BACHELOROPPGAVE - RAPPORT

NTNU
Kunnskap for en bedre verden

TITTEL:
Distribuert modellvisualisering i VR

KANDIDATNUMMER(E):
10035, 10047

DATO:	EMNEKODE:	EMNE:	DOKUMENT TILGANG:
22.01.19	IE303612	Bacheloroppgave	Åpen

STUDIUM:	ANT SIDER/VEDLEGG:	BIBL. NR:
Ingeniør - Data	82 / 5	

VEILEDER(E):
Mikael Tollefsen

Sammendrag:
Målet med denne oppgaven var å utvikle en prototype for RUFO AS der RUFO-representanter og kunder kunne møtes i et virtuelt møterom for å se på og diskutere designet til transportkasser i et 1:1 forhold, før kassene ble produsert. Applikasjonen skulle ha støtte for både stemmekommunikasjon og importering av modeller. Modellene er digitale representasjoner av transportkassene RUFO skreddersyr til sine kunder, og skal kunne bli manipulert i form av bevegelse, rotering, skalering og "eksplodering". Eksplodering i denne konteksten betyr å utvide og dekonstruere modellen, slik at hver enkelt del kan bli inspisert nærmere av kunden. Møterommet skal i tillegg være distribuert, som vil si at tilstanden til modellene skal være synkronisert mellom alle tilkoblede brukere i sanntid.

Vi valgte å løse oppgaven ved å bruke Unity3D, som er en populær grafikkmotor for VR-applikasjoner. Nettverksfunksjonalitet ble implementert via en kombinasjon av servertjenesten

Postadresse	Besøksadresse	Telefon	Telefax	Bankkonto
NRNU i Ålesund	Larsgårdsvegen 2	70 16 12 00	70 16 13 00	7694 05 00636
N-6025 Ålesund	Internett	Epostadresse		Foretaksregisteret
Norway	www.hials.no	postmottak@hials.no		NO 971 572 140

Photon og vår egen vert-til-vert filoverføringsløsning. Interaksjon med menyer ble implementert ved hjelp av Oculus sitt VR-bibliotek til Unity3D, og Oculus-kontrollerne.

Resultatet ble et robust produkt vi mener skal være lett å utvide takket være riktig bruk av komponentsystemet til Unity og ansvarlig bruk av designmønstre. Løsningen oppfylte alle kravene til oppdragsgiver med minimale restriksjoner; og vi lærte mye om utvikling i 3D miljøer, komponentbaserte systemer og om generell nettverksfunksjonalitet under prosjektets utvikling.

INNHOLD

FORPROSJEKT - RAPPORT

FOR BACHELOROPPGAVE

NTNU
Kunnskap for en bedre verden

SAMMENDRAG

Målet med denne oppgaven var å utvikle en prototype for RUFO AS der RUFO-representanter og kunder kunne møtes i et virtuelt møterom for å se på og diskutere designet til transportkasser i et 1:1 forhold, før kassene ble produsert. Applikasjonen skulle ha støtte for både stemmekommunikasjon og importering av modeller. Modellene er digitale representasjoner av transportkassene RUFO skreddersyr til sine kunder, og skal kunne bli manipulert i form av bevegelse, rotering, skalering og "eksplodering". Eksplodering i denne konteksten betyr å utvide og dekonstruere modellen, slik at hver enkelt del kan bli inspisert nærmere av kunden. Møterommet skal i tillegg være distribuert, som vil si at tilstanden til modellene skal være synkronisert mellom alle tilkoblede brukere i sanntid.

Vi valgte å løse oppgaven ved å bruke Unity3D, som er en populær grafikkmotor for VR-applikasjoner. Nettverksfunksjonalitet ble implementert via en kombinasjon av servertjenesten Photon og vår egen vert-til-vert filoverføringsløsning. Interaksjon med menyer ble implementert ved hjelp av Oculus sitt VR-bibliotek til Unity3D, og Oculus-kontrollerne.

Resultatet ble et robust produkt vi mener skal være lett å utvide takket være riktig bruk av komponentsystemet til Unity og ansvarlig bruk av designmønstre. Løsningen oppfylte alle kravene til oppdragsgiver med minimale restriksjoner; og vi lærte mye om utvikling i 3D miljøer, komponentbaserte systemer og om generell nettverksfunksjonalitet under prosjektets utvikling.

