofte er mer ønskelig å kaste gamle pakker enn å vente på dem[35]. I dag er forsinkelsen i nettet mindre

slik at effekten ikke er like stor, TCP er derfor mer egnet til overføring av sanntidsvideo[49].

UDP - User Datagram Protocol. UDP er en enklere protokoll enn TCP som kan brukes til å demultiplexe strømmen av innkommende pakker til en maskin og sende dem til riktig applikasjon ved bruk av port-nummer, samt å sjekke at pakkene ikke er ødelagt. UDP garanterer ikke for at pakker kommer fram eller at rekkefølgen beholdes og gir heller ingen informasjon om når pakken ble sendt.

I en multimediasammenheng er fordelen med UDP framfor TCP at applikasjonen ikke må vente på pakker som blir borte. Dette gjør at data kan behandles med en gang den kommer inn slik at forsinkelsen kan holdes på et minimalt nivå. Ulempen er at pakker kan komme frem i feil rekkefølge og gjøre det vanskelig å dekode den mottatte strømmen da mottakeren ikke vet at rekkefølgen er feil. Det finnes heller ikke informasjon om hvor gamle pakkene er, noe som blant annet gjør det umulig å vite om det er en metning i nettet. Ved metning i nettet bruker pakkene lenger og lenger tid i køer på vei gjennom nettet, etterhvert blir pakkene kastet. Senderen merker ikke dette, og kan derfor ikke justere senderaten og hindre metningen.

RTP - Real-time Transport Protocol. RTP kan brukes over en valgfri pakkebasert protokoll. For eksempel kan IP benyttes, men vanligvis benyttes UDP for å dra nytte av multiplexings- og feilsjekkfunksjonaliteten der. RTP har karakteristikk både fra transportlaget og applikasjonslaget.

RTP er en protokoll som gir støtte for overføring av sanntidsdata som video- og lydstrømmer. Tjenestene RTP tilbyr inkluderer tidsrekonstruksjon, oppdaging av tap, sikkerhet og innholdsidentifikasjon. RTP er hovedsakelig designet for multicastapplikasjoner, men kan også brukes til unicast. RTP er ment å brukes sammen med RTCP (Real-time Transport Control Protocol), som er en protokoll som muliggjør detaljert styring av strømmen.

I RTP numreres pakkene slik at mottakeren kan sette dem sammen i riktig sekvens. Pakkene tidsstemples også, noe som gjør det mulig å synkronisere flere strømmer (som en video- og en lydstrøm) eller å finne ut hvor lang tid det tar fra en pakke blir sendt til den blir mottatt.

RTP gir ingen garanti for at pakker kommer fram, så det er opp til applikasjonen å velge hva som skal gjøres ved pakketap eller forsinkede pakker. Det beste er ofte å bare glemme pakken som ble borte. RTP har også en innholdsidentifikasjon, som sier hva pakken inneholder. Det er definert en rekke slike identifikatorer for vanlige formater som PCM, MPEG1/MPEG2 lyd og video, JPEG video, H.261

	TCP	UDP	RTP
Pakkebasert	Nei	Ja	Ja
Garantert levering	Ja	Nei	Nei
Sekvensnumrering av pakker	Ja	Nei	Ja
Tidsstempling av pakker	Nei	Nei	Ja
Beregnet for Multimedia	Nei	Nei	Ja
I utstrakt bruk	Ja	Ja	Ja

Tabell 5: Sammenligning av transportlagsprotokoller

video med flere. Det er også meningen at applikasjonsutviklere skal kunne legge til egne formater dersom det er nødvendig.

RTCP brukes sammen med RTP og gir mulighet til å gi tilbakemelding om pakketap, forsinkelse med mere til senderen. RTCP kan derfor brukes for å unngå metning ved at senderen sender ut mindre data dersom mottakeren mister mange pakker eller at de er forsinket.

RTP er en åpen protokoll publisert under RFC 1889 [50]. RTP tilbyr ingen pre-definerte systemkall slik som TCP, som ofte er implementert i kjernen til operativsystemet. RTP implementasjonen er knyttet til selve applikasjonen, og implementasjonsvalg gjøres der. RFC 1889 inneholder en del kode som kan brukes som utgangspunkt og det finnes en del åpne implementasjoner som kan brukes.

Sammenligning. Tabell 5 sammenligner de overnevnte transportlagsprotokollene. Det er satt opp kriterier som er viktige for denne oppgaven.

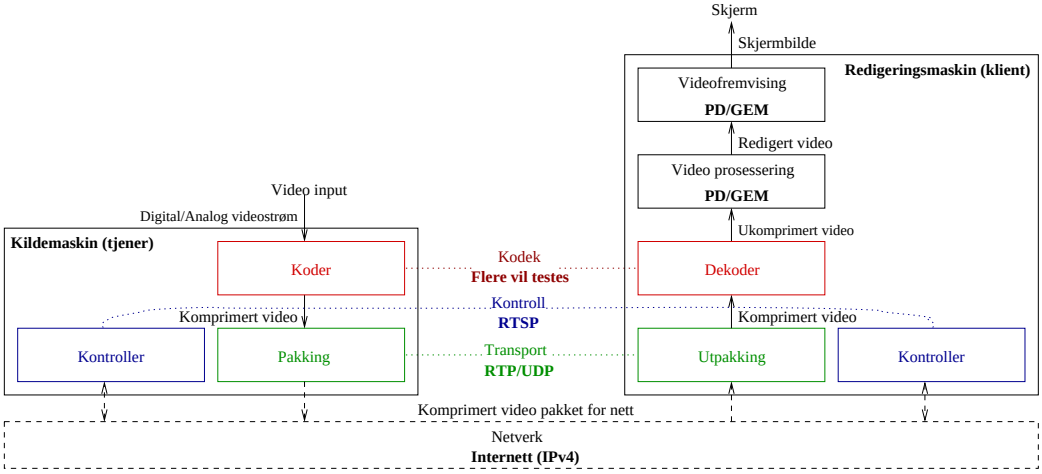
Valget faller på RTP, siden det er vanlig å bruke RTP sammen med RTSP. Kombinasjonen RTSP/RTP vil gjøre det enklere å koble systemet opp mot allerede eksisterende multimediasystemer.

Hittil er GEM valgt som videoredigeringsprogram, RTP valgt som transportprotokoll og RTSP som styringsprotokoll. I tillegg trengs en kodek til å komprimere video, her er det ønskelig å teste flere for å se hvilken som gir best kvalitet og forsinkelse. Figur 14 på neste side oppsummerer valgene så langt.

3.4 Valg av programvare

Både RTP og RTSP er kompliserte protokoller som vil ta lang tid å implementere, det vil også ta lang tid å implementere en videokodek. I stedet benyttes ferdige implementasjoner ved å integrere disse i GEM-objektet.

3.4 Valg av programvare



Figur 14: Oppsettet med protokoller og redigeringssystem valgt. De valgte elementene er med fet skrift.

3.4.1 FFMPEG

FFMPEG[51] er en programpakke som kan brukes til å lagre til disk, konvertere og å send lyd og video som strømmer. Den inkluderer biblioteket libavcodec, som brukes til komprimering og dekomprimering i mange videoprogramvarepakker med åpen kildekode. FFMPEG utvikles under Linux, men kan biblioteket kan brukes i de fleste operativsystemer, inkludert Windows.

Prosjektet består av følgende komponenter:

- **ffplay** er et enkelt program for avspilling av video- og lydfiler.
- **ffmpeg** er et kommandolinje verktøy for å konvertere en videofil fra ett format til et annet. Det kan også brukes til grabbing og koding i sanntid fra et TV-kort.
- **ffserver** er en HTTP (RTSP er under utvikling) multimedia streaming server for live kringkasting.
- **libavcodec** er et bibliotek som inneholder alle FFMPEGs audio og videokodere og dekodere (kodeker). De fleste av disse kodekene er utviklet fra bunnen av for å påse god ytelse og størst mulig gjenbruk av kode mellom de forskjellige kodekene.

- **libavformat** er et bibliotek som inneholder parsere og generatorer for alle vanlige audio- og videoformater, samt rutiner for lesing fra nett over HTTP og RTSP.

Tabell 6 viser de viktigste videokodekene FFMPEG støtter. Fordelen her er at FFMPEG støtter veldig mange kodeker slik at flere kan testes for å undersøke hvilken som gir minst forsinkelse og best bildekvalitet.

Støttet Kodek	Koding	Dekoding	Kommentar
MPEG1 video	X	X	
MPEG2 video	X	X	
MPEG4	X	X	Også kjent som DIVX4/5
MSMPEG4 V1	X	X	
MSMPEG4 V2	X	X	
MSMPEG4 V3	X	X	Også kjent som DIVX3
WMV7	X	X	
WMV8	X	X	Ikke helt ferdig
H263(+)	X	X	Også kjent som Real Video 1.0
H264		X	
MJPEG	X	X	
Tapsfri MPEG	X	X	
DV	X	X	
Huff YUV	X	X	
			"X" betyr støttet.

Tabell 6: Oversikt over kodeker i FFMPEG

I utgangspunktet inneholder FFMPEG flesteparten av de komponentene som trengs til denne oppgaven. FFMPEG er dessuten et bibliotek som er mye brukt i annen programvare, den er derfor godt testet. Men ved nøye undersøkelse kommer flere mangler frem i lyset. Det største problemet i forbindelse med dette prosjektet er RTP/RTSP støtten. Denne er under utvikling. Tester viste at det var mulig å bruke ffserver til å sende filer som videostrømmer over RTSP/RTP.

FFMPEG kan benyttes som en RTP/RTSP tjener. Videre kan libavcodec benyttes i GEM-objektet til å dekode video. Det trengs derfor ytterligere programvare som kan ta imot en RTSP/RTP strøm og sende denne til libavcodec.

3.4.2 MPlayer

MPlayer[52] er en av de mest brukte mediaavspillerene i Linux. Den kan spille av de aller fleste videofiler, samt å spille fra DVD, VCD og videostrømmer samt filer

fra nett, blant annet med RTSP/RTP. MPlayer er et stort prosjekt og er interessant fordi det støtter RTSP/RTP og benytter seg av blant annet FFMPEG. RTSP/RTP støtten er implementert ved å bruke biblioteket liveMedia fra live.com. MPlayer er hovedsakelig et C prosjekt mens live.com er skrevet i C++, implementasjonen kan derfor gjøres på samme måte i GEM, som også er skrevet i C.

MPlayer kan også brukes til å teste RTSP/RTP. Ved å teste MPlayer mot ffserver kan vil det klargjøres om implementasjonene av RTSP og RTP i MPlayer(liveMedia) og FFMPEG er kompatible før disse implementeres i prosjektet.

3.4.3 live.com - liveMedia

liveMedia[53] er en del av en pakke kalt live.com, som inneholder et sett C++ biblioteker beregnet på multimediastreaming. liveMedia tar seg av RTP/RTCP-, RTSP- og SIP-støtten i live.com. Bibliotekene kan kompileres på Unix (inkludert Linux og Mac OS X), Windows, QNX og andre POSIX systemer.

liveMedia vil her bli brukt til å ta imot RTP strømmen fra tjeneren og sende strømmen videre til FFMPEG til dekoding.

3.4.4 QuickTime Streaming Server

Apple har laget en løsning for å sende lyd og video over nett. Denne løsningen er bygd opp av flere programmer:

- **QuickTime Player** er en gratis applikasjon for avspilling av video, lyd, VR og grafikk som er kompatibel med QuickTime. QuickTime Player kan spille fra fil eller fra nettverk.
- **QuickTime Streaming Server**(QTSS) er et program som kan sende media over Internett i sanntid eller på forespørsel. Tjeneren beregnet på at brukeren skal kunne se media så fort den når maskinen, uten å vente på nedlasting. QTSS benytter seg av RTSP/RTP for å sende strømmene over nett. QTSS følger med Mac OS X Server.
- **Darwin Streaming Server** er en versjon av QTSS med åpen kildekode som kan benyttes på Linux, Solaris og Windows NT eller nyere. Forskjellen på Darwin Streaming Server og QTSS er at det følger grafiske administrasjons-verktøy med QTSS.

- **QuickTime Broadcaster** er applikasjon som komprimerer video i sanntid og sender den komprimerte strømmen til en maskin over internett. QuickTime Broadcaster støtter de fleste kodekene QuickTime støtter, og følger med Mac OS X Server i likhet med QTSS.

QuickTime Broadcaster har minimal støtte for sending over nettverk, den er derfor beregnet brukt sammen med QTSS. Sammen muliggjør de å sende komprimerte sanntidsstrømmer fra tilkoblet kamera over RTSP/RTP. QuickTime Broadcaster kan kun benyttes under Mac OS, noe som fører til at en Apple-maskin kreves for å benytte denne kombinasjonen.

3.4.5 Kompatibilitet mellom FFserver og MPlayer

For å sjekke at RTP/RTSP støtten i FFMPEG og liveMedia var kompatibel med hverandre ble det satt opp et lite testsystem. Testsystemet bestod av to maskiner: En maskin med FFserver, en med MPlayer.

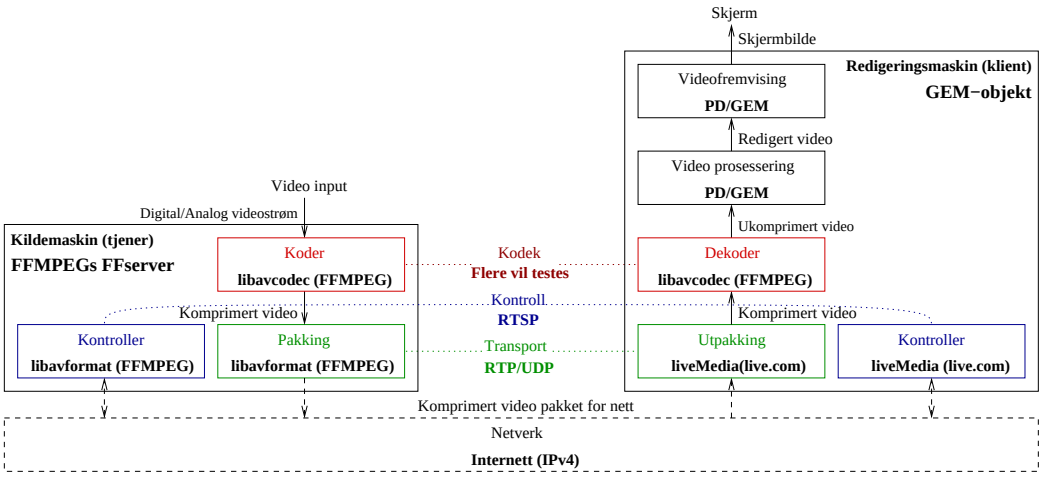
FFserver. FFserver ble installert fra FFMPEG CVS. Tjeneren ble konfigurert til å tilby en fil med MPEG-1 video uten lyd. RTSP/RTP ble valgt som protokoller. Tjenerenen ble startet og ble stående i lyttemodus og ventet på at noen ville spille av filen.

MPlayer. MPlayer ble kompilert med "–enable-live" som parameter for å lenke inn liveMedia biblioteket. Når dette var gjort ble MPlayer startet med rtsp-strømmen på den andre maskinen som parameter. MPEG filmen ble spilt av uten problemer. Ingen av maskinenene viste feilmeldinger, alt av tilbakemeldinger fra FFserver og MPlayer viste at overføringen fungerte.

Konklusjon. Kompatibilitetstesten viste at programmene er kompatible med hverandre og at overføring og avspilling fungerer. Dette vil hjelpe under utviklingen fordi den kan deles i opp i to faser, en for tjeneren og en for klienten. Tjeneren kan testes mot MPlayer, klienten mot FFserver.

Resultatet så langt er nå en arkitektur for sanntidsvideoredigering, vist i figur 15 på neste side. FFserver vil bli brukt i sin helhet som tjener. På klientsiden vil det utvikles et GEM-objekt, som bruker liveMedia til å kommunisere med tjeneren og overføre komprimert video som senere blir dekodet av libavcodec og sendt videre i GEM-grafen.

3.4 Valg av programvare



Figur 15: Foreslått arkitektur for et sanntidsvideoredigeringssystem. Valgte komponenter og teknologier med fet skrift.

Etter at FFserver ble forsøkt brukt som tjener for sending av sanntidsstrømmer over RTSP/RTP ble det klart at støtten for dette var alt for begrenset. Implementasjonen klarer kun å sende lagrede filer, ikke strømmer fra kamera. FFserver ble derfor droppet til fordel for QuickTime Streaming Server fra Apple, som også ble testet mot MPlayer for å sikre kompatibilitet med liveMedia og libavcodec.

Modell:	Selvkomponert PC
Prosessor:	Intel Pentium 4 1800 Mhz
Minne:	1024 MB RDRAM
Harddisk:	IC35L010AVER07-0
Grafikk:	NVIDIA GeForce2 MX 400
Operativsystem:	Debian Unstable m/Linux 2.4.24

Tabell 7: Spesifikasjon av redigeringsmaskin.

4 Utvikling av arkitektur for redigering

I forrige kapittel ble en arkitektur for redigering av distribuert sanntidsvideo valgt. Dette kapittelet omhandler utviklingen og implementering av denne arkitekturen og hvilke erfaringer som ble gjort underveis. For å få forsinkelsen ned til et minimum, ble en rekke parametre i tjeneren justert, og konsekvensene av endringene målt og dokumentert.

Kapittel 5 viser resultatet av målinger foretatt med tre forskjellige videokodeker. Disse blir sammenlignet med tanke på kravene som er listet i innledningen. Særlig viktig er forsinkelse, da systemet skal brukes i en sanntidssituasjon.

For å gjøre utviklingen av redigeringssystemet til en overkommelig oppgave deles oppgaven i to deler. Den ene delen er tjeneren, den andre er klienten. Grunnen til å gjøre denne oppdelingen er at dette er to separate systemer som kan virke uavhengig av hverandre og testes mot tilgjengelig programvare, siden det er valgt standard protokoller og kodeker for overføring og koding av videostrømmen.

Klienten vil testes mot FFserver, som kan sende video til den. Tjeneren, som skal kjøre på maskinen med kameraet, vil testes mot MPlayer.

4.1 Redigeringsmaskin

Redigeringsmaskinen er beskrevet i tabell 7.

Utviklingen av programvaren til redigeringsmaskinen deles opp for å muliggjøre testing underveis, og for å få bedre forståelse av hvordan komponentene som benyttes fungerer.

Applikasjonen ble utviklet i flere steg:

1. Evaluasjonsfase.
2. GEM-objekt som bruker FFMPEGs libavcodec til å dekode en MPEG-strøm.

3. GEM-objekt utvidet til å lese video fra nett over RTSP/RTP.
4. GEM-objekt utvidet til å støtte flere kodeker.

Utviklingen ble oppdelt på denne måten slik at feil kunne oppdages tidlig og for å teste de forskjellige programvarepakkene i praksis og lære hvordan de virker.

4.1.1 Evaluasjonsfase

I evaluasjonsfasen ble det utviklet et GEM-objekt som genererte tilfeldige bilder som kunne brukes i et PD/GEM oppsett. Målet var å lære å utvikle et GEM-objekt som senere kunne utvides til å ta inn video fra nett.

GEM versjon 0.87 og PD versjon 0.37 ble installert fra bunnen av med kildekode og kompilert uten noen patcher. Kombinasjonen PD/GEM ble testet ved å kjøre noen medfølgende eksempler.

Navnet på GEM-objektet er valgt til pix_videoRTP. Som mal ble pix_videoLinux valgt, siden denne hadde funksjonalitet som liknet mest på den som skulle utvikles. Figur 16 på neste side illustrerer hvordan det ferdige GEM-objektet var tenkt oppbygd. Figur 17 på neste side illustrerer hvordan et forenklet GEM-objekt, som kun viser tilfeldiggenererte bilder, er oppbygd og hvor den passer inn i en enkel GEM-graf.

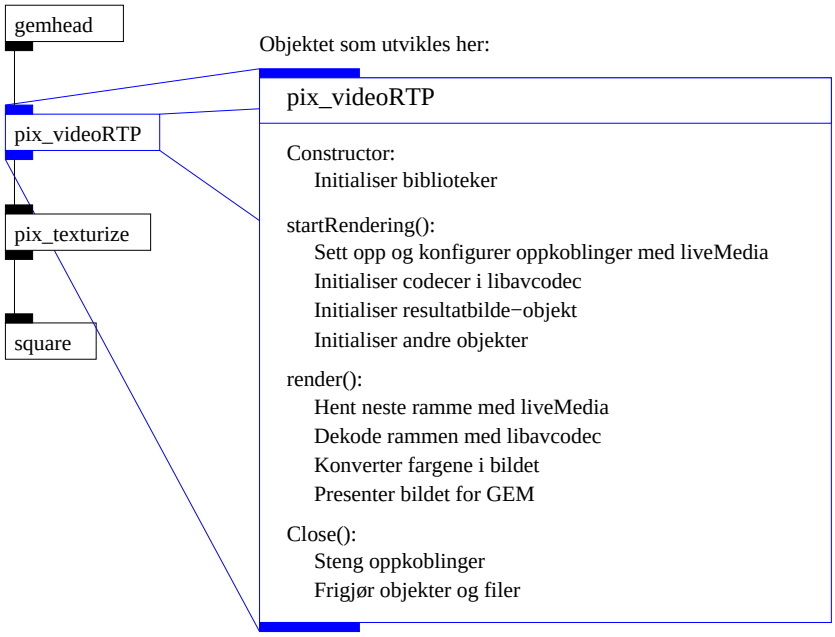
Filene som må lages er pix_videoRTP.h og pix_videoRTP.cpp. I tillegg må GEM-kildekoden endres slik at den registrerer det nye GEM-objektet ved start.

Når et GEM-objekt lages, kalles en konstruktørmetode hvor variabler og eventuelle eksterne objekter opprettes og initialiseres.

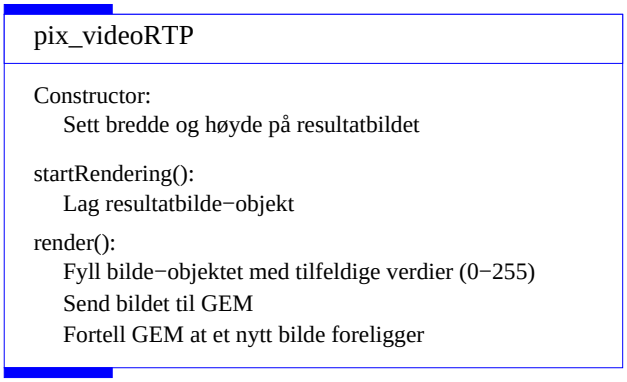
Når brukeren velger å starte bildeopptegning i GEM, kalles funksjonen `startRendering()` som definerer blant annet størrelsen og fargeformat på bildet og allokerer buffere som inneholder bildeinformasjon.

GEM bruker teksturformater fra OpenGL til å beskrive hvordan bildene lagres i minnet. I GEM er GL_RGB og GL_RGBA (samme som RGB, men med verdi for gjennomsiktighet) mest brukt. Siden bildet skal brukes som en OpenGL-tekstur, må både vertikal og horisontal størrelse være av en toerpotens og lagres fra nederste venstre hjørne. I motsetning lagrer "scanline"[6]-baserte formater bildene fra øverste venstre hjørne. Videobildet fra libavcodec må derfor speiles om X-aksen før bruk og teksturen må klippes, ved å definere hvilken del av det som inneholder bildeinformasjon, for å kun få med videobildet.

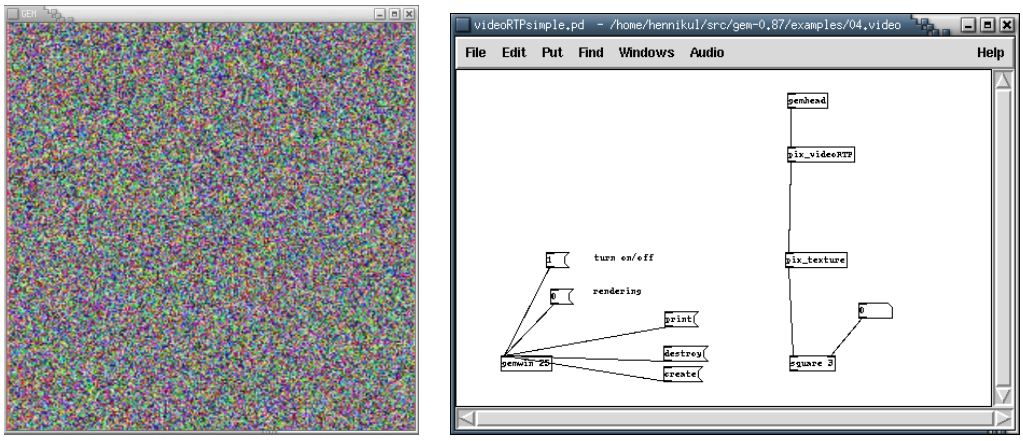
[6] Linjene tegnes ovenfra og ned, en linje om gangen



Figur 16: Design av GEM-objekt som leser video med liveMedia og dekoder med libavcodec.



Figur 17: Design av GEM-objekt som viser tilfeldige bilder.



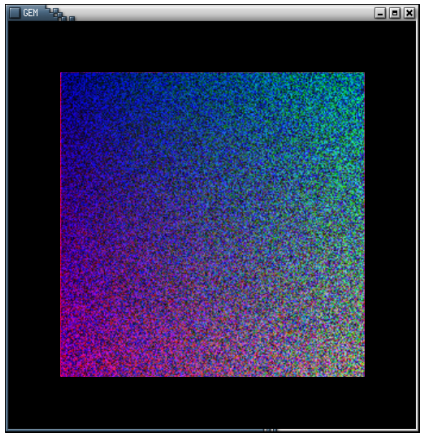
(a) Output (b) Oppsett

Figur 18: Skjermbilde med første versjon av GEM-objektet(venstre). Genererer et bilde med tilfeldige piksler 25 ganger i sekundet. Bildet til høyre viser oppsettgrafen i GEM.

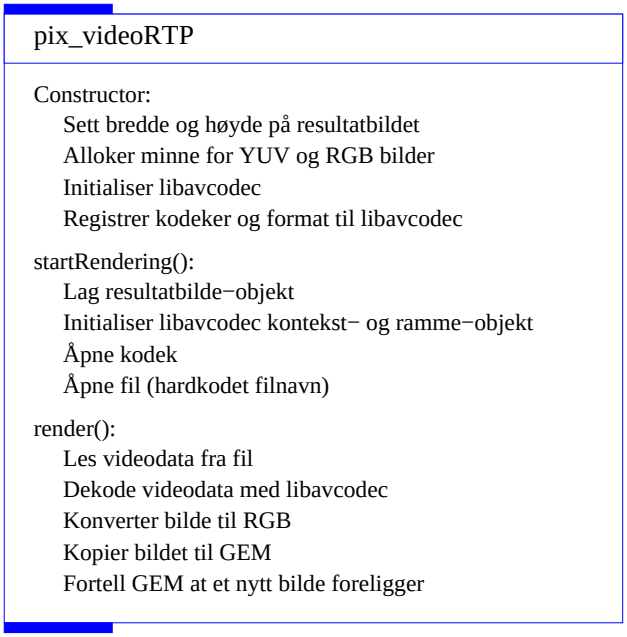
Under kjøringen kaller GEM funksjonen `render()` i GEM-objektet et bruker-styrt antall ganger i sekundet. Denne funksjonen har som oppgave å generere et bilde som GEM kan bruke videre i oppsettgrafen. I evalueringsfasen ble funksjonen `render()` implementert slik at det lages med tilfeldige piksler.

Figur 18(a) viser resultatet. Figur 18(b) viser GEM-oppsettet som er brukt. “gem_win”-objektet er vinduet hvor output vil bli vist, det har “25” som parameter for at GEM skal kalle `render()`-funksjonen 25 ganger i sekundet (PAL-frekvensen uten sammenfletting[7]). Høyre del av figur 18(b) viser PD/GEMs representasjon av en OpenGL-liste som defineres ved at første objekt er et gemhead-objekt. pix_videoRTP-objektet genererer bildet, et pix_texture-objekt gjør dette om til en tekstur og square-objektet legger denne på et kvadrat.

GEM-objektet ble utvidet videre slik at `render()`-funksjonen genererer et bilde som har mer grønnfarge i høyre del og mer rødfarge i nedre del. Resultatet i figur 19 på neste side bekrefter at bildet tegnes fra nederste venstre hjørne og oppover.



Figur 19: Skjermbilde med andre versjon av GEM-objektet. Viser et bilde med tilfeldige piksler hvor sannsynligheten for fargen i de forskjellige hjørnene er forhåndsbestemt.



Figur 20: Design av GEM-objekt som dekoder video med libavcodec.

4.1.2 GEM-objekt som dekoder video

Designet ble utvidet slik at GEM-objektet skulle kunne dekode en fil ved hjelp av FFMPEG's libavcodec, som illustrert i figur 20 på forrige side. Dessverre er dokumentasjonen av libavcodec meget mangelfull. FFMPEGs "ffplay", en enkel videoavspiller som bruker libavformat og libavcodec, ble derfor brukt som eksempel for hvordan libavcodec brukes for å dekode en videostrøm lest fra fil.

FFMPEG består av bibliotekene libavcodec og libavformat[8]. libavcodec ble integrert i GEM ved at kildekodetreet til libavcodec, fra FFMPEG versjon 0.4.6, ble kopiert inn i kildekodetreet til GEM og linket statisk. Dette er en enkel måte å koble inn et bibliotek, men det gjør det vanskeligere å dra nytte av ny funksjonalitet og bedret stabilitet i nyere versjoner av biblioteket, da integrasjonen må gjøres på nytt hver gang. Fordelen er at forandring i API til nye versjoner ikke vil virke inn på GEM-objektet.

For å lese video fra fil ble en ny funksjon `render()` laget. Denne leser videofilen rått inn i et buffer med `fread()`-kall og sender så bufferet til `avcodec_decode_video()`, som dekoder videostrømmen og returnerer et bilde dersom et helt bilde er lest fra strømmen.

I MPEG-video benyttes ikke RGB-bilder, der benyttes vanligvis YUV420. OpenGL støtter en slik fargemodell, men den er ikke tilgjengelig på alle plattformer, og GEM støtter det ikke. Derfor må bildet konverteres fra YUV420 til GL_RGB. Dette kan gjøres med libavcodecs `img_convert()`. Teksturkoordinater ble brukt for å forsøke å snu bildet, og for å kutte vekk sort kant rundt bildet.

Resultatet er vist i figur 21 på neste side. Bildet er opp-ned, uten farger, stripete og den sorte kanten rundt bildet er ikke borte. Dette tyder på at bruken av teksturkoordinater ikke virker og at det er problemer med bruken av fargekonvertering. Det som virket var å lese inn rammer fra MPEG-filen, og å dekode disse.

4.1.3 GEM-objekt som kan lese video via RTSP/RTP

Mens utviklingen pågikk begynte hovedutvikleren av GEM å integrere FFMPEG i den uoffisielle CVS-versjonen[9], som senere skulle bli versjon 0.90, av GEM. Denne ble installert og testet. Dessverre viste det seg at koden lot seg kompilere, men virket ikke.

[7] Engelsk: Interlacing
[8] Se avsnitt 3.4.1 på side 49
[9] Se ordforklaringer bakerst

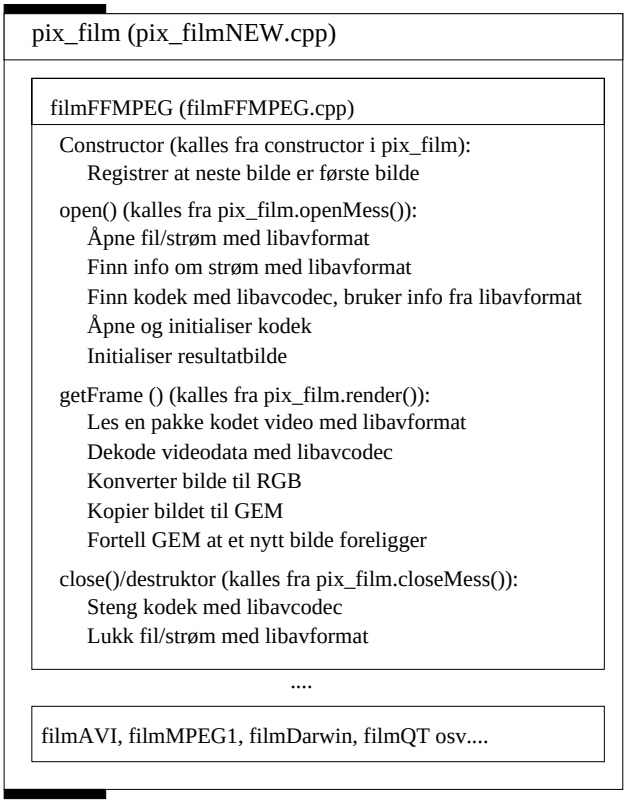
Figur 21: Skjermbilde med tredje versjon av GEM-objektet. Viser en forhåndsbestemt og meget enkel videostrøm (MPEG-ES uten lyd).

I det nye GEM-objektet var FFMPEG integrert ved å la koden linkes mot eksternt kompilerte FFMPEG-objektfiler i stedet for å legge kildekodetreet inn i GEM. libavformat var også integrert i tillegg til libavcodec, slik at flere mediafiler og lesing fra nett skulle være mulig. Det ble valgt droppe arbeidet med det gamle GEM-objektet og heller fortsette utviklingsarbeidet GEM-utviklerne hadde påbegynt, siden de hadde gjort mye arbeid for å gjøre integreringen bedre.

FFMPEG ble oppgradert til versjon 0.4.8, som var nyeste stabile. libavformat hadde i denne versjonen fått bedre støtte for RTSP/RTP. Dette førte til en beslutning om å droppe liveMedia, da det ble antatt at dette ville føre til mindre integrasjonsarbeid.

Det nye GEM-objektet heter filmFFMPEG, og brukes ved å legge inn et pix_film objekt i PD/GEM oppsettetgrafen. Figur 22 på neste side illustrerer hvordan dette GEM-objektet fungerer. pix_film kan også lese filtyper FFMPEG ikke støtter ved å bruke andre objekter som filmQT og filmAVI.

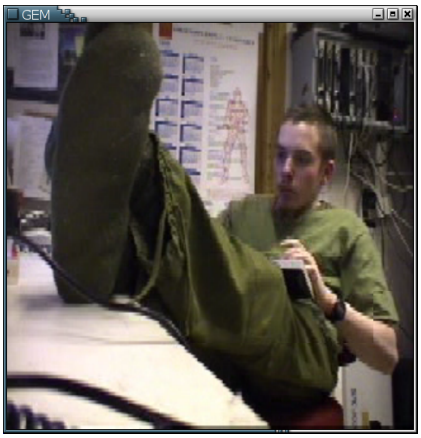
Problemet med å åpne filer skyldtes at et kall på `av_register_all()` var nødvendig for at FFMPEG skulle vite om alle filformater og kodeker. Ved å legge inn dette kallet kan GEM åpne forskjellige filer og vise video fra dem. For å få bildet til å vises riktig vei må i tilleg et upside_down flagg settes. Bildet ble gjort om til ønsket fargemodell, RGBA (RGB med en ekstra kanal til gjennomsiktighet), ved å kjøre `fromBGR()` i GEM fordi FFMPEGs PIX_FMT_RGBA32-format egentlig lagrer fargene i BGRA-rekkefølge på little-endian maskiner, og ARGB på big-endian. Etter disse endringene er resultatet som i figur 23 på side 62.



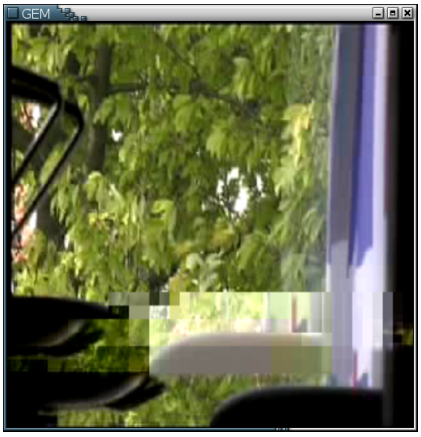
Figur 22: Design av GEMs filmFFMPEG i pix_film.

Det var nå mulig å spille av videofiler fra disk og filer fra en nett over HTTP. Siden FFMPEG 0.4.8 ikke har god nok støtte for RTSP/RTP, må CVS-versjonen benyttes. Med denne fungerte GEM mot QuickTime Streaming server med RTSP/RTP over TCP og MPEG4 som kodek. Forsinkelsen fra kamera til framvisning var på over 10 sekunder og bildet viste til tider problemer med digital støy nederst, som vist i figur 24 på neste side.

I `getFrame()` sjekkes variabelen m_readNext. Dersom denne er sann, blir neste bilde hentet, hvis ikke returnerer funksjonen med en peker til det gamle bildet. m_readNext blir satt automatisk av GEM, dersom "auto" er satt, dette er en parameter som forteller pix_film at bilder skal byttes automatisk. Dersom en annen tråd setter denne variabelen til sann i tidsrommet mellom den sjekkes og settes til falsk i `getFrame()`, for å indikere at neste bilde ikke skal leses før den er satt til sann igjen, henger programmet. GEM vil ikke lenger prøve å sette den til sann, siden den allerede har gjort det, og den vet ikke at en annen tråd har satt den til falsk. Oppdatering av denne variabelen er en kritisk region som må låses.



Figur 23: Skjermbilde fra filmFFMPEG i GEM. Viser en samme fil som i figur 21 på side 60.



Figur 24: MPEG4 over RTP, using libavformat in GEM. Feilene i bildet er tydelige.

JPEG over RTP i libavformat. I CVS-versjonen støttet ikke libavformat andre kodeker enn MPEG4 over RTP. JPEG over RTP krever et eget pakkeformat[39] som ble implementert i FFMPEG ved å legge til følgende funksjonalitet:

1. Når en pakke leses inn må JPEG headeren, som ligger etter den vanlige RTP-headeren, leses og sendes til dekoderen.
2. Et komprimert bilde (ramme) er delt opp i flere RTP-pakker. Den siste pakken må detekteres slik at pakkene kan settes sammen til en hel ramme.

RTP headeren har et "marker bit" som er satt av til å markere en spesiell hendelse. Med JPEG over RTP er denne satt til "1" i siste pakke av en ramme og kan derfor brukes til å sette sammen pakkene.

Dekoderen i libavcodec forventer et standard JFIF JPEG bilde. JPEG over RTP legger data og headerinformasjon slik at bildet lettere kan dekodes selv om deler blir borte etter at det er delt opp i små pakker. Siden MPlayer klarer å spille JPEG-strømmer over RTP, ble kildekoden undersøkt, der genereres JFIF-headere utifra RTP- og JPEG-headerene ved hjelp av liveMedia. For å implementere dekoding av JPEG over RTP ble funksjonen `createJPEGHeader()` fra liveMedia brukt. Siden denne er implementert i C++ og libavformat er implementert i C, måtte koden dessuten oversettes til C.

Kvaliteten på video ble bedre enn med MPEG4, fremvisningen hakket mindre, bildet var mindre blokkete og hyppigheten av feil i bunnen av bildet var mindre. Det viste seg at forsinkelsen fra kamera til fremvisning fremdeles var på over 10 sekunder, noe som gjør dette oppsettet lite egnet for sanntids-applikasjoner. Ved å finstille QuickTime Streaming Server (se 4.2.3 på side 71) ble denne forsinkelsen redusert.

RTP over UDP i libavformat. RTP-strømmene ble sendt over TCP i stedet for UDP. UDP-støtten i FFMPEG var slått av som standard, og slås på ved å kommentere vekk en linje i kildekoden. Avspillingen over UDP viste mindre feil i bildet og forsinkelsen var lavere.