TERMINOLOGI

Begreper

PC	Personal Computer, personlig datamaskin.
Sprint	En fast, kortvarig periode (ofte 1-4 uker) brukt i smidige metoder, der utviklerne jobber med å implementere funksjonalitet fra sprintreserven.
Spagettikode	Ett nedsettende uttrykk som brukes om tett sammenkoblet kode som er vanskelig å endre eller utvide. Sier ingenting om hvor optimalisert kode er.
Navnerom	Eng: namespace, et sett med symboler brukt for å organisere forskjellige objekter, slik at de kan bli referert til ved navn.

Postadresse	**Besøksadresse**	**Telefon**	**Telefax**	**Bankkonto**
Høgskolen i Ålesund	Larsgårdsvegen 2	70 16 12 00	70 16 13 00	7694 05 00636
N-6025 Ålesund	**Internett**	**Epostadresse**		**Foretaksregisteret**
Norway	www.hials.no	postmottak@hials.no		NO 971 572 140

Mesterklient	Den autoritære klienten i Photon rommet. Klienten som kobler til rommet først blir mesterklient. Denne klienten fungerer som en filserver i tillegg til en vanlig klient og har kontroll over hvilke modellfiler som kan bli lastet inn.
Frysing	Når en applikasjon slutter å oppdatere seg grafisk sier vi at den fryser.
Scene	Den grafiske verdenen.
Tråd	En underprosess skapt av hovedprosessen. Den deler minneområde med hovedprosessen, men oppfører seg ellers som en egen prosess.
Trådbasseng	En samling av tråder som blir skapt når programmet åpnes. Trådbassenger blir brukt for å hindre at programmet skaper for mange tråder og dermed reduserer ytelsen til datamaskinen.
Strålesjekk	En kollisjonsdeteksjonsmetode som forteller om en stråle med gitt start- og sluttpunkt krysser én eller flere kollisjonsrammer, og gir eventuell informasjon om de kryssende objektene. Dette blir gjort i Unity3D sitt fysikksystem.
Pakke	Nyttelast representert i binær form som er sendt over nettverket.
Socket	Et endepunkt brukt for kommunikasjon over internettet. Også kalt internettsocket eller nettverksocket.
Null (nullpeker)	En peker som representerer/refererer til ingenting eller et ugyldig objekt.
Dødsone	Begrep ofte brukt om styrespaker. Styrespaken kan dras og snus på for å signalisere en retning i to akser. Om styrespaken står stille/oppreist, skal inndataen vise 0. Siden hardware ikke er perfekt, bruker vi en dødsone som sier at inndataen er 0 om styrespak posisjonen er innenfor en viss sone: dødsonen.
Avatar	En avatar er en grafisk representasjon av en bruker.

Forkortelser

VR – Virtual Reality

FPS – "Frames per second" på engelsk. Antall bilder som blir vist per sekund.

RPC – Remote Procedure Calls.

PUN – Photon Unity Networking. Nettverksmodulen for å håndtere nettverkstrafikk for å synkronisere klienter.

GPU – Graphical Processing Unit

CPU – Central Processing Unit

RAM – Minnet til datamaskinen. (Eng: Random Access Memory)

Formler

Formel 1: veiformelen $s = v_0 t + \frac{1}{2} a t^2$

1 INNLEDNING

1.1 Prosjekt Beskrivelse

1.1.1 Bakgrunn

Bedriften RUFO er et firma som i over 30 år har spesialisert seg på å lage transportkasser for aktører i lyd-, lys- og bildebransjen. De tilbyr standardmodeller og skreddersydde kofferter, transportkasser og stativer tilpasset til hver kundes behov[33].

RUFO forteller at 9 av 10 kunder er fornøyde etter produksjon og levering av produktet. RUFO antar at de kundene som er utilfreds kommer av misforståelser eller feil som kunden ikke fant gjennom bedriftens nåværende løsning for bestilling av produkt; en 3D-visualisering på nettet.

RUFO ønsker å hindre feilbestillinger og å bedre tilfredsstille kundene sine ved å vise frem produktene på en måte som oppleves mer naturlig for kunden. Derfor ba de oss lage et VR-møterom hvor de kan vise frem kassene til kunder i et 1:1 forhold og diskutere eventuelle endringer i bestillingen før produksjon blir satt i gang.

Denne oppgaven fokuserer på å skape et velfungerende produkt/prototype, som vil oppfylle oppdragsgiver sine mål og ønsker.