UDP er en forbindelsesløs protokoll hvor mottakeren ikke gir tilbakemelding på mottak av pakker. Senderen kan derfor ikke vite om mottakeren eksisterer. For å hindre at ressurser kastes bort ved å sende til ikkeeksisterende mottakere, skal tilbakemeldinger om mottak sendes jevnlig med RTCP når RTP benyttes. QuickTime Streaming Server har en opsjon "rtp_timeout", med standardverdi 120 sekunder, som bestemmer hvor lenge tjeneren venter på en RTCP-pakke før den avslutter oppkoblingen[54].

Fra koden til libavformat er det tydelig at RTCP kun er støttet i senderdelen, ikke i mottakerdelen. Derfor blir ingen RTCP pakkeer sendt fra mottaker og i tjeneren (QTSS) tolkes det som om mottakeren ikke mottar pakker, med det resultat at tjeneren avslutter sesjonen etter to minutter.

Konklusjonen blir at FFMPEG ikke er klar for å sende RTSP/RTP-strømmer over RTSP/RTP.

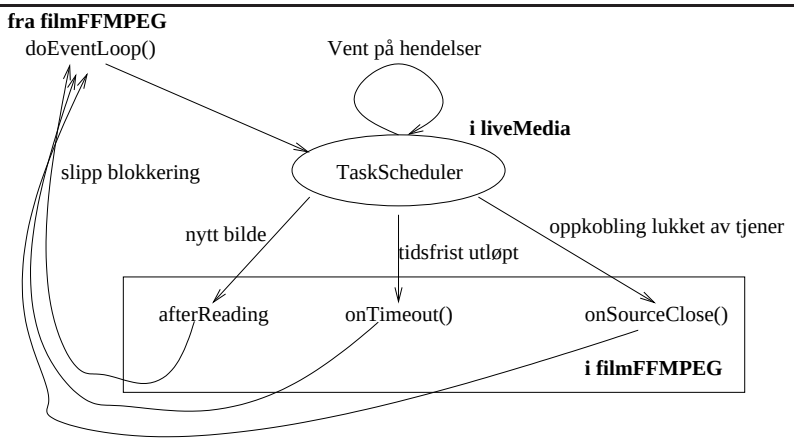
liveMedia isteden for libavformat. I motsetning til libavformat, som viste seg å være vanskelig å integrere, har liveMedia støtte for RTP over UDP. Samme funksjonalitet implementeres nå med liveMedia slik at liveMedia leser rammene fra nett før de sendes til dekoderen i libavcodec. MPlayer ble brukt som en modell for hvordan dette skulle implementeres.

MPlayer bruker hovedsakelig to filer for å implementere lesing fra nett med liveMedia: “libmpdemux/demux_rtp.cpp” for opprettelse av forbindelse mot tjener og “libmpdemux/demux_rtp_codec.cpp” for behandling av kodeker.

Ved kompilering av GEM må opsjonen “–with-live” brukes for at liveMedia-koden i filmFFMPEG skal benyttes. Koden for integrering av libavformat beholdes i bruk til alle typer URLer bortsett fra de som starter med "rtsp://", i disse tilfellene benyttes liveMedia.

liveMedia benytter seg av generelle klasser som utgjør utviklingsgrensesnittet og subklasser av disse som brukes internt, blant annet for å skille mellom forskjellige typer RTP-strømmer. Oppkobling mot tjeneren vil derfor innebære å opprette nødvendige objekter som selv tar seg av selve kommunikasjonen. Når forbindelsen er opprettet returnerer objektene er en rekke sesjonsobjekter som behandles for å finne kodeker, størrelse på video og andre egenskaper. Koden for oppkobling og mottak av pakker kan kopieres direkte fra MPlayer.

For å lese rammer fra nett brukes kode som er hentet fra funksjonen `getBuffer()` i MPlayer. I motsetning til libavformat, som av design er et blokkerende bibliotek, og leser pakker fra nett ved kall på den blokkerende funksjonen get_frame, er liveMedia ikke blokkerende. liveMedia har i stedet en tidsplanlegger (scheduler) som tar seg av alle hendelser, inkludert nye pakker og rammer fra nettet. Ideen til liveMedia er at tidsplanleggeren skal integreres i hovedløkken til applikasjonen. På den måten vil ikke liveMedia blokkere i det hele tatt. Dette ville forårsaket store strukturelle endring av PD/GEMs kode. MPlayer bruker en blokkeringsvariabel som gjør kallet blokkerende ved at tidsplanleggeren kjører en hendelsesløkke (ved å kalle `doEventLoop()`) som blokkerer til bildet er klart. Når bildet er klart brukes en tilbakekallsfunksjon til å informere tidsplanleggeren om at den kan slutte å blokkere slik at eksekveringen fortsetter. Figur 25 på neste side illustrerer hvordan tidsplanleggeren jobber.



Figur 25: Tidsplanleggeren i liveMedia har en egen hendelsesløkke som behandler hendelser som nye pakker. Tidsplanleggeren kaller på funksjoner i filmFFMPEG når hendelsene har inntruffet.

Figur 26 på neste side viser det endelige designet etter at liveMedia er integrert. Figur 27 på side 67 viser dataflyt mellom klient og tjener og hvilke komponenter som er involvert i en normal kjøring.

Som forventet ble det ikke observert bildefeil i kjøring med liveMedia, også bildeflyten var bedre. En dobling av oppdateringshastigheten til GEM hjalp ytterligere på bildeflyten.

Minimalisering av effekt ved blokkerende kall. liveMedia har støtte for å legge inn en tidsbegrensning (timeout) i blokkeringen, denne ble tatt i bruk for å fjerne noe av hakkingen forårsaket av blokkerende kall. Dersom tidsplanleggeren til liveMedia blokkerer for lenge, vil tidbegrensningen overskrides og kallet returnere uten nytt bilde.

En slik tidsbegrensning brukes i ved å kalle `scheduleDelayedTask()` i tidsplanleggeren, med blant annet antall millisekunder tidsbegrensning som parameter. Dersom tidsbegrensningen utløper, kalles en funksjonen hvor blokkeringen må oppheves, og et flagg settes slik at `getFrame()`-funksjonen kan returnere uten nytt bilde. Dersom flagget ikke er satt når blokkeringen oppheves, må `unscheduleDelayedTask()` kalles for å fjerne tidsbegrensningen fra tidsplanleggeren. Tidsbegrensningsparameteren må justeres for å få mest mulig jevn oppdatering, samtidig som den blokkerende funksjonen må ha tid nok til å lese rammene før den blir avbrutt. som nå gjenstår av hakking er Som resultat av denne justeringen ble det under kjøring observert at flyten på animasjonene i GEM ble

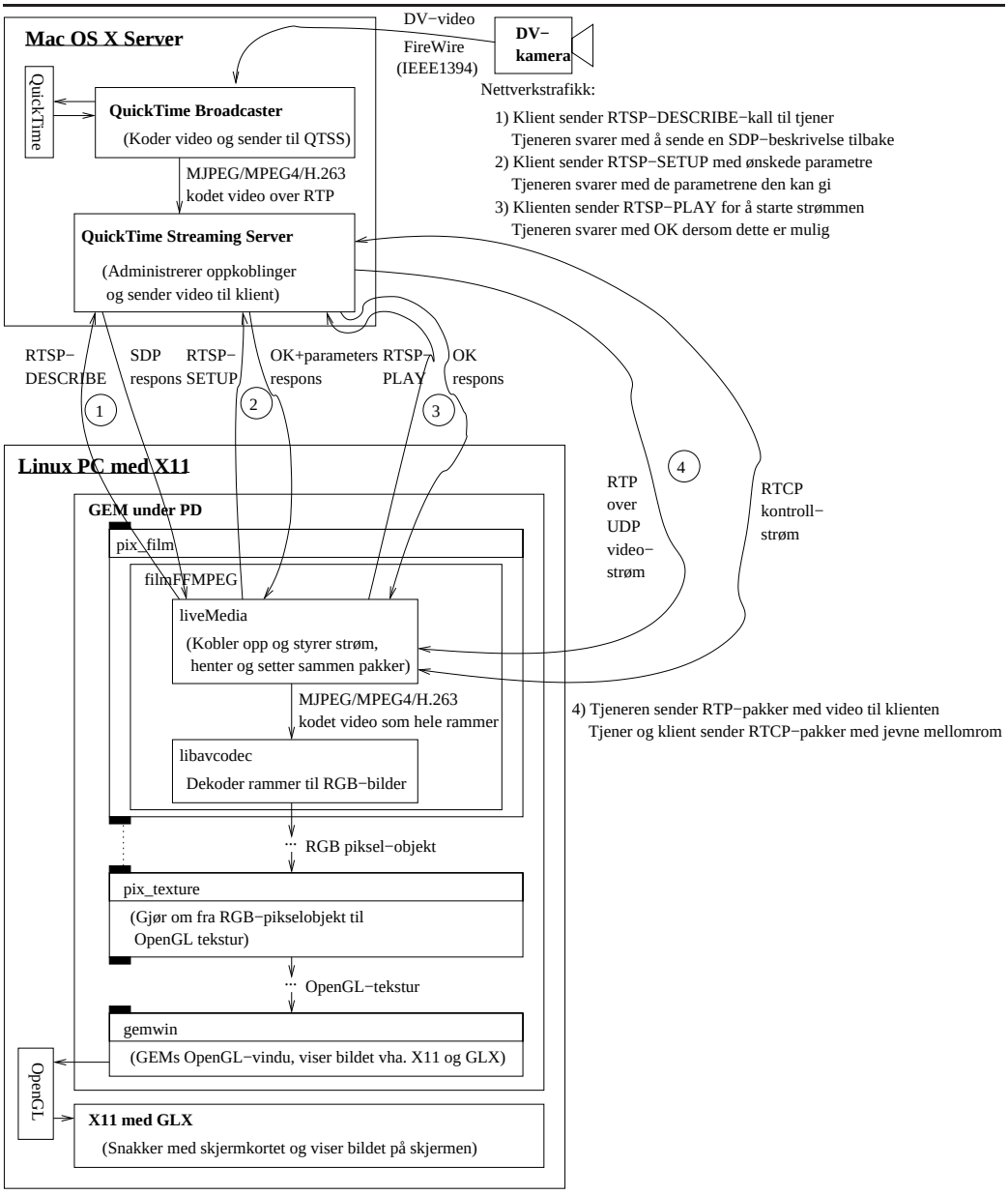
Figur 26: Design av GEMs filmFFMPEG i pix_film i den endelige versjonen. libavformat er også støttet, men er ikke tatt med i figuren.

jevne og uten hakking. Oppdateringen av bildene fra videostrømmen er fremdeles ujevn. Det viste seg senere at dette skylles ujevnheter i flyten på bildestrømmen fra tjeneren.

Bruk av MPEG4-kodek med liveMedia. For å implementere MPEG4 med liveMedia må parameteren “config” som tjeneren sender i svaret på RTSP-DESCRIBE legges inn i codec->extradata for at libavcodec skal klare å dekode pakkene. Linjen i svaret fra tjeneren ser slik ut:

```
a=fmtp:96 profile-level-id=1;config=000001B0F3000001B5
0EE040C0CF000001000000012000 8440FA285020F0A31F
```

4.1 Redigeringsmaskin



Figur 27: Virkemåte for hele oppsettet i den endelige versjonen. Nettverkstrafikk og dataflyt er illustrert.

Bruk av H.263-kodek med liveMedia. I tillegg till MJPEG og MPEG4 ble andre kodeker utprøvd fra QuickTime Broadcaster til MPlayer. Det viste seg at H.263 også fungerte. Støtte for denne kodeken ble implementert i GEM ved å velge H.263 som kodek i libavcodec. H.263 benytter veldefinerte størrelsesformatene på bildene, de vanligste er CIF og QCIF, som er heholdsvis 352x288 punkter og 176x144 punkter. Bildestørrelsen i QuickTime Broadcaster ble derfor satt til 352x288 punkter for å fylle hele CIF-bildet i H.263 videostrømmen.

4.1.4 Konklusjon for arbeid med redigeringsmaskin

Den endelige versjonen av GEM-objektet på redigeringsmaskinen følger designet i 3.4.5 på side 52. PD versjon 0.37 og Gem fra CVS, tilsvarende versjon 0.90 brukes som redigeringsprogramvare. filmFFMPEG er utvidet til å bruke liveMedia for overføring over RTSP/RTP over UDP, og er testet og funnet stabil over lang tids kjøring. filmFFMPEG kan med liveMedia jobbe med videostrømmer komprimert med MJPEG, MPEG4 og H.263.

4.2 Kildemaskin

FFserver og QuickTime Streaming Server er mulige kandidater for tjeneren. Disse ble undersøkt i 3.4 på side 48.

4.2.1 Oppsett av FFserver til å sende strømmer over RTSP/RTP

FFserver støtter RTSP/RTP, samt å sende sanntidsvideo fra kamera i tillegg til ferdigkodede filer. Gjennom arbeidet med å sette opp FFserver til å sende video fra kamera over RTSP/RTP ble det klart at RTSP/RTP-støtten til FFserver i FFMPEG versjon 0.4.8 var mangelfull. Videostrømmer fra kamera kunne bare sendes HTTP. I CVS-versjonen av FFMPEG er den generelle RTSP/RTP-støtten bedre, men FFserver kan i denne versjonen kun sende lagrede filer.

Kringkasting av live video i FFserver fungerer ved at FFserver leser en feed-fil som ffmpeg, FFMPEGs videokoder, skriver til. For å kunne bruke FFserver må koden som leser denne filen fikses. RTP-støtten begrenser seg dessuten til å kun støtte MPEG4 som videokodek, dersom andre kodeker skal testes må støtte for disse legges inn i RTP-koden. Siden det å rette problemene ville innebære store mengder arbeid, ble bruk av FFserver forkastet til fordel for QuickTime Broadcaster og QTSS.

Modell:	Power Mac G4 (AGP Graphics)
Prosessor:	PowerPC G4 (2.7) 450 MHz
Minne:	256 MB SDRAM
Harddisk:	WDC WD1000BB-75CHE0
Grafikk:	ATI Rage 128 Pro
Operativsystem:	Mac OS X Server 10.3.4

Tabell 8: Spesifikasjon av maskin til QuickTime Broadcaster.

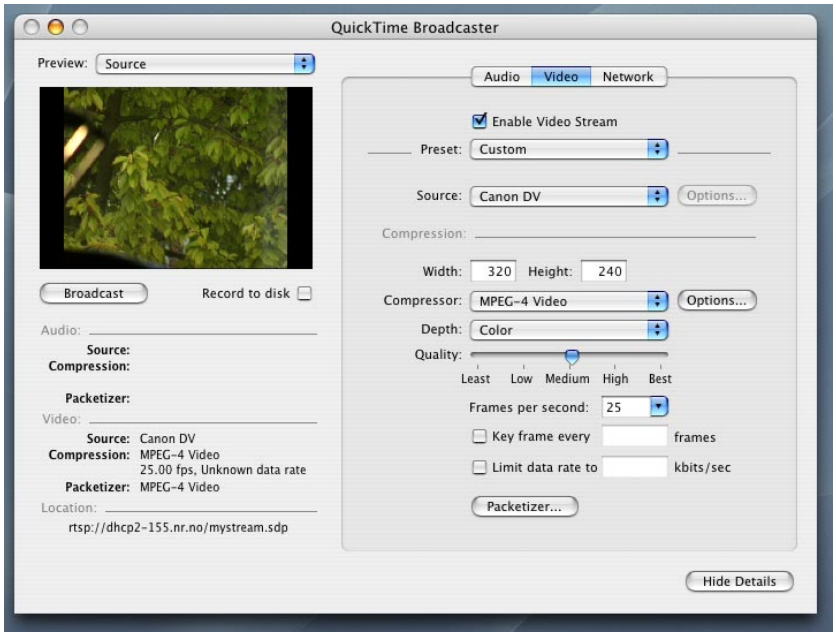
4.2.2 Oppsett av QuickTime Streaming Server og QuickTime Broadcaster for overføring av sanntidsstrømmer med RTSP/RTP

QuickTime Broadcaster og QuickTime Streaming Server (QTSS) følger med Mac OS X Server, som ble installert med versjon 10.3 på en Apple G4[10] og oppdatert til versjon 10.3.4 med oppdateringsprogrammet i operativsystemet for å få siste versjon av QuickTime med støtte for JPEG pakkeformat i RTP. Deretter ble kameraet koblet til FireWire-inngangen til maskinen og QuickTime Broadcaster konfigurert til å bruke DV-kameraet som videokilde med "Automatic Unicast (Announce)" som overføring under nettverksinnstillinger-fanen. MPEG-4 ble valgt som kodek og som RTP-pakkeformat. Denne tjeneren ble testet mot MPlayer, ffplay og xine på Linux.

"Automatic Unicast" betyr at en klient kan koble seg opp med RTSP og få en RTP-Unicast sesjon. Alternativene er manuell Unicast og Multicast. Manuell Unicast brukes dersom QuickTime Broadcaster skal kunne sende til kun en maskin, og konfigureres derfor manuelt med IP-adressen til denne maskinen. Dette brukes som oftest sammen med en annen tjener som bruker QuickTime Streaming Server som en proxy for å sende video videre til klientmaskinene. Multicast sender video ut til en Multicast adresse som flere klienter kan lytte til.

I videoinnstillingsbildet (se figur 28 på neste side) er "Source" er satt til "Canon DV", som er DV-kameraet koblet til FireWire porten. Under "Compression", kan brukeren velge størrelsen på bildet, hvilken kodek som skal brukes for komprimering, hvilken kvalitet som er ønsket og detaljinnstillinger for kodeken. I dette eksempelet MPEG4 valgt som kodek. "Packetizer"-knappen brukes til å velge hvilket RTP-pakkeformat som skal benyttes. I dette eksempelet er "MPEG-4 video" valgt, noe som tilsvarer MP4V-ES MIME-typen med pakkeformatet beskrevet i RFC 3016[55]. Bruk av QuickTime pakkeformatet, som er utviklet av Apple og fungerer best mellom Apples QuickTime-programvare, er standard valg. Kodekene som har alternative pakkeformat er H.261, H.263 (og VC-H.263),

[10] se tabell 8



Figur 28: Innstilling av kodeker i QuickTime Broadcaster. Viser valgene for kodek og overføringsformat.

MPEG4, MJPEG (Motion JPEG A, Photo-JPEG) og Sorenson Video (og SV 3)[11].

Konklusjon. MPEG4-strømmen (MPEG4 med MPEG4 pakkeformat) og MJPEG-strømmen ("Photo-JPEG" med JPEG pakkeformat) kunne spilles av i MPlayer. ffplay fra FFMPEG versjon 0.4.8 feilet. ffplay fra CVS-versjonen av FFMPEG klarte kun å spille MPEG4-strømmen.

4.2.3 Tuning av QuickTime Streaming Server for å redusere forsinkelsen

I det opprinnelige oppsettet var forsinkelsen fra kamera til fremvisning på over 10 sekunder. For sanntidsapplikasjoner er denne forsinkelsen for høy. I QuickTime Broadcaster kan forsinkelsen påvirkes ved å endre en opsjonen "buffer delay", ved å endre frekvensen av I-rammer i kodeker som benytter dette eller ved å slå av forhåndsvisningen. "buffer delay" endrer ikke forsinkelsen i tjeneren, men angir mange sekunder videobuffer klienten burde ha før den begynner å spille av strømmen. Endring av denne parameteren har derfor ikke effekt dersom klienten ikke støtter den.

QuickTime Broadcaster koder video fra kameraet og sender komprimert video videre til QTSS som tar seg av selve nettverkskommunikasjonen. QTSS konfigureres ved å endre innstillinger i "/Library/QuickTimeStreaming/Config/streamingserver.xml". Fra leverandøren er filen satt opp som beskrevet i brukerveiledningen[54], der forklares også hva et utvalg av opsjonene styrer.

Parametrene som måles er:

- **reflector_buffer_size_sec**
  Denne parameteren styrer bufferstørrelsen i QTSS for mottak av data fra QuickTime Broadcaster. Standardverdien for denne parameteren er 10 sekunder, noe som forklarer hvorfor forsinkelsen er på over 10 sekunder. Ved å sette denne til null går forsinkelsen ned til et snitt på 100ms. Denne parameteren er endret i eksperiment 1 på side 74.

- **overbuffer_rate**
  Denne parameteren styrer hvor mye fortere enn normal datarate tjeneren kan sende. Verdien multipliseres med dataraten for å regne ut hvor stor bufferen i klienten skal være. Fra leverandøren er denne verdien satt til 2.0. I eksperiment 2 på side 76 endres parameteren ved å gjøre denne bufferen mindre for å undersøke om forsinkelsen endres.

[11] Proprietær kodek som tidligere har vært mye brukt i QuickTime-filer

- **send_interval**
  Denne parameteren styrer minimumstiden tjeneren venter mellom hver gang den sender data til klienten. Fabrikkinnstillingen er 50 millisekunder. I eksperiment 3 på side 76 blir ventetiden redusert for å undersøke om dette har positiv effekt på forsinkelsen.

- **reflector_bucket_offset_delay_msc**
  Denne parameteren er ikke dokumentert i [54], men den er tatt med da navnet tyder på at den styrer noe som har med forsinkelse på strømmen mellom QuickTime Broadcaster og QTSS å gjøre. I eksperiment 4 på side 76 er denne parameteren endret.

- **always_thin_delay**
  QTSS har en innebygget algoritme, kalt "thinning", som skal hindre metning i nettet ved å analysere RTCP-pakker og finne forsinkelsen på RTP-pakker, for så å justerer båndbreddebruken ut fra resultatene.

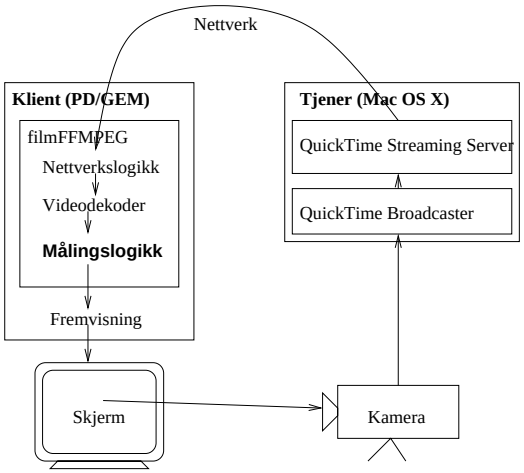
  Denne parameteren styrer hvor mye forsinket en pakke må være før "thinning"-algoritmen alltid reduserer båndbredden. Standardverdien er 750 millisekunder.

- **start_thicking_delay**
  Denne parameteren styrer hvor forsinket en pakke kan være før algoritmen begynner å bruke mer båndbredde. Standardverdien er 250 millisekunder. I eksperiment 6 på side 77 er always_thin_delay og start_thicking_delay satt til verdier som gjør at algoritmen endrer båndbreddebruken tidligere.

- **thin_all_the_way_delay**
  Denne parameteren styrer hvor mye forsinket en pakke må være før "thinning"-algoritmen reduserer båndbreddebruken. maksimalt i forhold til valgt kvalitetsnivå. Standardverdien er 1500 millisekunder. I eksperiment 7 på side 79 er denne og de to foregående parametrene satt til null for å hindre at algoritmen brukes. Båndbreddebruken vil da være konstant på lavest mulig nivå i med valgt kvalitet.

- **Forhåndsvisning i QuickTime broadcaster**
  QuickTime Broadcaster har forhåndsvisning av video (se figur 28 på side 70). Denne forhåndsvisningen oppdateres omtrent en gang i sekundet. I eksperiment 5 på side 77 måles forskjell i forsinkelse når forhåndsvisningen slås av.

- **Dæmoner i tjeneren**
  I UNIX systemer kjøres en rekke tjenester i bakgrunnen som kalles



Figur 29: Målingslogikken tar tiden, sender ut hvitt bilde langs pilene som registreres når det kommer tilbake og tidsdifferansen registreres.

dæmoner. Disse kan til tider bruke mye ressurser. I eksperiment 8 på side 80 er de mest ressurskrevende dæmonene i Mac OS X server fjernet.

Under eksperimentene ble QuickTime Broadcaster konfigurert til å sende en MJPEG-strøm (Photo-JPEG) med 352x288 (CIF) oppløsning og lav kvalitet. Grunnen til å velge lav kvalitet var for å hindre at prosessorforbruk på komprimering skulle få for stor innflytelse på målingene. Maskinene ble koblet sammen direkte med en krysset TP-kabel for å hindre påvirkning fra utenforliggende maskiner og utstyr som switcher og HUB-er i nettverket.

Måling av forsinkelse. RTCP gir kun informasjon er pakketap og forsinkelses-varians på nettverksnivå. Forsinkelsen fra kamera til maskin, koding av video, utsending, dekoding osv. blir ikke med i RTCP.

For å måle den totale forsinkelsen fra kameraet registrerer et bilde til brukeren ser bildet på skjermen, ble det implementert kode i GEM-objektet som viser et hvit bilde annenhvert sekund. Ved å rette kameraet mot output-vinduet fra GEM, slik at det registrer det hvite blinket og sender bildet tilbake til GEM-objektet, kan den totale forsinkelsen mellom vising og mottak måles.

Konkret er dette gjort ved at tiden måles etter at bildet er dekodet som illustrert i figur 29. Dersom sekundmåleren er delelig på to, og mikrosekundmåleren er mindre enn 20000 mikrosekunder, genereres et hvit bilde som erstatter det dekodede bildet. Tidssjekken er gjort for å hindre at en ny puls sendes før forrige

Gjennomsnittlig forsinkelse:	100.14 ms.
Standardavvik:	69.66 ms.
Høyeste forsinkelse:	520.21 ms.
Laveste forsinkelse:	25.26 ms.

Tabell 9: Statistikk for forsinkelse etter at reflector_buffer_size_sec var endret.

er målt. Målinger viste at forsinkelsen alltid var under ett sekund, dersom det hvite bildet ikke gjenkjennes i løpet av to sekunder regnes det som tapt. Gjenkjenning av det hvite bildet er implementert slik at to av punktene i bildet sjekkes, dersom de er hvite måles og registreres tiden fra bildet ble generert. Det blir også registrert at tiden er målt slik at den ikke blir målt dersom neste bilde også er hvit, noe som kan forekomme på grunn av etterslep i skjerm og kamera.

For å hindre ekko som resultat av at hvite bilder som har blitt gjenkjent vises og registreres av kameraet som et nytt hvitt blink, genereres sorte bilder som erstatter dekodede bilder mellom blinkene. Det hindrer også at lyse punkter på skjermen forsterkes, som ved audio feedback, og registreres som hvite ved feiltakelse. Totalt fører denne testen, inkludert generering av hvite og sorte bilder, til en ekstra forsinkelse på 2.7 ms. i snitt.

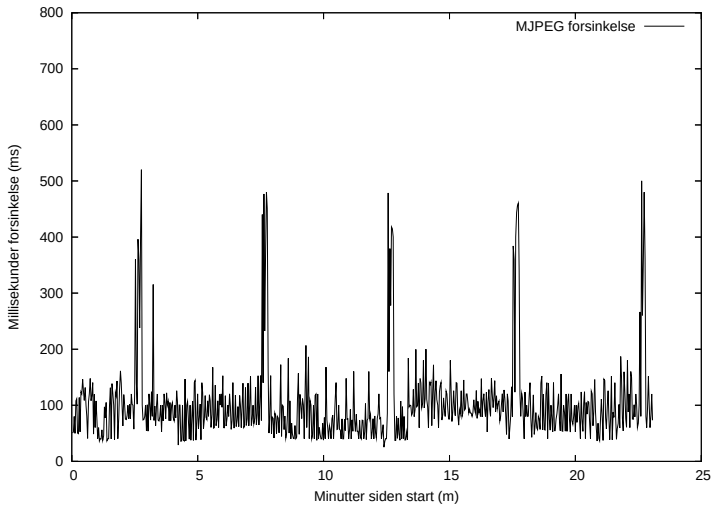
Eksperiment 1 (reflector_buffer_size_sec=0.) Denne opsjonen var hovedårsaken til den lange forsinkelsen. Standardverdien innførte en forsinkelse på 10 sekunder som ble borte ved å sette denne til 0.

Denne bufferen brukes når Broadcaster prøver å re-sende pakker som blir borte mellom seg og QTSS, det trengs derfor en forsinkelse i QTSS slik at den kan hente seg inn igjen. Dette hindrer klientene i å merke pakketapet. Siden pakketap mellom Broadcaster og QTSS er ikke så interessant for denne applikasjonen, hvor begge kjører på samme maskin, ble variabelen satt til 0, for å fjerne hele bufferet. Resultatet av målingene vises i tabell 9.

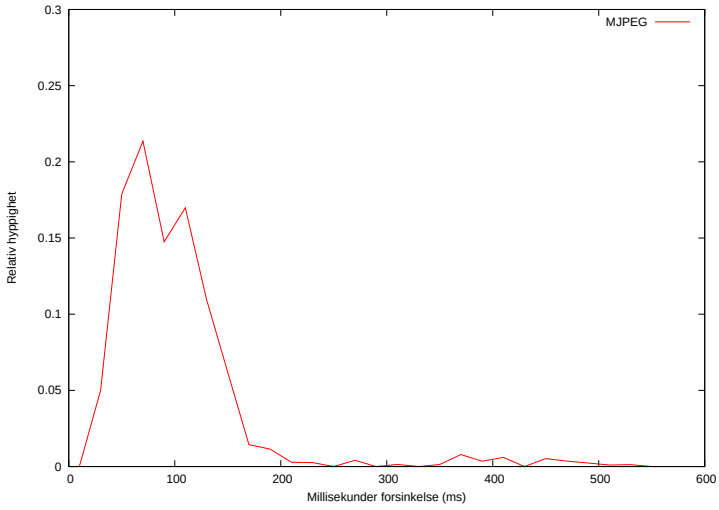
Den gjennomsnittlige forsinkelsen ligger på 100 millisekunder, forbedringen er på 10 sekunder som forventet. Standardavviket er på 70 millisekunder, som betyr at forsinkelsen varierer mye. Dette kan også observeres visuelt når applikasjonen brukes i praksis.

Figur 30 på neste side viser hvordan forsinkelsen varierer over tid. Toppene hvert femte minutt skyldes økt belastning på tjeneren når en dæmon[12] jobber i bakgrunnen. Eksperiment 8 på side 80 viser hva som skjer dersom disse fjernes. Bortsett fra disse toppene varierer forsinkelsen fra 50 til 200 millisekunder.

[12] se ordforklaringer bakerst



Figur 30: Forsinkelse for MJPEG etter at reflector_buffer_size_sec ble satt til 0 sekunder. Legg merke til toppene hvert femte sekund.



Figur 31: Forsinkelsehistogram for MJPEG etter at reflector_buffer_size_sec ble satt til 0 sekunder.

Gjennomsnittlig forsinkelse:	101.93 ms.
Standardavvik:	64.85 ms.
Høyeste forsinkelse:	580.27 ms.
Laveste forsinkelse:	26.37 ms.

Tabell 10: Statistikk for forsinkelse etter at overbuffer_rate var endret.

Gjennomsnittlig forsinkelse:	109.34 ms.
Standardavvik:	62.05 ms.
Høyeste forsinkelse:	610.33 ms.
Laveste forsinkelse:	25.53 ms.

Tabell 11: Statistikk for forsinkelse etter at send_interval var endret.

Figur 31 på forrige side viser relativ hyppighet av forsinkelse innenfor ulike intervaller. Figuren viser at forsinkelsen vanligvis er lav, men at målinger med høy forsinkelse drar opp snittet. Målingene ved 400-500 millisekunder tilsvarer toppene i figur 30.

Eksperiment 2 (overbuffer_rate=1.0.) Verdien ble endret til 1.0, som betyr en halvering av det opprinnelige bufferet. Dette bufferet benyttes når data mottas med en rate som er høyere enn normal senderate, opsjonen styrer hvor stor bufferet skal være. Resultatet er i tabell 10, innstillingen i eksperiment 1 er beholdt.

Tabellen viser at forsinkelsen er på 102 ms. og standardavviket på 65 ms. Endringen ved å sette parameteret til “1.0” er ubetydelig. Forsinkelsen over tid er som før. Endring av denne opsjonen vil ha større betydning ved overføring av lagret video, da bufferen fylles når data sendes med en rate høyere enn den opprinnelige raten. Siden overføring av sanntidsvideo ikke kan gjøres fortere enn den blir generert, vil bufferet aldri fylles opp.

Eksperiment 3 (send_interval=0.) Standardverdi på denne parameteren er 50 millisekunder, den ble endret til 0. Parameteren styrer minimum ventetid mellom hver gang tjeneren sender data. Resultatet vises i tabell 11, innstillingen i eksperiment 1 er beholdt. Tabellen viser at hyppigere sending fører til økt forsinkelse. Forsinkelsen økte med 9 ms, standardavviket med 3 ms.

Eksperiment 4 (reflector_bucket_offset_delay_msc=0.) Denne parameteren har 73 millisekunder som standardverdien, den ble endret til 0. Parameteren er relatert til reflector_buffer_size_sec. Resultatet av denne testen er i tabell 12 på neste side, innstillingene er ellers som i eksperiment 3.

Denne innstillingen påvirker hverken standardavvik eller forsinkelse.

Gjennomsnittlig forsinkelse:	109.25 ms.
Standardavvik:	62.80 ms.
Høyeste forsinkelse:	574.99 ms.
Laveste forsinkelse:	29.52 ms.

Tabell 12: Statistikk for forsinkelse etter at send_interval og reflector_bucket_offset_delay_msc var endret.

Gjennomsnittlig forsinkelse:	89.08 ms.
Standardavvik:	44.06 ms.
Høyeste forsinkelse:	480.69 ms.
Laveste forsinkelse:	30.13 ms.

Tabell 13: Statistikk for forsinkelse etter at forhåndsvisning ble slått av.

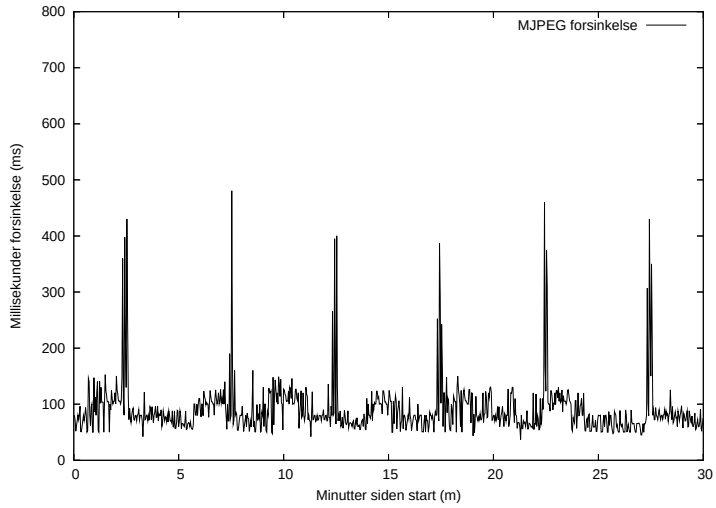
Eksperiment 5 (Forhåndsvisning i QuickTime Broadcaster.) I dette eksperimentet ble forhåndsvisningen slått av. Resultatet er i tabell 13, innstillingene i QTSS er som i eksperiment 1.

Forsinkelsen er forbedret med 11 millisekunder. Standardavviket til forsinkelsen er forbedret med 25 millisekunder. Dette gjør at forsinkelsen varierer mindre og gjør at bildeflyten virker mindre hakkete for brukeren.

Figur 32 på neste side viser forsinkelsen over tid. Sammenligning med figur 30 viser at variasjonen ikke er like hyppig, grafen er jevnere.

Forhåndsvisningen kan brukes for å verifisere at QuickTime Broadcaster kan lese videosignalet fra kameraet, men bør slås av etter at dette er verifisert. For å bestemme kamerautsnitt, justere fokus, zoom og andre kamerainnstillinger er forhåndsvisningen på kameraet er bedre.

Eksperiment 6 (always_thin_delay=100, start_thicking_delay=50.) Disse opsjonene har standardverdier på henholdsvis 750 og 250 ms. De ble endret til 100 og 50 ms. Verdiene styrer hvor stor forsinkelsen på pakkene kan være før tjeneren reduserer eller øker båndbreddebruken. Standardverdiene er tilpasset bruk over Internett, hvor en forsinkelse på 750 ms. vil bety at det er metning slik at båndbreddebruken må reduseres. Når forsinkelsen er på under 50 ms. er metningen over og båndbreddebruken kan økes for å forbedre bildekvaliteten. I dette oppsettet er ikke metning i nettet et problem, det største problemet ser ut til å være overbelastning av tjeneren. Når båndbreddebruken reduseres vil også kvalitetskravet reduseres som fører til redusert belastning på CPUen. Opsjonene er derfor justert slik at båndbreddebruken reduseres tidligere. Resultatet av denne testen er i tabell 14 på neste side, innstillingene er ellers som i eksperiment 5.



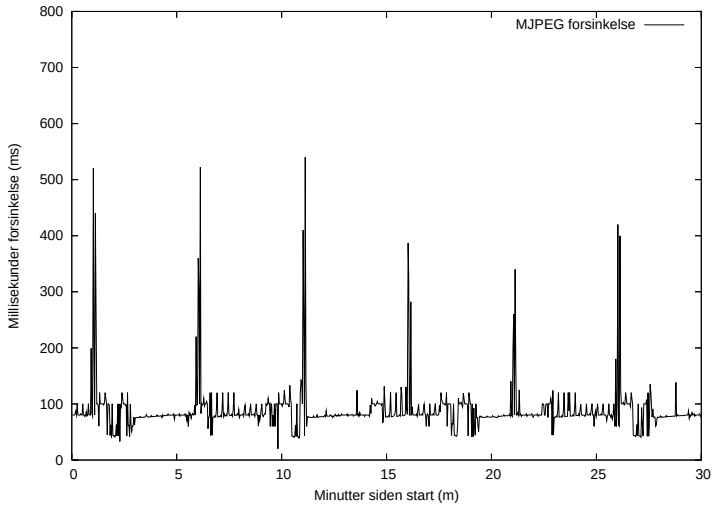
Figur 32: Forsinkelse for MJPEG etter at forhåndsvisning var slått av.

Gjennomsnittlig forsinkelse:	91.81 ms.
Standardavvik:	41.87 ms.
Høyeste forsinkelse:	510.21 ms.
Laveste forsinkelse:	46.24 ms.

Tabell 14: Statistikk for forsinkelse etter at always_thin_delay er satt til 100 og start_thicking_delay er satt til 50.

Gjennomsnittlig forsinkelse:	88.96 ms.
Standardavvik:	37.98 ms.
Høyeste forsinkelse:	540.26 ms.
Laveste forsinkelse:	19.31 ms.

Tabell 15: Statistikk for forsinkelse etter at thin_all_the_way_delay, always_thin_delay og start_thicking_delay er satt til 0.



Figur 33: Forsinkelse for MJPEG etter at thin_all_the_way_delay, always_thin_delay og start_thicking_delay er satt til 0.

Tabellen viser at forsinkelsen har økt med nesten 3 millisekunder. Standardavviket er blitt 2 millisekunder lavere. Oppførselen over er uendret.

I eksperiment 7 justeres opsjonene ytterligere.

Eksperiment 7 (thin_all_the_way_delay=0) Standardverdien på denne opsjonen er 1500 millisekunder, den ble endret til 0. Det samme ble opsjonene always_thin_delay og start_thicking_delay. thin_all_the_way_delay styrer hvor forsinket en pakke må være før båndbreddeforbruket blir redusert maksimalt i forhold til "thinning"-algoritmen. Når alle disse opsjonene er satt til null vil tjeneren alltid sende med lavest mulig båndbredde ut fra "thinning"-algoritmen. Resultatet av denne kjøringen er i tabell 15, innstillingene er ellers som i eksperiment 5.