1.1.2 Mål

1.1.2.1 Kravspesifikasjon

- Raskt og responsivt sluttprodukt. (~90 FPS er vanlig for VR applikasjoner)
- Produktet skal være kompatibelt med Oculus Quest, et produkt som ikke har blitt lansert enda. VR-brillene er Android-baserte, og trenger ikke en datamaskin for å fungere.
- Virtuelt møterom man kobler til over nettet. Rommet er ment for 2-4 møtedeltagere, men støtter flere.
- Møtedeltagere skal kunne bevege seg rundt i rommet.
- Hver møtedeltager skal være synlig for andre møtedeltagere i møterommet.
- Hver deltager skal kunne skrive inn et kallenavn. Kallenavnene blir brukt i møterommet for å identifisere hver deltager.
- Støtte for stemmekommunikasjon i møterommet.
- Man skal kunne vise frem importerte modeller i sanntid. Kun en klient skal behøve å ha filene på forhånd for å kunne vise dem frem til resten av deltagerne.
- Modellene skal vises i et tilnærmet 1:1 forhold, mellom virkeligheten og løsningens visualisering.
- Man skal kunne flytte på, rotere og skalere importerte modeller. Disse operasjonene skal reflekteres hos alle møtedeltagere.
- Man skal kunne "eksplodere" en modell for å kunne se nærmere på hver del av modellen.
- Brukergrensesnittet skal være skrevet på engelsk.

1.1.2.2 Personlige mål

- Få erfaring i å planlegge og produsere IT-prosjekter.
- Bli bedre kjent med C#, Unity3D og VR, samt hvordan de henger sammen.

- Bli bedre kjent med nettverkssynkronisering med Photon og modulens konsekvenser over prosjektet.
- Utvide vår kunnskap om god arkitektur og kodestruktur.

2 TEORETISK GRUNNLAG

2.1 Smidige metoder for programvareutvikling

Ifølge denne studien[29], koster store IT-prosjekt ofte betydelig mer enn planlagt. Det kan gå så langt at det setter hele organisasjonen i fare. Det ble regnet ut at i gjennomsnitt overskrider 45% av store IT-prosjekter (med budsjett over $15 millioner) budsjettene sine, 7% går over tidsrammene og 56% leverer mindre verdi enn antatt.

Grunnen til at så mange IT-prosjekter har store problemer under utviklingsprosessene sine, er mangfoldig[30]:

- Fossefallsmetoden som fungerer for andre typer prosjektarbeid, ikke fungerer like bra for IT-prosjekter.
- IT-industrien mangler standarder, som betyr at det ikke finnes en avtalt måte for hvordan man skal utvikle programvare. I tillegg finnes det godt over 500 forskjellige programmeringsspråk som for det meste er inkompatible med hverandre.
- Operativsystemer og plattformer er inkompatible med hverandre. For eksempel er programvare utviklet for Windows ikke kjørbar på OSX, og Android-apper ukjørbare på IOS.
- Klient/produkteier tror de vet hva de ønsker, men ofte forstår de ikke hva som er lett, og hva som hadde tatt et dedikert lag med forskere 5 år å lage. Alle vet at en bru vil kollapse uten et fundament. Det samme er sant for programvare, men er ikke like lett å forstå uten en dypere kunnskap om programmering og systemarkitektur.

Dette gjør kostnads- og tidsestimater for IT-prosjekter meget vanskelig.

For å kunne forbedre noen av disse estimatene kan utviklingsorganisasjoner gå over til å bruke smidige (eng: Agile) metoder. Smidige metoder er et samlebegrep for flere forskjellige utviklingsmetodologier som Scrum, Extreme Programming (XP) og Kanban. De ble utviklet som en respons til at de klassiske utviklingsmetodene var så upålitelige. I stedet for å ha en rigid plan med klare sektorer for hva som skal skje og når, fokuserer smidige metoder på å kunne reagere raskt når endringer oppstår under utviklingen. Smidige metoder setter også kunden i en mer sentral stilling enn klassiske utviklingsmetoder, dette er for å lettere overkomme kommunikasjonsproblemer og for å sikre seg at kunder blir fornøyde.

"Agile" manifestet[31] beskriver hovedprinsippene og filosofien bak smidige metoder:

- Individer og interaksjoner over prosesser og verktøy.
- Fungerende programvare over omfattende dokumentasjon.
- Kundesamarbeid over kontrakt forhandling.
- Reagere på endringer over å følge en plan.

Det vil si at selv om det er verdi i prinsippene til høyre, verdsetter smidige-metoder-prinsippene til venstre mer.