Forsinkelsen er som i eksperiment 5. Standardavviket har blitt redusert med 4 ms. i forhold til eksperiment 6, og totalt 6 ms. i forhold til eksperiment 5.

Ved å se på forsinkelsen over tid (se figur 33), er det tydelig å se hvordan

Gjennomsnittlig forsinkelse:	87.23 ms.
Standardavvik:	35.88 ms.
Høyeste forsinkelse:	327.71 ms.
Laveste forsinkelse:	12.69 ms.

Tabell 16: Statistikk for forsinkelse etter at lookupd og netinfod ble stopet.

variasjonen ikke lenger er like hyppig, men jevnere.

Eksperiment 8 (Fjerning av ressurskrevende dæmoner i tjeneren) Alle de tidligere grafene viser en topp i forsinkelsen hvert femte minutt. Tendensen er veldig tydelig når intervaller på 30 minutter undersøkes.

Måling av belastning på klienten viste at belastningen var rundt 20% når testene kjørte. Det er derfor ikke noe problem for klienten å ta mot og behandle data fort nok. Tjeneren har en belastning på 70-80% ved sending av MJPEG. Dersom en annen prosess aktivt konkurrerer om ressurser, vil ikke QuickTime Broadcaster ikke får behandlet video fort nok, resultatet blir økt forsinkelse. "Activity Monitor" i Mac OS X ble brukt for å overvåke aktiviteten på maskinen, og viste at prosessen "lookupd" førte til at aktiviteten økte hvert femte minutt.

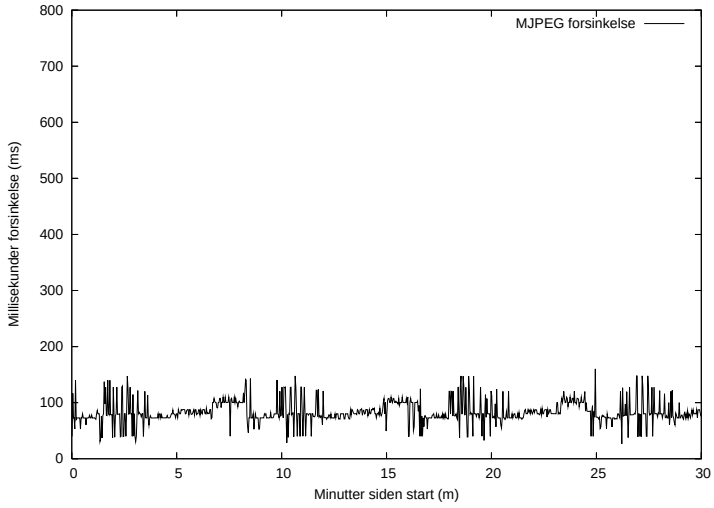
"lookupd" benyttes av Mac OS X som en hurtigbuffer ved oppslag av verdier som brukernavn, grupper, e-post adresser og annet. Den startes automatisk av en annen dæmon, "netinfod", ved behov. I dette eksperimentet ble "netinfod" og "lookupd" stoppet og hindret i å starte igjen. Resultatet er i tabell 16, innstillingene er ellers som i eksperiment 7.

Den største forskjellen er høyeste forsinkelse, som er redusert med over 200 ms. Det vil si at "worst case" forsinkelsen er kraftig forbedret.

Figur 34 på neste side viser tydelig at forsinkelsen varierer mindre. Toppene er nå borte, det som nå gjenstår er små svingninger som virker mindre systematiske.

Etter at "netinfod" og "lookupd" var slått av, var det ikke lenger var mulig å logge inn på maskinen etter en omstart og operativsystemet måtte re-installeres. Et alternativ til å stoppe prosessene fulstendig er å redusere prioriteten deres og samtidig øke prioriteten til QuickTime Broadcaster. Dette fører til noe mindre topper, men de er fremdeles veldig tydelige.

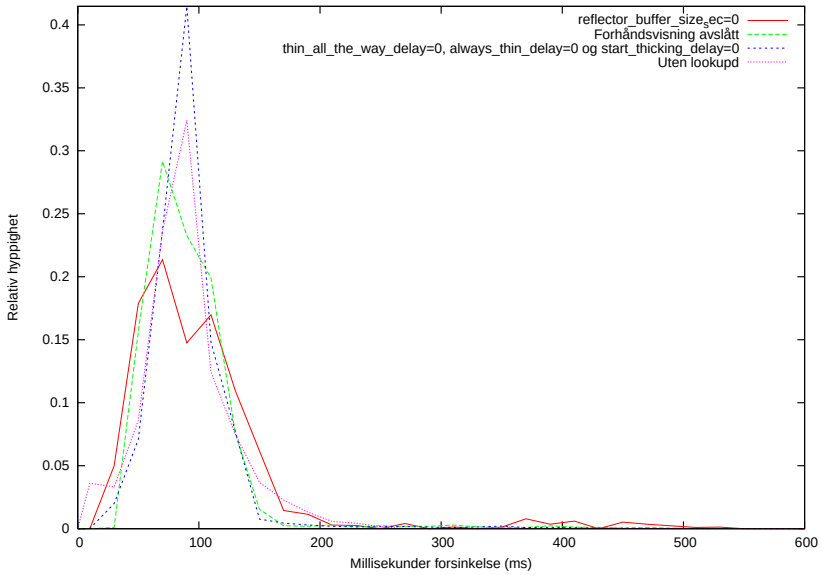
Endringene i dette eksperimentet kan ikke anbefales for bruk i en applikasjon. For at disse toppene, hvis årsak nå er kjent, ikke skal forstyrre de videre målingene blir endringene likevel beholdt for ytterligere målinger. En bedre løsning vil være å ha en kraftigere tjenermaskin hvor utslagene ikke vil være like store.



Figur 34: Forsinkelse for MJPEG etter at thin_all_the_way_delay, always_thin_delay og start_thicking_delay er satt til 0.

Parameter	**Redusert forsinkelse**	**Redusert standardavvik**
reflector_buffer_size_sec=0	10 s.	Ikke målt
Forhåndsvisning slått av	11 ms.	25 ms.
thin_all_the_way_delay=0 always_thin_delay=0 start_thicking_delay=0	0 ms.	6 ms.

Tabell 17: Oppsummering av anbefalte parameterinnstillinger i tjeneren



Figur 35: Sammenligning av forsinkelse med ulike parametre.

Oppsummering. Anbefalte endringer oppsummeres i tabell 17 på forrige side.

Figur 35 viser histogrammene til de målingene hvor konklusjonen ble at endringen skulle beholdes. Histogrammet fra målingen hvor “lookupd” og “netinfod” var fjernet er også tatt med, selv om det ikke ble konkludert med at disse endringene skulle beholdes.

Figuren viser at målingene har en topp som ligger under av 100 ms., grunnen til at den gjennomsnittlige forsinkelsen har gått ned er derfor at det er færre målinger med veldig høy forsinkelse, eller flere med lav forsinkelse. Hovedvekten i grafene går mot høyre, dette er særlig tydelig i målingen uten “lookupd”. Figuren viser at kurvene blir smalere etter endringene, som resultat av at variasjonen av forsinkelsen blir mindre. Unntaket her er målingen uten “lookupd”, hvor kurven er lavere siden det er flere målinger med lav forsinkelse og forsinkelse rett til høyre for snittet, men mangelen på forsinkelse over 350 ms. gjør at snittet og standardavviket er bedre.

5 Sammenligning av MJPEG, MPEG4 og H.263

I dette kapittelet sammenlignes ytelsen til MJPEG-, MPEG4- og H.263-kodekene i QuickTime ved streaming fra QTSS til GEM. Kriteriene som blir brukt for sammenligning er subjektiv kvalitet, hvor kvalitet video og stillbilde vurderes, målt forsinkelse og forsinkelsevarians, prosessorbruk og båndbreddekrav.

Parametrene på tjeneren er slik de var i eksperiment 8 på side 80. Den eneste parameteren som justeres underveis er valg av kodek, som velges på nytt for hver test. Kvaliteten til kodekene er satt til “low”.

I alle testene med unntak av forsinkelsemålingene er det benyttet tre sekvenser hvor bildemotivet er forskjellig. Sekvens 1 er et utendørsmotiv av blader i et tre, med skarpe farger og bevegelse i bladene. Sekvens 2 er et innendørsmotiv med et kontor hvor belysning og farger tilsvarer det som brukes i en vanlig videokonferanse. Sekvens 3 er samme motiv som sekvens 2, men med mye bevegelse i form av vinking. Klippene ble lagret på DV-tape slik at variasjon i videomateriale ikke skulle ha innflytelse på resultatene.

5.1 Subjektiv bildekvalitet

Flere artikler er skrevet hvor subjektiv bildekvalitet er målt. I [56] drøftes noen av disse artiklene, og hvilke fordeler måling av subjektiv bildekvalitet har over måling av objektiv bildekvalitet som f.eks. med PSNR. Det er utført en subjektiv måling for å finne ut hvilken kodek en bruker syntes gir best resultat.

Målingene er utført etter “Single Stimulus Numerical Categorical Scale” (SSNCS) metoden i ITU-R BT.500-11 [57], som ble valgt fordi den ikke krever to identiske skjermer hvor en skjerm har referansevideo og den andre videoklippet hvor kvaliteten skal måles. De tre videoklippene ble presentert for 7 testkandidater, først ukomprimert for å bli kjent med innholdet og så komprimert med de forskjellige kodekene. Testkandidatene ga karakterer for videokvaliteten på en skala fra 0 til 10, hvor 10 var best. Rekkefølgen på kodekene var tilfeldig og ikke lik for de forskjellige kandidatene, og kandidatene fikk ikke vite hvilken kodek som var i bruk.

I tillegg ble ett stillbilde fra sekvens 1 og ett fra sekvens 2 presentert. Testkandidatene ble bedt om å vurdere hvilket bilde som så best ut, og hvilket som så dårligst ut. Formålet med denne testen var å ha en referanse hvor bildeflyt ikke har innflytelse, resultatene blir brukt til å støtte opp om, eller kritisere resultatene fra vurderingen av videokvalitet.

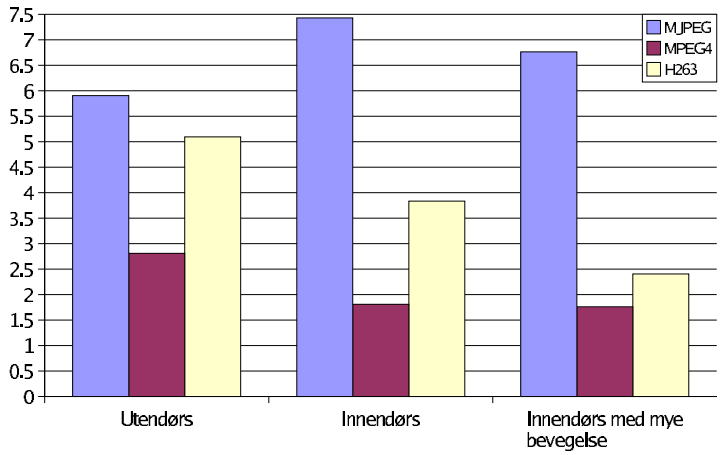
5.1 Subjektiv bildekvalitet

Vindusstørrelsen til presentasjonene var 352x288 punkter, noe som er samme som videostrømmen er fra tjeneren. Objektet videostrømmen projiseres på er også skalert for å ta hensyn til bredde/høyde-forholdet, slik at videosekvensen nøyaktig dekker vinduet uten at pikslene er skalert.

5.1.1 Kvalitet på videosekvenser med forskjellige kodeker

Karakterene til kandidatene ble normalisert slik at alle bruker karakterer fra 0 til 10, med formel 5 fra [57]. Gjennomsnittet for alle målingene er vist i figur 36.

Figur 36: Subjektiv kvalitetsmåling av kodeker.



Med alle sekvensene er MJPEG foretrukket ovenfor de andre kodekene. Kun i sekvens 1 er H.263 i nærheten av MJPEG, med en differanse på 0,8. H.263 er nest best i alle sekvensene, MPEG4 dårligst.

Ved spørsmål om hvilke kriterier kandidatene la til grunn for karaktergivingen, hadde kandidatene delte oppfatninger. Halvparten mente flyten på videosekvensen, hvor mye den hakker, var viktigst. Resten mente hakkingen ikke var forstyrrende, men at hvor synlighetsgraden av komprimeringsblokkene i bildet var utslagsgivende.

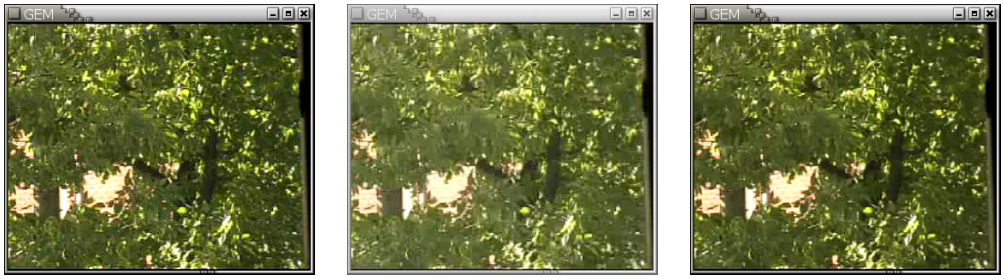
I 5.3 på side 87 vises det at hovedårsaken til hakkingen er for liten CPU i tjenermaskinen. Det er derfor interessant å også se på kvaliteten av enkeltbildene. Med en raskere tjener ville hakkingen vært mindre og kvaliteten vil være mer styrt av kvaliteten på enkeltbildene.

5.1.2 Kvalitet på stillbilde

Skjermbilder hvor bildene er komprimert med de forskjellige kodekene ble presentert for kandidatene. Rekkefølgen på kodekene var tilfeldig i begge sekvensene, men lik for alle kandidatene. Bildene var merket med a, b og c, fra venstre mot høyre.

Utendørs natur. Et bilde fra sekvens 1 komprimert med de tre kodekene sammenlignes, disse er vist i figur 37.

Figur 37: Skjermbilder av video tatt av et tre utenfor vinduet. Bladene har skarpe farger og beveger seg konstant i vinden.



(a) MJPEG (b) MPEG4 (c) H.263

Alle testkandidatene svarte at bildet kodet med MPEG4 hadde dårligst kvalitet. Fire av kandidatene mente MJPEG var best, tre mente H.263 var best. Dette stemmer overens med resultatene fra testen over videokvalitet og støtter opp om konklusjonen der.

Innendørs med lite bevegelse. Bildet fra sekvens 2 er vist i figur 38 på neste side komprimert med de tre kodekene.

Alle kandidatene mente bildet kodet med MPEG4 hadde dårligst kvalitet. Fem mente MJPEG hadde best kvalitet, to mente H.263. Dette stemmer også overens med kvalitetsmålingen for video, også der var MJPEG tilsvarende bedre enn H.263.

Hadde H.263 blitt vurdert som bedre i denne testen ville det tydet på at hakkingen i videostrømmen var mest utslagsgivende for vurderingene i kvalitetsmålingen for video. At tallene er omtrent de samme her støtter opp om at kvaliteten på MJPEG-kodeken er bedre.

Figur 38: Skjermbilder av video tatt av en person inne på et kontor. Farger og lys er som i et konferanseoppsett med lite bevegelse.

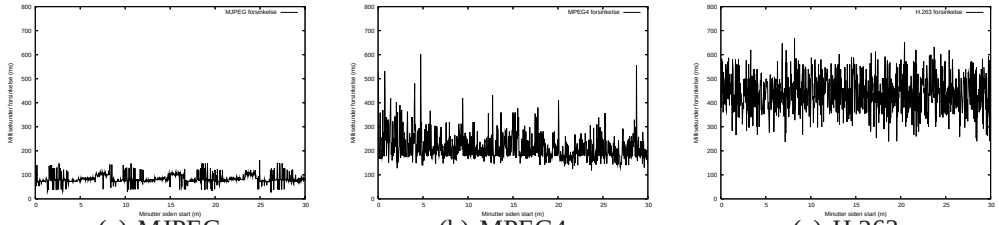
(a) MJPEG (b) MPEG4 (c) H.263

5.2 Målt forsinkelse og forsinkelsesvarians

Forsinkelsen av videostrømmen ble målt for de forskjellige kodekene. Målingen ble gjort på samme måte som i forrige kapittel. Styrken med disse målingene er at de gir et nøyaktig bilde av hvor lang tid det tar fra kameraet registrerer noe til det er synlig på mottakerskjermen, svakheten er at videosekvensen som blir sendt ikke er en videosekvens med naturlige bilder. Det er meget mulig at forsinkelsestallene ville vært annerledes dersom video av en annen art hadde vært brukt i stedet for kun sorte og hvite bilder, som er meget enkle å komprimere.

Målingene er gjort med samme oppsett som i figur 34 på side 81, altså med "lookupd" og "netinfod" fjernet. Kjøringene ble kjørt i 12 timer eller mer for å få mange målinger til histogrammene.

Figur 39: Forsinkelse for MJPEG, MPEG4 og H.263 over 30 minutter



(a) MJPEG (b) MPEG4 (c) H.263

Figur 39 viser forsinkelsen over tid de første 30 minuttene. Grafen til MJPEG er

kjent fra figur 34, og viser en forholdsvis jevn forsinkelse med noen ujevnheter innimellom. MPEG4 har mer ujevn forsinkelse. Her går forsinkelsen sjelden under 200 millisekunder, men holder seg rundt der og går opp med ujevne mellomrom. H.263 har enda høyere forsinkelse, snittet ser ut til å ligge på rundt 450 millisekunder, og forsinkelsen varierer hele tiden.

Tabell 18: Statistikk for forsinkelse av MJPEG, MPEG4 og H.263.

	MJPEG	**MPEG4**	**H.263**
Gjennomsnittlig forsinkelse:	87.23 ms.	195.70 ms.	401.78 ms.
Standardavvik:	35.88 ms.	51.90 ms.	87.03 ms.
Høyeste forsinkelse:	327.71 ms.	720.20 ms.	820.01 ms.
Laveste forsinkelse:	12.69 ms.	79.10 ms.	179.85 ms.

Tabell 18 viser at den gjennomsnittlige forsinkelsen var 87 millisekunder for MJPEG, 196 millisekunder for MPEG4 og 402 millisekunder for H.263. Dette viser at forsinkelsen er over dobbelt så høy for MPEG4 som for MJPEG og H.263 har dobbelt så høy forsinkelse som MPEG4 igjen, altså fire ganger forsinkelsen til MJPEG, som også kunne bekreftes i de subjektive målingene i kapittel 5.1.1 på side 84.

Det empiriske standardavviket fra gjennomsnittet er også forskjellig for MJPEG, MPEG4 og H.263. Det er henholdvis 36, 52 og 87 millisekunder. I figur 39 er avviket synlig som svingninger.

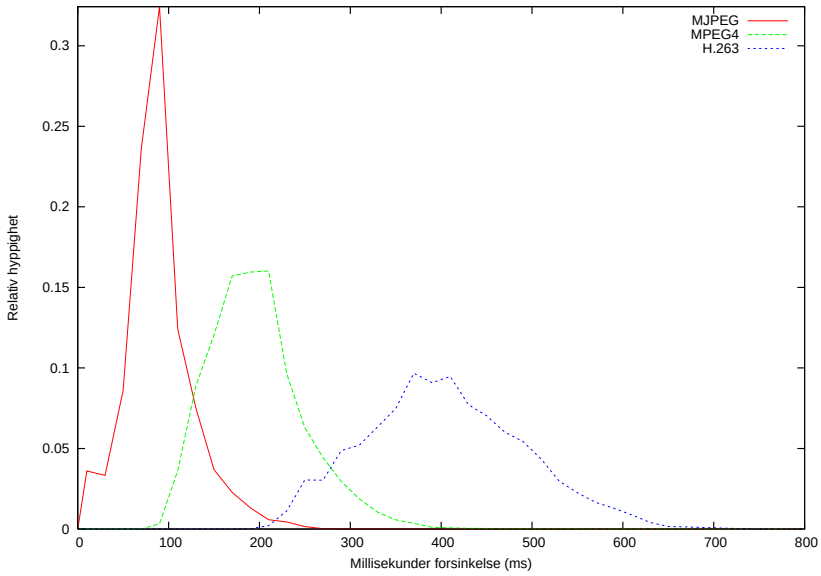
Figur 40 på neste side viser en sammenligning av histogrammene til de tre kjøringene. Her har MJPEG en høy og smal kurve, som resultat av lavt standardavvik, sentrert litt under 100 millisekunder. MJPEG er sentrert rundt 200 millisekunder, og har en noe bredere kurve. H.263 har en veldig bred kurve og er sentrert rundt 400 millisekunder.

5.3 Prosessorbruk i QuickTime Broadcaster

Måling av prosessorbruken kan forklare noe av hakkingen og forsinkelsen. Dersom prosessoren ikke klarer å kode video fort nok til å sende 25 bilder i sekundet vil videofremvisningen være hakkete. Dersom prosessoren nesten ikke er belastet bør video kunne sendes ut veldig fort slik at forsinkelsen blir mindre.

Prosessorbruk er målt ved å se på prosessorbruken rapportert av QuickTime Broadcaster og notere høyeste og laveste verdi. Broadcaster rapporterer også bildefrekvensen den sender ut. Denne er også registrert dersom den er mindre enn 25 bilder i sekundet.

Figur 40: Sammenligning av forsinkelse til MJPEG, MPEG4 og H.263.



Utendørs natur. Dette er den mest prosessorintensive kjøringen. Her bruker MJPEG fra 68-86% prosessorkraft. MPEG4 bruker 90-100%, med bildefrekvens som noen ganger er under 25 og H.263 bruker 96-100% med rundt 17 bilder i sekundet.

Innendørs med lite bevegelse. Dette er den minst prosessorintensive kjøringen. MJPEG bruker her 60-82% prosessorkraft, MPEG4 bruker 90-98%, denne gangen med 25 bilder i sekundet. H.263 oppnår rundt 17 bilder i sekundet med 96-100% prosessorkraft.

Innendørs med mye bevegelse. Med mer bevegelse i bildet stiger prosessorbruken til MJPEG til 67-86%, MPEG4 bruker 90-100% og H.263 bruker fremdeles 96-100% med 20 bilder i sekundet.

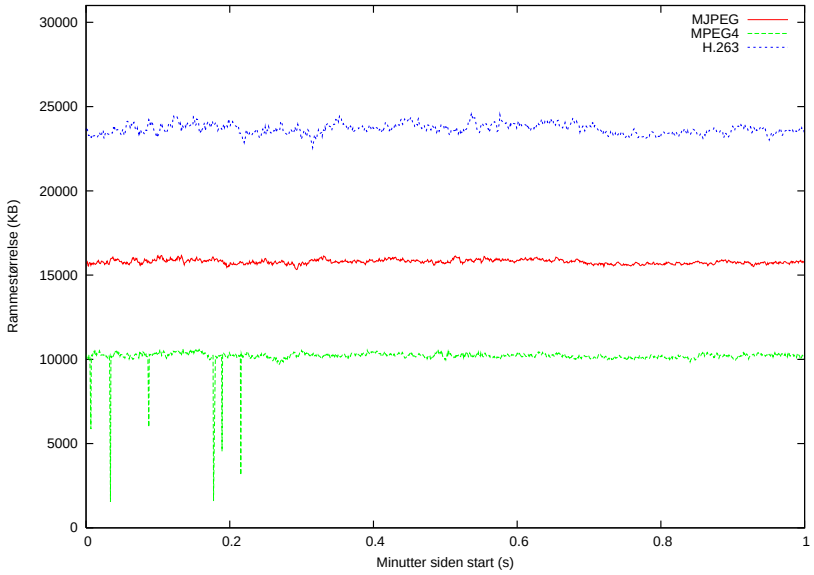
I alle testene bruker H.263 100% prosessorkraft og klarer ikke å sende 25 bilder i sekundet. Dette forklarer hvorfor video oppfattes som hakkende og forsinkelsen høy. MPEG4 må også noen ganger gå ned på bildefrekvensen, noe som vil føre til hakking der også.

Denne målingen burde vært gjort mer nøyaktig med et profileringsverktøy på tjeneren slik at gjennomsnittlig prosessorbruk kunne måles.

5.4 Båndbreddekrav

Under alle testene i 5.1 ble båndbreddebruken målt ved å se på størrelsen av en innkommende ramme[13] før det dekomprimeres. Båndbreddebruk over tid regnes ut ved å summere størrelsen på rammene og dele på tiden. Antall bilder per sekund kan også regnes ut ved å telle antall bilder og dele på antall sekunder. Dette er også gjort, for å støtte opp under tallene over med mer konkrete tall målt i klienten.

Figur 41: Sammenligning av rammestørrelse for MJPEG, MPEG4 og H.263 med utendørs natur. H.263 øverst, MJPEG i midten og MPEG4 nederst.



Utendørs natur. Her er båndbreddebruken høyest. Figur 41 viser størrelsen i byte per ramme i løpet av det første minuttet av denne testen. Den viser tydelig at H.263 har størst gjennomsnittlig rammestørrelse, MJPEG nest størst og MPEG4 minst.

[13] komprimert bilde

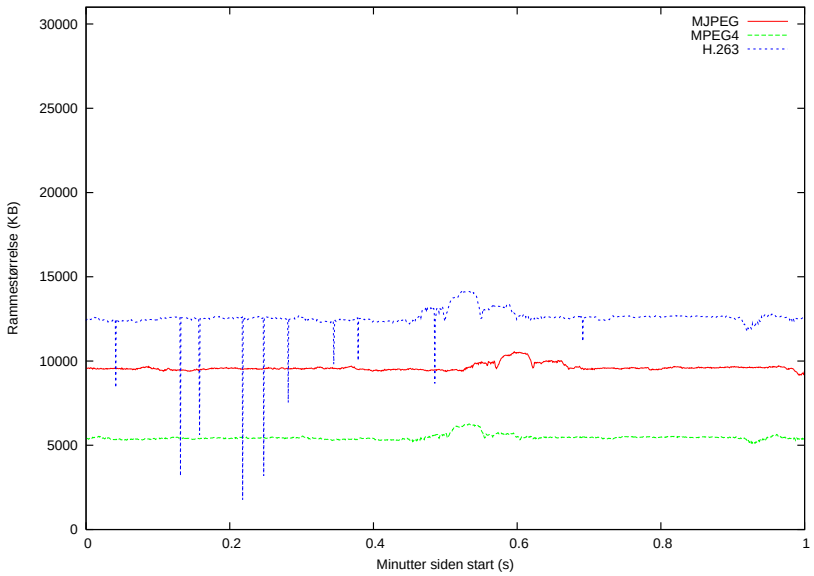
Tabell 19: Statistikk for båndbreddebruk under utendørs omgivelser.

	MJPEG	**MPEG4**	**H.263**
Gj.snittlig størrelse pr. ramme (byte):	15830.99	10240.20	23761.96
Gj.snittlig båndbreddebruk (kbit/sek):	3058.46	1883.22	3122.19
Gj.snittlig bildefrekvens (fps):	24.73	23.54	16.82

Statistikken oppsummeres i tabell 19, som viser at MJPEG nesten klarer 25 bilder i sekundet, med gjennomsnittlig båndbreddebruk på rundt 3.1 Mbit/sekund. MPEG4 klarer ikke å kode alle bildene, men bruker en gjennomsnittlig båndredde på rundt 1.9 Mbit/sekund . Grunnen til at H.263 har omtrent samme båndbreddebruk som MJPEG, samtidig som rammestørrelsen er 8 KB større, er at bildefrekvensen er lavere. H.263 klarer kun 17 bilder i sekundet.

Kurvene til MPEG4 i figur 41 har noen hakk som viser at det kommer mindre bilder med sporadiske mellomrom. Disse bildene er ødelagte bilder fra tjeneren som ikke kan dekodes. Bortsett fra disse hakkene viser figuren at MPEG4 har nesten konstant båndbreddebruk, H.263 har flere svingninger.

Figur 42: Sammenligning av rammestørrelse for MJPEG, MPEG4 og H.263 innendørs med lite bevegelse. H.263 øverst, MJPEG i midten og MPEG4 nederst.



Innendørs med lite bevegelse. Figur 42 på forrige side, viser at gjennomsnittlig rammestørrelse fremdeles er størst for H.263, nest størst for MJPEG og minst for MPEG4.

Tabell 20: Statistikk for båndbreddebruk under innendørs omgivelser med lite bevegelse.

	MJPEG	**MPEG4**	**H.263**
Gj.snittlig størrelse pr. ramme (byte):	9661.75	5505.93	12649.66
Gj.snittlig båndbreddebruk (kbit/sek):	1887.59	1012.77	1714.92
Gj.snittlig bildefrekvens (fps):	25.01	23.54	17.35

Tabell 20 viser at MJPEG nå klarer å sende strømmen uten å kaste bilder, og bruker i snitt 1.9 Mbit/sekund båndbredde. MPEG4 klarer kun å kode 24 bilder i sekundet, noe som fører til hakkete fremvisning. I snitt bruker MPEG4 5,5 KB per ramme. H.263 klarer fremdeles ikke å kode alle bildene, med denne sekvensen klarer den rundt 18 bilder i sekundet, dette gjør at båndbreddebruken er lavere enn for MJPEG selv om gjennomsnittlig rammestørrelse er nesten 3 KB større.

Rammestørrelsekurven til H.263 som har avvik forårsaket av ødelagte bilder.

Innendørs med mye bevegelse. Forventningen for denne testen var at gjennomsnittlig rammestørrelse ville være høyere her enn i forrige test. Men resultatene viser det motsatte. Dette kan forklares ved at kompleksiteten i bildet blir mindre fordi etterslep gjør bildet mindre skarpt.

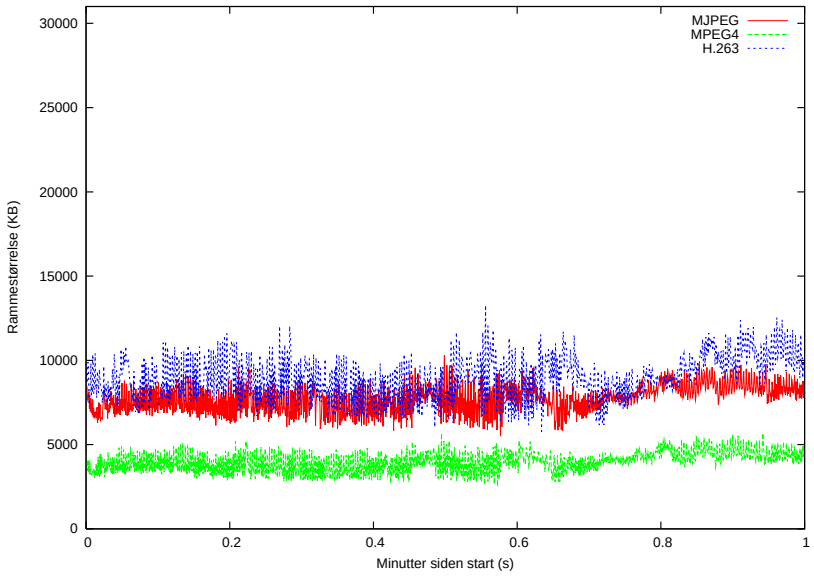
Figur 43 på neste side viser at forholdet mellom kodekene er omtrent det samme, med unntak av H.263 som ser ut til å nærme seg MJPEG. Figuren viser også at rammestørrelsen varierer mye hyppigere nå enn den har gjort i de andre testene.

Tabell 21: Statistikk for båndbreddebruk under innendørs omgivelser med mye bevegelse.

	MJPEG	**MPEG4**	**H.263**
Gj.snittlig størrelse pr. ramme (byte):	7570.82	3901.75	8613.81
Gj.snittlig båndbreddebruk (kbit/sek):	1464.54	762.28	1362.49
Gj.snittlig bildefrekvens (fps):	24.76	25.01	20.25

Tabell 21 viser statistikk for denne testen. Den gjennomsnittlige rammestørrelsen er mindre i sekvens 2 for alle kodekene, forskjellen er størst med H.263, hvor rammene i snitt er 4 KB mindre. H.263 klarer å sende 20 bilder i sekundet, de andre kodekene klarer 25 bilder i sekundet.

Figur 43: Sammenligning av rammestørrelse for MJPEG, MPEG4 og H.263 innendørs med mye bevegelse. H.263 øverst, MJPEG i midten og MPEG4 nederst.



5.5 Oppsummering

Tabell 22 på neste side oppsummerer resultatene. Tabellen viser at MJPEG er den kodeken som har fungert best når det gjelder bildekvalitet, forsinkelse, prosessorbruk og bildefrekvens. H.263 har også god kvalitet, men den har høy forsinkelse. MPEG4 bruker minst båndbredde, men dette går ut over kvaliteten på bildet. Tjenermaskinen har problemer med å komprimere video raskt nok med MPEG4 og H.263 kodekene, dette forklarer noe av hakkingen.

Den anbefalte kodeken i dette oppsettet er MJPEG, da bildeflyt og forsinkelse er meget viktig. Båndbreddekravet er tilstrekkelig lavt til at 10 eller flere videostrømmer kan sendes samtidig i et dedikert lokalnett.

Dersom strømmen skal gå over internett, bør MPEG4 vurderes fordi den har lavere krav til båndbredde. Men for dette kreves en raskere tjenermaskin.

5.5 Oppsummering

Tabell 22: Oppsummering av resultater.

	MJPEG	**MPEG4**	**H.263**
Subjektiv bildekvalitet			
Stillbilde utendørs:	Bra farger og kontrast, støy rundt kanter	Dårlige farger og kontrast, synlige blokker	Jevnt og pent bilde
Bevegelsesflyt utendørs:	Jevne bevegelser	Hakkete, hvite blink	Hakkete, hvite blink
Stillbilde innendørs:	Bra farger og detaljer, noe urenheter	Dårlige farger og kontrast, blokker uten farge	Klare farger, jevnt bilde, blokker på veggen.
Bevegelsesflyt innedørs:	Bra, jevne bevegelser, lav forsinkelse	Ganske jevne bevegelser, litt forsinkelse	Noe hakkete, hvite blink, påfallende forsinkelse
Stillbilde innedørs med bevegelse:	Skarpt. Unaturlige fargeoverganger på hånd.	Veldig uklart, dårlige farger, tydelige blokker.	Jevnt og behagelig, blokker på bakgrunn og hånd.
Bevegelsesflyt innedørs med bevegelse:	Jevne og fine bevegelser, lav forsinkelse	Noe hakking, lav forsinkelse, men varierende	Hakker veldig, hvite blink, høy forsinkelse
Målt forsinkelse og forsinkelsesvarians			
Gjennomsnittlig forsinkelse:	87 ms.	196 ms.	402 ms.
Empirisk standardavvik:	36 ms.	52 ms.	87 ms.
Prosessorbruk i QuickTime Broadcaster			
Utendørs	68-86%	90-100%	96-100%
Innendørs	60-82%	90-98%	96-100%
Innendørs med mye bevegelse	67-86%	90-100%	96-100%
Båndbreddebruk og bildefrekvens			
Rammestørrelse utendørs:	15,8 KB.	10,2 KB.	23,8 KB.
Bildesfrekvens utendørs:	25 fps.	24 fps.	17 fps.
Rammestørrelse innendørs:	9,7 KB.	5,5 KB.	12,6 KB.
Bildesfrekvens innendørs:	25 fps.	24 fps.	17 fps.
Rammestørrelse innendørs med bevegelse:	7,6 KB.	3,9 KB	8,6 KB.
Bildesfrekvens innendørs med bevegelse:	25 fps.	25 fps.	20 fps.

6 Konklusjon

Denne oppgaven har vist at det er mulig for “Hvermannsen” å sette opp et system for redigering av distribuert sanntidsvideo. Den totale kostnaden på prosjektet er på godt under 50000 kroner, et tilsvarende system vil i dag koste under 30000 kroner. PD/GEM brukes av kunstnere allerede i dag[58], muligheten for redigering av video fra nett åpner for nye muligheter.

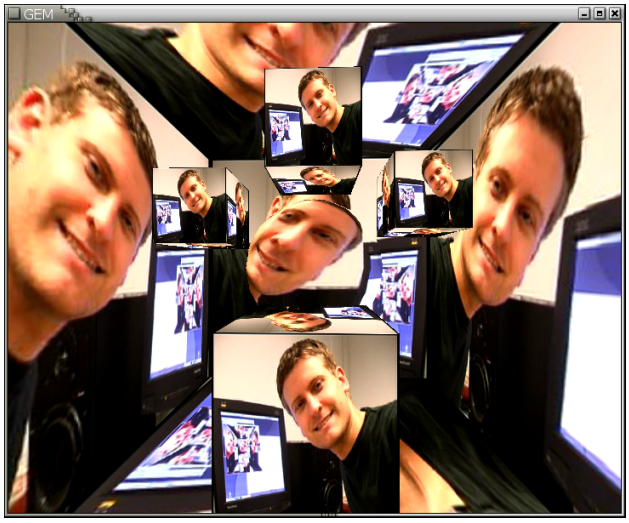
Løsningen dekker flesteparten av brukerens krav til funksjonalitet, mest fordi programmet er veldig fleksibelt. Siden integreringen av nettversksfunksjonaliteten ikke gjør dyptgripenden endringer av PD/GEM og heller ikke er avhengig av PD/GEMs funksjonalitet utover datastrukturer for overføring av bilde, kan den lett flyttes over til annen programvare.

Eksperimenter viser at MJPEG er den best egnede kodeken for overføring av video i et system for redigering av distribuert sanntidsvideo. Målingene fra eksperimentene viser at MJPEG konsekvent har lavest forsinkelse samtidig som båndbreddebruken ikke er høyere enn at det er mulig å kjøre flere samtidige strømmer på et 100Mbit nett. Bildekvaliteten var også god på MJPEG.

Med MJPEG var gjennomsnittlig total forsinkelse på bildet 100 millisekunder. Tjeneren var til tider høyt belastet, og målinger viste at økt belastning fører til økt forsinkelse. Det er derfor grunn til å tro at en raskere tjener kunne fått forsinkelsen ytterligere ned. Den minste målte forsinkelsen var på 13 millisekunder, som tilsvarer under en rammes (1/25 sekund) forsinkelse. Oppgaven har også vist at det er mulig å forbedre forsinkelsen fra QuickTime Streaming Server kun ved redigering av en konfigurasjonsfil, det er også testet ulike parametre og funnet et oppsett med minimal forsinkelse.

6.1 Er problemstillingen løst?

Et system for redigering av sanntidsvideo over nett er satt opp. Det er utviklet en klient med komponenter fra programmer med åpen kildekode som støtter forskjellige kodeker og kan bygges ut til å støtte flere. Se figur 44 på neste side for et eksempelsystem.



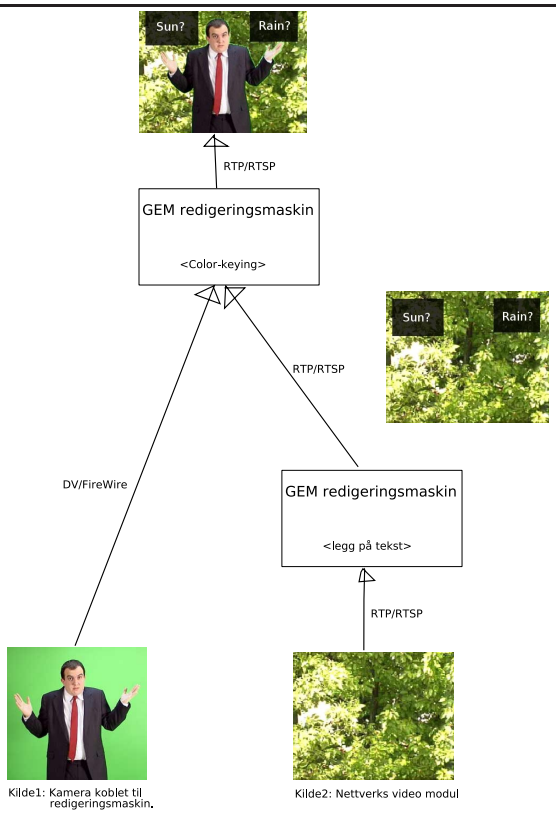
Figur 44: Bilde fra et testoppsett i GEM som bruker video overført over nett. Video projiseres på en kule og noen bokser som roterer rundt kula. Boksene og kula er igjen inne i en større boks, som roterer motsatt vei.