2.1.1 Scrum

Scrum er en smidig arbeidsmetode først presentert i 1995 av Jeff Sutherland og Ken Schwaber[30]. Det er et rammeverk for å utvikle, levere og vedlikeholde komplekse produkter[32]. Scrum er bygget opp av empirisk kontrollteori eller empirisme. Filosofien dikterer at kunnskap kommer av erfaring, der man gjør valg basert på hva man vet. Scrum bruker en iterativ, trinnvis modell for å forbedre forutsigbarheten til prosjektet og å minimisere risiko.

I praksis består scrum-rammeverket av gitte roller, artefakter og møter:

- Roller
    - **Produkteier** – personen som er ansvarlig for å maksimere verdien av produktet utviklingslaget produserer. De har full autoritet over produktreserven (eng: product backlog), som beskriver hvilke funksjoner/bruksområder som skal prioriteres for utviklingslaget. De kan ikke si hvordan funksjonene skal bli implementert, bare når.

      Utviklingslaget – består av profesjonelle utviklere som jobber med funksjonene listet opp i sprintreserven under hver sprint. Utviklingslaget er selvorganisert, så de bestemmer selv hvordan de skal utvikle funksjonene produkteier ønsker, og hvem som er ansvarlig for en gitt oppgave under hver sprint. Utviklingslaget har verken prosjektleder eller et hierarki.

      Størrelsen på utviklingslaget ligger ideelt rundt 3-9 personer. Mindre enn 3 utviklere kan føre til en mindre produktivitetsgevinst og de kan ha loproblemer med manglende ferdigheter under sprinten. Om man har flere enn 9 utviklere behøver man for mye koordinasjon til at en empirisk utviklingsfilosofi kan være nyttig.
    - **Scrum-mester** – person som observerer og passer på at scrumlaget opprettholder en scrumbasert arbeidsflyt. Dette blir gjort gjennom å hjelpe de andre medlemmene med å forstå scrumteori, -prinsipper, -regler og -verdier.
- Artefakter
    - **Produktreserven –** en prioritert liste med elementer som beskriver hva utviklingslaget skal fokusere på. Dette inkluderer funksjonalitet (oppgitt som brukerhistorier/bruksområder), fiksing av feil, forbedringer og redesigning av eksisterende systemer. Eies av produkteieren, og endres ofte gjennom prosjektets utvikling. Den beskriver hva som skal gjøres, ikke hvordan.
    - **Sprintreserven –** delsettet av produktreserven som beskriver hva utviklerne skal gjøre i en gitt sprint. Elementene blir til sprintoppgaver som beskriver hvordan det skal implementeres. Utviklingslaget bestemmer antallet element som blir med, men prioriteringene må opprettholdes.
- Møter
    - **Sprintplanleggingsmøte –** Møte mellom utviklingslaget og produkteier der det blir bestemt hvilke elementer fra produktreserven som skal bli gjort i den kommende sprinten. Elementene vil bli delt opp i oppgaver som blir lagt til i

sprintreserven. Hver oppgave får tildelt en poengverdi som er et estimat på hvor vanskelig den gitte oppgaven blir å implementere for utviklingslaget.

- **Daglig scrum –** et kort møte på rundt 15 min som blir gjort hver dag under sprinten, av utviklingslaget. Her vil utviklerne planlegge den påløpende dagen, og diskutere hva som ble gjort dagen før, hva som ikke ble gjort og problem som støtes på.
- **Sprintgjennomgang –** uformelt møte som blir holdt på slutten av sprinten der produkteier og kunden kan inspisere det trinnvise produktinkrementet og oppdatere produktreserven om nødvendig. Her diskuterer prosjektorganisasjonen hva som ble gjort under sprinten, og hvilke muligheter som kan gjøres for å optimalisere verdien av produktet.
- **Sprint retrospektiv –** Møte som foregår etter sprintgjennomgangen, der scrum laget kan inspisere den forrige sprinten og diskutere hva som gikk bra og hva som gikk mindre bra, for å planlegge forbedringer til neste sprint.

2.2 Generell kodedesign og -struktur

Designprinsipper er retningslinjer for å skrive kode som er lett å forstå og modifisere[4]. Designprinsipper sier ingenting om hva en klasse skal gjøre, men heller hvordan et hierarki av klasser skal konstrueres for å unngå spagettikode. Spagettikode er et nedsettende begrep for klasser og kode som er tett sammenkoblet. Tett kobling gjør klasser mer avhengige av hverandre, som gjør de vanskeligere å endre senere uten å påvirke andre klasser. Dette medfører at man bruker mere tid på å fikse systemarkitekturen, som heller kunne blitt brukt til å utvide den.