Forsinkelsen på systemet er bra nok for mange bruksområder, særlig når synkronisering mellom lyd og bilde ikke er et poeng.

6.2 Videre arbeid

Streaming fra GEM. Grunnideen bak denne oppgaven er et system for distribuert redigering av video (se 1.3 på side 6). Denne oppgaven løser problemet med hvordan video skal kunne hentes fra en maskin på et nett og brukes i et videoredigeringssystem. Neste steg er å se på hvordan video kan sendes videre fra videoredigeringssystemet og ut til en annen maskin på nettet.

Dersom denne funksjonaliteten implementeres i GEM, for eksempel basert på libavcodec for å kode video og liveMedia for å sende den ut over RTP/RTSP, vil video kunne sendes fra maskin til maskin gjennom nettet, og behandles på veien. Se figur 45 på neste side for et eksempeloppsett hvor to maskiner med GEM samarbeider om å redigere i et distribuert system.



Figur 45: Eksempelsystem hvor en maskin legger tekst på video, og en annen kjører colorkeying for å tilsammen gi et resultat.

Gjenta tester på en kraftigere tjener. Tjenermaskinen som ble brukt i testene her hadde problemer med å levere MPEG4 og særlig H.263 fort nok til å gi 25 bilder i sekundet. Testene burde vært utført på en raskere tjener for å gi et bedre bilde av hva selve kodekene klarer, og ikke bare hva tjeneren klarer å gi med kodekene.

Hindre kvalitetstap gjennom flere redigeringer av komprimert video. For hver gang video dekomprimeres og komprimeres igjen, med en kodek som ikke er tapsfri, blir noe av kvaliteten tapt. For å oppnå et system for distribuert redigering må dette tapet forhindres. Ulike metoder for å forhindre dette må undersøkes og implementeres i GEM.

6.2 Videre arbeid

Netverks Video Modul. Et tenkt bruksområde for resultatet av denne oppgaven er behandling av video til bruk ved overvåking og sikkerhet. Såkalte Nettverks Video Moduler, små maskiner med nett og videokilde, kan brukes for å sende videostrømmer til en sentral maskin som behandler videosignalene, ser etter bestemte faresignaler og viser behandlet video til overvåkingspersonell. Slike videomoduler har i dag meget begrenset prosessorkraft grunnet krav til størrelse og strømforbruk, samt kostnad. Det er derfor en utfordring å få satt opp en videostrøm med bra nok kvalitet og liten nok båndbredde. Ulike metoder for å få til dette må undersøkes.

Optimalisering av QSS for andre kodeker enn MJPEG. I 4.2.3 på side 71 ble ulike opsjoner i QuickTime Streaming Server justert for å få minst mulig forsinkelse. Dette ble kun testet med MJPEG, de samme testene bør også gjøres med andre kodeker for å se om andre innstillinger er mer optimale for disse.

Legge inn støtte for JPEG2000 i liveMedia og libavcodec. JPEG2000 lover høyere kompresjon og bedre kvalitet enn standard JPEG. Det er derfor interessant å legge inn støtte for JPEG2000 i liveMedia og libavcodec slik at dette kan benyttes i redigeringssystemet.

Detaljmåle kildene til forsinkelse Nå som systemet er ferdig utviklet er det mulig å legge inn målinger flere steder for å måle forsinkelsen i hvert ledd. Dette er interessant for å se hvor forbedringer må gjøres for å minske forsinkelsen ytterligere. Denne oppgaven antyder at komprimeringen er største kilde til forsinkelse, da det er stor forskjell på forsinkelsen med de forskjellige kodekene. Men dette er ikke bevist. Det er også interessant å se på hvor lang tid det tar å komprimere et bilde for de forskjellige kodekene.

A Kildekode

Listing 1: filmFFMPEG.cpp

```
////////////////////////////////////////////////////
//
// GEM - Graphics Environment for Multimedia
//
// zmoelnig@iem.kug.ac.at
//
// Implementation file
//
//    Copyright (c) 1997-1999 Mark Danks.
//    Copyright (c) Günther Geiger.
//    Copyright (c) 2001-2002 IOhannes m zmoelnig. forum::für::umläute. IEM
//    Copyright (c) 2004 Henning Kulander
//    liveMedia integration based on code from MPlayer 1.0-pre4
//    For information on usage and redistribution, and for a DISCLAIMER OF ALL
//    WARRANTIES, see the file, "GEM.LICENSE.TERMS" in this distribution.
//
////////////////////////////////////////////////////
#define MAX_RTP_FRAME_SIZE 500000
#define STATISTICS 1
#define LATENCY 1

#include "Pixes/filmFFMPEG.h"
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

#include <fstream>
#include <cstdlib>

////////////////////////////////////////////////////
//
// filmFFMPEG
//
////////////////////////////////////////////////////
// Constructor
//
////////////////////////////////////////////////////

filmFFMPEG :: filmFFMPEG(int format) : film(format) {
  static bool first_time=true;
  if (first_time) {
#ifdef HAVE_LIBFFMPEG
    post("pix_film:: libffmpeg support");
#endif
    first_time = false;
  }
#ifdef HAVE_LIBFFMPEG
  m_Format=NULL;
  m_Pkt.data=NULL;

  av_register_all(); /* Let FFMPEG know what codecs/formats are available */
#endif

#ifdef HAVE_LIVEMEDIA
  mediaSession = NULL;
  rtspClient = NULL;
  env = NULL;
  scheduler = NULL;

  using_livemedia = false;
#endif

}

////////////////////////////////////////////////////
// Destructor
//
////////////////////////////////////////////////////
filmFFMPEG :: ~filmFFMPEG()
{
  close();
}
#ifdef HAVE_LIBFFMPEG
void filmFFMPEG :: close(void)
{
#ifdef HAVE_LIVEMEDIA
  if (using_livemedia && mediaSession) {
```

```
78      if (rtspClient != NULL) {
          MediaSubsessionIterator iter(*mediaSession);
80        MediaSubsession* subsession;

82        while ((subsession = iter.next()) != NULL) {
            rtspClient->teardownMediaSubsession(*subsession);
84        }
        }
86      Medium::close(mediaSession);
        Medium::close(rtspClient);
88
        delete(sdpDescription);
90
        avcodec_close(videoBufferQueue->codec);
92      delete(videoBufferQueue);

94      delete(env);
        delete(scheduler);
96    }
  #endif
98
     if (m_Format){
100     avcodec_close(&m_Format->streams[m_curTrack]->codec);
        av_close_input_file(m_Format);
102     m_Format=0;
      }
104 }

106 ////////////////////////////////////////////////////////
  // really open the file ! (OS dependent)
108 //
  ////////////////////////////////////////////////////////
110 bool filmFFMPEG :: open(char *filename, int format)
  {
112   AVFormatParameters *ap=0;
      AVCodec* codec=0;
114
      int err, i;
116
      if (format>0)
118     m_wantedFormat=format;

120 #ifdef HAVE_LIVEMEDIA
      using_livemedia=0;
122   if ( !strncmp(filename, "rtsp://", 7)) { /* Use liveMedia to read rtsp */
        using_livemedia=1;
124     scheduler = BasicTaskScheduler::createNew();
        if (scheduler == NULL) {
126       error("Couldn't create liveMedia scheduler");
          goto unsupported;
128     }
        env = BasicUsageEnvironment::createNew(*scheduler);
130     if (env == NULL) {
          error("Couldn't create liveMedia Usage Environment");
132       goto unsupported;
        }
134     rtspClient = RTSPClient::createNew(*env, VERBOSE, "GEM");
        if (rtspClient == NULL) {
136       error("Failed to create RTSP client: %s", env->getResultMsg());
          goto unsupported;
138     }
        /* RTSP: send DESCRIBE call to server */
140     sdpDescription = rtspClient->describeURL(filename);
        if (sdpDescription == NULL) {
142       error("Failed to get a SDP description from URL \"%s\": %s\n",
              filename, env->getResultMsg());
144       goto unsupported;
        }
146     mediaSession = MediaSession::createNew(*env, sdpDescription);
        if (mediaSession == NULL) {
148       error("Failed to create Media Session");
          goto unsupported;
150     }

152     // Create RTP receivers (sources) for each subsession:
        MediaSubsessionIterator iter(*mediaSession);
154     MediaSubsession* subsession;
        unsigned desiredReceiveBufferSize;
156     while ((subsession = iter.next()) != NULL) {
          // Ignore any subsession that's not audio or video:
158       if (strcmp(subsession->mediumName(), "audio") == 0) {
            desiredReceiveBufferSize = AUDIO_BUFFER_SIZE;
160       } else if (strcmp(subsession->mediumName(), "video") == 0) {
```

```
        desiredReceiveBufferSize = VIDEO_BUFFER_SIZE;
      } else {
        continue;
      }
      if (!subsession->initiate()) {
        error("Failed to initiate \"%s/%s\" RTP subsession: %s\n",
             subsession->mediumName(), subsession->codecName(),
             env->getResultMsg());
      } else {
        post("Initiated \"%s/%s\" RTP subsession\n",
             subsession->mediumName(), subsession->codecName());

        // Set the OS's socket receive buffer sufficiently large to avoid
        // incoming packets getting dropped between successive reads from this
        // subsession's demuxer.  Depending on the bitrate(s) that you expect,
        // you may wish to tweak the "desiredReceiveBufferSize" values above.
        int rtpSocketNum = subsession->rtpSource()->RTPgs()->socketNum();
        int receiveBufferSize
          = increaseReceiveBufferTo(*env, rtpSocketNum,
                                    desiredReceiveBufferSize);
#if VERBOSE
          post("Increased %s socket receive buffer to %d bytes \n",
               subsession->mediumName(), receiveBufferSize);
#endif

        /* RTSP: send SETUP call to server */
        if (rtspClient != NULL) {
          // Issue a RTSP "SETUP" command on the chosen subsession:
          if (!rtspClient->setupMediaSubsession(*subsession, False,
                                                RTSP_STREAM_OVER_TCP))
            break;
        }
      }
    } /* End subsession iteration loop *
    /* RTSP: send PLAY call to server */
    if (rtspClient != NULL) {
      if (!rtspClient->playMediaSession(*mediaSession)) {
        error("Error issuing RTSP Play command to server");
        goto unsupported;
      }
    }

    /* Need to get Codec info from streams and setup ffmpeg to use info */
    iter.reset();
    while ((subsession = iter.next()) != NULL) {
      if (subsession->readSource() == NULL)
        continue; // not reading this

      unsigned flags = 0;
      if (strcmp(subsession->mediumName(), "audio") == 0) {
        post("liveMedia: Ignoring audio stream");
        continue;
      } else if (strcmp(subsession->mediumName(), "video") == 0) {
        videoBufferQueue = new ReadBufferQueue(subsession);
        /* This is where we set the codecinfo */
        videoBufferQueue->m_Pkt = &m_Pkt;
        videoBufferQueue->codec = avcodec_alloc_context();
        videoBufferQueue->GEM_Object = this;

        codec = avcodec_find_decoder_by_name(subsession->codecName());
        if (codec == 0) {
          if (!strcmp(subsession->codecName(),"JPEG")) {
            codec = avcodec_find_decoder(CODEC_ID_MJPEG);
          } else if (!strcmp(subsession->codecName(),"MP4V-ES")) {
            codec = avcodec_find_decoder(CODEC_ID_MPEG4);
            unsigned configLen;
            videoBufferQueue->codec->extradata =
               parseGeneralConfigStr(subsession->fmtp_config(), configLen);
            videoBufferQueue->codec->extradata_size=configLen;
          } else if (!strcmp(subsession->codecName(),"H263-1998")) {
            codec = avcodec_find_decoder(CODEC_ID_H263);
          } else {
            error("Couldn't find codec for %s", subsession->codecName());
            goto unsupported;
          }
        }
        post("Found codec: %d(%s)", codec->id, codec->name);
        if ( (err = avcodec_open(videoBufferQueue->codec, codec)) < 0) {
          error("Error opening codec! (%d)", err);
          goto unsupported;
        }

        m_image.image.xsize = subsession->videoWidth();
```

```
244           m_image.image.ysize = subsession->videoHeight();
            }
246       }
    } else /* Don't use liveMedia */
248 #endif
      {
250       ap = (AVFormatParameters *) av_mallocz( sizeof *ap );
  #if LIBAVCODEC_BUILD >= 4700
252       ap->initial_pause = 1; /* Force pause when starting RTSP stream */
  #endif
254       ap->image_format = 0;

256       if (m_Format) {
            av_free(m_Format);
258         m_Format = 0;
          }
260
          err = av_open_input_file(&m_Format,filename,NULL,0,ap);
262
          if (err < 0) {
264         error("GEM: pix_film (ffmpeg): Unable to open file: %s %d",
                filename, err);
266         goto unsupported;
          }
268
          /* Start the stream if it is an RTSP stream or similar */
270 #if LIBAVCODEC_BUILD >= 4700
          av_read_play(m_Format);
272 #endif

274       err = av_find_stream_info(m_Format);
          if (err < 0) {
276         error("pix_film: can't find stream info for %s",filename);
            goto unsupported;
278       }

280       m_numTracks = m_Format->nb_streams;
          for (i=0;i<m_Format->nb_streams;i++) {
282         if ((codec=avcodec_find_decoder(m_Format->streams[i]->codec.codec_id))
              == 0)
284           {
              error("pix_film: Can't find decoder for %s",
286                 m_Format->streams[i]->codec.codec_id);
              goto unsupported;
288           }
            if (!codec || avcodec_open(&m_Format->streams[i]->codec, codec) < 0)
290           {
              error("pix_film: Can't open codec");
292           goto unsupported;
            }
294         if (m_Format->streams[i]->codec.codec_type == CODEC_TYPE_VIDEO)
              break; // We found our first video stream, don't look for more
296       }
          if (i == m_Format->nb_streams) {
298         error("No valid stream found. Checked %d.\n", i);
            goto unsupported;
300       }
          m_curTrack = i;  // remember the stream
302       m_image.image.xsize = m_Format->streams[i]->codec.width;
          m_image.image.ysize = m_Format->streams[i]->codec.height;
304
          av_free(ap);
306     }

308   // This part is executed regardless of liveMedia use:
      // ---------------------------------------------------
310
      // Get the length of the movie
312   m_numFrames = -1;
      m_curFrame = 0;
314   first_picture = 1;

316   // get all of the information about the stream
      m_image.image.type  = GL_UNSIGNED_BYTE;
318   m_image.image.format= m_wantedFormat;
      post("Got width and height (%dx%d@%d)", m_image.image.xsize,
320        m_image.image.ysize, m_image.image.format);

322   m_image.image.upsidedown = 1;
      m_image.image.reallocate();
324
      m_avFrame = avcodec_alloc_frame();
326
```

```
    return true;
328  unsupported:
    if (ap)
330   av_free(ap);
    post("FFMPEG: unsupported!");
332   close();
    return false;
334 }

336 #ifdef HAVE_LIVEMEDIA
    static void afterReading(void* clientData, unsigned frameSize,
338                          unsigned /*numTruncatedBytes*/,
                             struct timeval presentationTime,
340                          unsigned /*durationInMicroseconds*/) {
    ReadBufferQueue* bufferQueue = (ReadBufferQueue*)clientData;
342   struct timeval current_time;
    struct timezone tz;
344   Boolean hasBeenSynchronized =
      bufferQueue->rtpSource()->hasBeenSynchronizedUsingRTCP();
346
    #if STATISTICS
348   gettimeofday(&current_time, &tz);

350   if (hasBeenSynchronized) {
      if (bufferQueue->initial_delta == 0)
352   {
        post("Time info: %6.4f %6.4f %d %d %d %d",
354           ((double) presentationTime.tv_sec
              + (double) presentationTime.tv_usec/1000000),
356           ((double) current_time.tv_sec
              + (double) current_time.tv_usec/1000000),
358           presentationTime.tv_sec,
              presentationTime.tv_usec,
360           current_time.tv_sec,
              current_time.tv_usec);
362     post("Timestamp\t Size\t Drift\t Init_Delta\t Latency\t Jitter\t");
        bufferQueue->initial_delta = ((double) presentationTime.tv_sec
364                                   + (double) presentationTime.tv_usec/1000000)
          - ((double) current_time.tv_sec
366           + (double) current_time.tv_usec/1000000);
        bufferQueue->latency=0;
368   }
      float drift = ((double) presentationTime.tv_sec
370                 + (double) presentationTime.tv_usec/1000000)
      - ((double) current_time.tv_sec
372     + (double) current_time.tv_usec/1000000) - bufferQueue->initial_delta;

374   RTPReceptionStatsDB::Iterator statsdb(bufferQueue->rtpSource()->receptionStatsDB());
      RTPReceptionStats *stats = statsdb.next();
376   post("%6.4f %d\t %6.4f\t %6.4f\t %6.4f\t %u",
           (double) presentationTime.tv_sec
378        + (double) presentationTime.tv_usec/1000000,
           frameSize,
380        drift,
           bufferQueue->initial_delta,
382        bufferQueue->latency,
           stats->jitter());
384

    }
386 #endif
    bufferQueue->m_Pkt->size = frameSize;
388
    // Signal any pending 'doEventLoop()' call on this queue:
390 bufferQueue->timeout_flag = false;
    bufferQueue->blockingFlag = ~0;
392 }

394 static void onTimeout(void* clientData) {
    ReadBufferQueue* bufferQueue = (ReadBufferQueue*)clientData;
396
    bufferQueue->timeout_flag = true;
398
    // Signal any pending 'doEventLoop()' call on this queue:
400 bufferQueue->blockingFlag = ~0;
  }
402
  static void onSourceClosure(void* clientData) {
404 ReadBufferQueue* bufferQueue = (ReadBufferQueue*)clientData;

406 // Signal any pending 'doEventLoop()' call on this queue:
    bufferQueue->blockingFlag = ~0;
408 }
```

```
410 #endif
    ////////////////////////////////////////////////////
412 // render
    //
414 ////////////////////////////////////////////////////
    pixBlock* filmFFMPEG :: getFrame(){
416   int i;
      int gotit = 0;
418   int ret;
      AVCodecContext *codec;
420
    #ifdef HAVE_LIVEMEDIA
422   if (m_Format || using_livemedia) {
    #else
424   if (m_Format) {
    #endif
426     if (!m_readNext){
        return &m_image;
428     }
        m_readNext = false; /* FIXME: This is a critical region, need lock */
430
        while (!gotit) {
432       if (m_Pkt.data)
    #ifdef HAVE_LIVEMEDIA
434         if (using_livemedia && !videoBufferQueue->timeout_flag)
    #endif
436         av_free_packet(&m_Pkt);
    #ifdef HAVE_LIVEMEDIA
438       if (using_livemedia) {
          if (videoBufferQueue == NULL) {
440         error("No RTP video stream configured\n");
            break;
442         }
            if (!videoBufferQueue->timeout_flag)
444           if ((ret = av_new_packet(&m_Pkt,MAX_RTP_FRAME_SIZE)) < 0) {
              error("Error %d allocating new AVPacket", ret);
446           break;
            }
448
          videoBufferQueue->blockingFlag = 0;
450       if (!videoBufferQueue->timeout_flag) {
            videoBufferQueue->readSource()->getNextFrame(m_Pkt.data,
452                                                         MAX_RTP_FRAME_SIZE,
                                                            afterReading,
454                                                         videoBufferQueue,
                                                            onSourceClosure,
456                                                         videoBufferQueue);
          }
458       videoBufferQueue->timeout_flag = false;
          TaskScheduler& scheduler =
460         videoBufferQueue->readSource()->envir().taskScheduler();
          TaskToken timeout =
462         scheduler.scheduleDelayedTask(TIMEOUT, onTimeout, videoBufferQueue);

464       //Block here waiting for picture or timeout
          scheduler.doEventLoop(&videoBufferQueue->blockingFlag);
466
          if (videoBufferQueue->timeout_flag) { // No new picture...
468         m_readNext = true;
            gotit = false;
470         m_image.newimage=0;
            return &m_image;
472       } else {                              // New picture, cancel timeout
            scheduler.unscheduleDelayedTask(timeout);
474       }
          codec = videoBufferQueue->codec;
476     } else
    #endif
478 #if LIBAVCODEC_BUILD >= 4700
          if (av_read_frame(m_Format,&m_Pkt) < 0) {
480 #else
          if (av_read_packet(m_Format, &m_Pkt) < 0) {
482 #endif
            error("Error reading frame, trying to rewind!");
484 #if LIBAVCODEC_BUILD >= 4700
            av_seek_frame(m_Format,-1,m_Format->start_time);
486 #endif
            // ?? TODO is this the only way to say goodbye
488         m_numFrames = m_Format->streams[m_curTrack]->codec.frame_number;
            m_readNext = true;
490         gotit = false;
            m_image.newimage=0;
492         return &m_image;
```

```
        }
#ifdef HAVE_LIVEMEDIA
      if (!using_livemedia)
#endif
      {
        codec = &m_Format->streams[m_curTrack]->codec;
      }

#if LIBAVCODEC_VERSION_INT >= 0x000408
      ret = avcodec_decode_video(codec,
                                 m_avFrame,
                                 &gotit,
                                 m_Pkt.data,
                                 m_Pkt.size);
#else
      error("Your version of ffmpeg is to old...");
#endif

      if (ret < 0) { // TODO recover gracefully
        error("Decoding video failed: %d", ret);
        break;
      }

      for(int i=0;i<4;i++){
        m_Picture.data[i]=m_avFrame->data[i];
        m_Picture.linesize[i]=m_avFrame->linesize[i];
      }

      if (gotit) {
        AVPicture rgba;
        uint8_t *data;
        int dstfmt=0;
        int ret;

        switch(m_wantedFormat){
        case GL_LUMINANCE:      dstfmt = PIX_FMT_GRAY8;  break;
        case GL_YCBCR_422_GEM: dstfmt = PIX_FMT_YUV422; break;
        default:
        case GL_RGBA:           dstfmt = PIX_FMT_RGBA32; break;
        }
        m_image.image.setCsizeByFormat(m_wantedFormat);

        int width  = m_image.image.xsize = codec->width;
        int height = m_image.image.ysize = codec->height;
        int fmt = codec->pix_fmt;

        // Set up a temporary AVPicture:
        data = (uint8_t *) av_malloc(width*height*m_image.image.csize);
        ret = avpicture_fill(&rgba,(uint8_t *) data,dstfmt,width,height);
        if (ret<0) {
          error("Failed to initialize destination picture. Bad format: %d",
                dstfmt);
          break;
        }

        ret = img_convert(&rgba,dstfmt,&m_Picture,fmt,width,height);
        if (ret<0) {
          error("pix_film: image conversion failed (%d->%d)", fmt, dstfmt);
          break;
        }
        m_curFrame = codec->frame_number;
        m_image.image.fromBGRA((uint8_t *) data);
        av_free(data);

#if STATISTICS
#if LATENCY
        if ( videoBufferQueue->rtpSource()->hasBeenSynchronizedUsingRTCP() ) {
          struct timeval current_time;
          struct timezone tz;

          gettimeofday(&current_time, &tz);

          if (m_image.image.data[(width/2*height+height/2)*m_image.image.csize]
              > 80 ||
              m_image.image.data[(width/2*height+height/2)*m_image.image.csize+4]
              > 80)
            {
              if (flash_time.tv_sec)
                {
                  /* Picture has returned, calculate delay */
                  videoBufferQueue->latency =
                    ((double) current_time.tv_sec +
                     (double) current_time.tv_usec / 1000000 -
```

```
576                     ((double) flash_time.tv_sec +
                         (double) flash_time.tv_usec / 1000000)) *1000;
578                 flash_time.tv_sec = 0;
                    flash_time.tv_usec = 0;
580               }
                  /* Cancel echo */
582               {
                    int i;
584                 for (i = 0; i < m_image.image.csize*width*height; i++)
                      m_image.image.data[i] = 0;
586               }

588               m_image.image.data[0] = 0;
                  m_image.image.data[1] = 255;
590               m_image.image.data[2] = 0;
                  m_image.image.data[3] = 255;
592             } else if (!(current_time.tv_sec % 2) && current_time.tv_usec < 200000) {
                flash_time.tv_sec = current_time.tv_sec;
594             flash_time.tv_usec = current_time.tv_usec;

596             int i;
                for (i = 0; i < m_image.image.csize*width*height; i++)
598               m_image.image.data[i] = 255;
            } else        /* Cancel echo */
600           {
                int i;
602             for (i = 0; i < m_image.image.csize*width*height; i++)
                  m_image.image.data[i] = 0;
604           }
        }
606


608
    #endif
610 #endif

612         m_image.newimage=1;

614         return &m_image;
          }
616     } /* End while loop */
    } /* End if m_Format */
618
    error("error while decoding");
620 m_readNext = true;
    gotit = false;
622 m_image.newimage=0;
    return 0;
624 }

626 int filmFFMPEG :: changeImage(int imgNum, int trackNum){
    m_readNext = true;
628
    return FILM_ERROR_DONTKNOW;
630 }
    #endif
632
    #ifdef HAVE_LIVEMEDIA
634 ReadBufferQueue::ReadBufferQueue(MediaSubsession* subsession)
      : prevPacketWasSynchronized(False), prevPacketPTS(0.0),
636   fReadSource(subsession == NULL ? NULL : subsession->readSource()),
      fRTPSource(subsession == NULL ? NULL : subsession->rtpSource()),
638   initial_delta(0.0), timeout_flag(false) {
    }
640
    ReadBufferQueue::~ReadBufferQueue() {
642 }
    #endif
```

B Ordforklaringer

I denne delen av oppgaven følger en del ordforklaringer på ord det ikke kan forventes at alle har kjenskap til. Ordene kommer både fra mediverdenen, kommunikasjonsverdenen og andre IT miljøer.

API: Forkortelse for Application Programming Interface. API er et grensesnitt programmerere bruker for å benytte funksjonalitet laget av andre programmerere. Funksjonaliteten legges ofte ut som biblioteker med funksjonskall.

COM: Forkortelse for Component Object Model. COM er Microsofts rammeverk for å lage og støtte programkomponentobjekter. COM gir underliggende støtte for grensesnitt forhandling, vurderer når et objekt kan fjernes fra et system, behandler lisenser og hendelser.

CVS: Forkortelse for Concurrent Versions System. CVS er et system for versjonskontroll som brukes av de fleste programpakker med åpen kildekode. En uoffisiell versjon hentet fra CVS kalles en CVS-versjon.

DSM-CC: Forkortelse for Digital Storage Medium Command and Control DSM-CC er en del av MPEG-2 standarden og beskriver hvordan en MPEG-2 strøm kan styres fra en annen maskin.

dæmon: Program som kjører i bakgrunnen uavhengig av interaksjon med brukeren. Benyttes ofte til tjenerprogramvare.

ix86: Et samlebegrep for Intels 32bit prosessorer. Begrepet gjelder i denne oppgaven Intel 386, 486 og Pentium serien. Kompatible prosessorer som AMDs Athlon faller også inn under dette begrepet.

Hvermansen: Hvermansen er et begrep som brukes for å beskrive folk flest, en fiktiv gjennomsnittsperson.

Kodek: Engelsk: Codec. Forkortelse for “Koder/Dekoder” (Eng: “Coder/Decoder” eller “Compressor/Decompressor”). En kodek er en algoritme for å kode til et format og dekode dette formatet igjen. Brukes mye innen digital video for å redusere plassbehov ved lagring og båndbreddebehov ved overføring.

MPEG: Motion Picture Expert Group etablert av den internasjonale standard organisasjonen (ISO) for å gi en basis for bilde koding, komprimeringssystem og transportsystem.

PAL: Phase Alternation Line. PAL er et farge TV system utviklet i Tyskland. PAL benyttes i mesteparten av Europa, Afrika, Australia og Sør-Amerika. PAL gir et interlaced bilde med 625 linjer, 25 bilder i sekundet (50 halve bilder i sekundet)

Piksel: Av engelsk pixel, Picture element. Et digitalt bilde er satt sammen av flere punkter. Hvert punkt kalles en piksel.

RGB: Fargemodell som basserer seg på at alle farger kan lages ved å kombinere rødt, grønt og blått lys. Brukes i de fleste skjermer, og er standard fargemodell på vanlig datagrafikk.

sammenflettet: Engelsk: interlaced. Et triks ofte brukt for å få bedre flyt i bevegelser uten å øke båndbredden på video data. Et bilde deles i to deler. Den ene delen inneholder alle odde linjer (linje 1, 3, 5 osv.) den andre inneholder alle partall linjer (linje 2, 4, 6 osv.). Når bildet vises, vises først den ene delen, så den andre. Effekten er en dobling av oppdateringshastigheten.

SMIL: Forkortelse for Synchronized Multimedia Integration Language. Dette er et språk for å definere sammenhenger mellom ulike multimedia objekter på web.

URL: Forkortelse for Uniform Resource Locator. En URL er adressen til en fil eller resurs tilgjengelig via Internett. En URL er vanligvis bygd opp på formen: “protokoll://maskin/fil”.

VeeJay: (VJ). Video Jockey. Tilsvarer begrepet DJ, som er en artist som presenterer musikk live for et publikum. En VeeJay er en artist som presenterer visuelle effekter live for et publikum.

YUV420: Fargemodell hentet fra TV-verdenen. Bildet deles i et sorthvit bilde, og to fargekanaler som brukes for å regne ut farger i sorthvit bildet. 420 betyr at fargekanalene har halvparten av oppløsningen til sorthvit bildet i vertikal og horisontal retning.

C Lister over figurer og tabeller

Figurer

Tabeller

Referanser

[1] Rick Karr. TechnoPop: The Secret History of Technology and Pop Music. Web pages, http://www.npr.org/programs/morning/features/2002/technopop/, 2002.

[2] Philippe Owerzarski. Enforcing Multipoint Multimedia Synchronisation in Videoconferencing Applications. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 65–76, 2000.

[3] Tom Tucker. Audio-to-Video Delay-Watermarking Provies a Means of Automatic Correction. *SMPTE-Journal*, 110(8):523–526, August 2001.

[4] Randall Hoffner. Audio-Video Synchronization Across DTV Transport Interfaces: The Impossible Dream? *SMPTE-Journal*, 109(11):881–884, November 2000.

[5] Kjell Bjørgeengen. NR/ PROSJEKTFORSLAG forsøk på presisering. Internt skriv, Norsk Regnesentral.

[6] Lisa Amini, Jorge Lepre, and Martin Kienzle. Distributed Stream Control for Self-Managing Media Processing Graphs. *ACM Multimedia*, Vol.2:s.99–102, 1999.

[7] Taehyun Kim and Jack Brassil. Dynamic Program Insertion in High Quality Video over IP. In *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 32–40. ACM Press, 2003.

[8] Yoshinobu Tonomura, Akihito Akutsu, Kiyotaka Otsuji, and Toru Sadakata. VideoMAP and VideoSpaceIcon: tools for anatomizing video content. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages s.131–136. ACM Press, 1993.

[9] Andreas Vogel, Brigitte Kerhervé, Gregor von Bochmann, and Jan Gecsei. Distributed Multimedia and Quality of Service: A Survey. *IEEE-MultiMedia*, 2(2):10–19, Summer 1995.

[10] Marco Lohse, Philipp Slusallek, and Patrick Wambach. Extended Format Definition and Quality-driven Format Negotiation in Multimedia Systems. In *Multimedia 2001 – Proceedings of the Eurographics Workshop*, pages 65–75. Springer Verlag Wien, 2001. Web pages, http://www.networkmultimedia.org/NMM/Publications/2001/egmm01-neg

[11] Network-Integrated Multimedia Middleware for Linux. Web pages, http://www.networkmultimedia.org/NMM/index.html.

[12] Marco Lohse and Philipp Slusallek. Middleware Support for Seamless Multimedia Home Entertainment for Mobile Users and Hetrogenous Environments. In *The 7th IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA)*, August 2003. Web pages, http://www.networkmultimedia.org/NMM/Publications/2003/middleware

[13] Discreet company homepage. Web pages, http://www.discreet.com.

[14] Terence Curren. Online Finishing Systems. *DV*, March:s18–24, 2002.

[15] GlobalCaster Studio homepage. Web pages, http://www.globalstreams.com/products/studio/.

[16] vizrt company homepage. Web pages, http://www.vizrt.com.

[17] Max homepage. Web pages, http://www.cycling74.com/products/max.html.

[18] NATO homepage. Web pages, http://www.eusocial.com/.

[19] Jeremy Bernstein. A discussion of NATO.0+55+3d modular. Web pages, http://www.bootsquad.com/nato/, 2001.

[20] PD (PureData) homepage. Web pages, http://www.crca.ucsd.edu/~msp/software.html.

[21] Johannes M. Zmölnig. HOWTO write an External for PureData. Web pages, http://iem.kug.ac.at/pd/externals-HOWTO/, 2001.

[22] GEM (Graphical Environment for Multimedia) Homepage. Web pages, http://gem.iem.at/.

[23] FreeJ Homepage. Web pages, http://freej.dyne.org/.

[24] EffecTV Homepage. Web pages, http://effectv.sourceforge.net/index.html.

[25] DirectShow Info from Microsoft. Web pages, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/

[26] Philipp Slusallek and Marco Lohse. Presentation: Multimedia Middleware. Web pages, http://graphics.cs.uni-sb.de/Courses/ss03/Multimedia/folien/Middl

[27] Mehran Azimi, Panos Nasiopoulos, and Rabab K. Ward. Implementation of MPEG System Target Decoder. In *2001 IEEE Canadian Conference on Electrical and Computer Engineering Proceedings*, pages s.943–948, 2001.

[28] QuickTime Homepage at Apple. Web pages, `http://www.apple.com/quicktime/`.

[29] Eric Hoffert, Mark Krueger, Lee Mighdoll, Michael Mills, Jonathan Cohen, Doug Camplejohn, Bruce Leak, Jim Batson, David Van Brink, Dean Blackketter, Michael Arent, Rich Williams, Chris Thorman, Mitch Yawitz, Ken Doyle, and Sean Callahan. QuickTime^{TM}: An Extensible Standard for Digital Multimedia. In *Proceedings of the thirty-seventh international conference on COMPCON*, pages s. 15–20, January 1992.

[30] GStreamer Homepage. Web pages, `http://www.gstreamer.net`.

[31] Nathan J. Muller. Improving and Managing Multimedia Performance Over TCP/IP Nets. *International Journal of Network Management*, Vol.8:s.356–367, 1998.

[32] H. Harlyn Baker, Nina Bhatti, Donald Tanguay, Irwin Sobel, Dan Gelb, Michael E. Goss, John MacCormick, Kei Yuasa, W. Bruce Culbertson, and Thomas Malzbender. Computation and performance issues In coliseum: an immersive videoconferencing system. In Lawrence A. Rowe, Harrick M. Vin, Thomas Plagemann, Prashant J. Shenoy, and John R. Smith, editors, *ACM Multimedia*, pages 470–479. ACM, 2003.

[33] Yantian Lu and Kenneth J. Christensen. Using Selective Discard to Improve Real-Time Video Quality on an Ethernet Local Area Network. *International Journal of Network Management*, Vol.9:s.106–117, 1999.

[34] Larry L. Peterson and Bruce S. Davie. *OSI Architecture*, pages s36–8. In Mann [59], 2000.

[35] Injong Rhee. Error Control Techniques for Interactive Low-Bit Rate Video Transmission over the Internet. In *SIGCOMM*, pages 290–301, 1998.

[36] Color in Image and Video. Web pages, `http://www.cs.sfu.ca/undergrad/CourseMaterials/CMPT479/material/n`

[37] Huffyuv. Web pages, `http://neuron2.net/www.math.berkeley.edu/benrg/huff`

[38] Jose A. Lay and Ling Guan. *The DCT Coefficients in JPEG and MPEG Media*, pages s412–414. In Ling Guan et al. [60], 2001.

[39] L. Berc, W. Fenner, R. Frederick, S. McCanne, and P. Stewart. rfc2435: "RTP Payload Format for JPEG-compressed Video". Web pages, `http://www.networksorcery.com/enp/rfc/rfc2435.txt`, October 1998.

[40] Wolfgang Leister, Svetlana Boudko, Ole Aamot, and Peter Holmes. Digital TV – a survey. Technical report, Norsk Regnesentral, December 2002. Web pages, `http://publications.nr.no/digitv.pdf`.

[41] Larry L. Peterson and Bruce S. Davie. *Video Compression (MPEG)*, pages s548–53. In Mann [59], 2000.

[42] Tsuhan Chen. *H.263*, pages s9–10. In Ling Guan et al. [60], 2001.

[43] Tsuhan Chen. *MPEG-4*, page s11. In Ling Guan et al. [60], 2001.

[44] Roy Fielding et al. rfc2616: "Hypertext Transfer Protocol – HTTP/1.1". Web site, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, June 1999.

[45] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. rfc2543: "SIP: Session Initiation Protocol". Web pages, `http://www.ietf.org/rfc/rfc2543.txt`, March 1999.

[46] V. Balabanian, L. Casey, N. Greene, and C. Adams. An Introduction to Digital Storage Media – Command and Control (DSM-CC). *Web pages*, *`http://drogo.cselt.stet.it/ufv/leonardo/mpeg/documents/dsmcc.html`* 1996.

[47] H. Schulzrinne, A. Rao, and R. Lanphier. rfc2326: "Real Time Streaming Protocol (RTSP)". Web pages, `http://www.normos.org/rfc/rfc2326.txt`, April 1998.

[48] RTSP Resource Center. Web pages, `http://www.rtsp.org/`.

[49] Charles Krasic, Kang Li, and Jonathan Walpole. The Case for Streaming Multimedia with TCP. *Lecture Notes in Computer Science*, 2158:213–218, 2001.

[50] H. Schulzrinne, S. Casner, R. Frederic, and V. Jacobsen. rfc1889: "A Transport Protocol for Real-Time Applications". Web pages, `http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1889.html`, January 1996.

[51] FFmpeg Multimedia System. Web, `http://ffmpeg.sourceforge.net/`.

[52] mplayer - movie player for Linux. Web pages, `http://www.mplayerhq.hu/homepage/`.

[53] LIVE.COM Streaming Media. Web pages, `http://www.live.com/liveMedia/`.

[54] Apple Inc. QuickTime^{TM} Streaming Server modules: QTSS objects. Web pages, `http://developer.apple.com/documentation/QuickTime/QTSS/Concepts/`

[55] Y. Kikuchi, T. Nomura, S. Fukunaga, Y. Matsui, and H. Kimata. rfc3016: "RTP Payload Format for MPEG-4 Audio/Visual Streams". Web pages, `http://www.networksorcery.com/enp/rfc/rfc3016.txt`, November 2000.

[56] Michael Zink, Oliver Künzel, Jens Schmitt, and Ralf Steinmetz. Subjective Impression of Variations in Layer Encoded Videos. In Kevin Jeffay, Ion Stoica, and Klaus Wehrle, editors, *IWQoS*, volume 2707 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 2003.

[57] ITU Telecom. Standardization Sector of ITU. *Methodology for the Subjective Assessment of the Quality of Television Pictures, Recommendation ITU-R BT.500-11*, 2002.

[58] Global Visual Music Homepage. Web pages, `http://www.visualmusic.org/gvm.htm`.

[59] Larry L. Peterson and Bruce S. Davie. *Computer Networks - A Systems Aproach*. Morgan Kaufmann Publishers, 2000.

[60] Ling Guan, Sun-Yuan Kung, and Jan Larsen, editors. *Multimedia image and video processing*. CRC Press LLC, 2001.