2.2.1 Kodestil og -kvalitet

Å bruke et sett med konsistente kodekonvensjoner kan hjelpe med utviklingen av programvaren/systemet. Det gjør det lettere å samarbeide med andre som jobber i prosjektet, da både formen på koden, navngivning og strukturen til koden følger de samme reglene gjennom hele prosjektet. Gjennom dette prosjektet brukte vi hovedsakelig Microsofts kodestil for C#[2] og forholdte oss til de fleste av deres retningslinjer[3]. Kodekommentarer var et felt der vi ikke fulgte Microsofts kodestil, istedenfor har begge gruppemedlemmene brukt kommentarstil lik det som er anbefalt for Java-kommentarer i Netbeans IDE. For å danne dokumentasjon gjennom Doxygen var kommentarer anbefalt å bruke, spesielt for komplekse klasser og strukturer. Alle klasser og metoder er kommentert for å oppnå best mulig lesbarhet både i koden og i den genererte dokumentasjonen.

2.2.2 Ansvarsdrevet design [5]

Ansvarsdrevet design handler om å designe klasser med veldefinerte unike arbeidsoppgaver. Den vanligste måten å oppnå dette er å analysere applikasjonens behov, for så å avgjøre hvilke klasser man trenger, og hvilke arbeidsoppgaver de skal ha. Når man har kommet frem til et bra design kan man begynne å implementere klassene. Mens man planlegger klassehierarkiet er det viktig å tenke på de viktigste designprinsippene i ansvarsdrevet design, nemlig lav kobling og høy kohesjon[4].

Kobling i kodesammenheng beskriver hvor sammenkoblede klassene i programmet er. Jo mer avhengig en klasse er av en eller flere andre klasser, jo høyere kobling.

Man ønsker lav kobling i programmer, slik at man lett kan vedlikeholde og utvide klasser uten at det skal påvirke andre klasser i applikasjonen. Et viktig designprinsipp for å oppnå lav kobling er innkapsling av datastrukturer og hvordan de aksesseres av andre klasser. Metoder som blir brukt for kommunikasjon mellom klasser burde være små. De skal kun gjøre én ting, og samtidig være så abstrakte som mulig. Et eksempel er å ha en aksessmetode for et internt objekt som returnerer en supertype av objektet. Om man senere bytter det interne objektet med en annen subtype, behøver man kun å endre aksessmetoden, der man slipper å måtte oppdatere andre klasser som bruker den.

Kohesjon i kodesammenheng betyr at hver klasse skal representere en veldefinert entitet i et veldefinert bruksområde. Det vil si at klassen har én jobb og holder seg til funksjonalitet som omhandler denne. Det samme gjelder for metodekohesjon, der hver metode skal være ansvarlig for én veldefinert oppgave. Dette gjør det lettere å unngå kodeduplikasjon, siden all funksjonalitet relatert til en gitt oppgave ligger kun ett sted i applikasjonen.

2.2.3 Arv og polymorfi

For å redusere mengden kodeduplikasjon, spesielt blant klasser der oppgavene er like, kan man bruke arv[4]. Arv er et kjerneprinsipp i objektorienterte programmeringsspråk som Java og C#. Det gjør at man kan utvide klasser basert på en superklasse/baseklasse (Java/C#) der underklassene arver alle felt og metoder fra superklassen som ikke er private. Det lar oss lage større superklasser som inneholder delt funksjonalitet fra klasser som arver fra dem, uten å måtte duplisere kode.

Et annet positivt aspekt med arv og objektmodellen er polymorfi[4]. I objektorienterte språk kan variabler deklarert med en viss type holde på objekter av den deklarerte typens subtyper. Dette gjør at om man kaller en metode på denne variabelen, kan andre metoder bli kalt i stedet for den deklarerte typens implementasjon. Metoden som faktisk blir kalt, kommer fra den dynamiske typen til objektet. Det er mulig fordi subtyper kan overstyre supertypens implementasjon av den gitte metoden. Denne funksjonaliteten gjør design som tilstandsmønstre mulig.

2.3 Interaktivt design

Interaktivt design handler om å lage engasjerende brukergrensesnitt som er intuitive og lette å lære seg[33]. For å kunne oppnå dette må man forstå hvordan teknologi og brukere kommuniserer med hverandre. Dette gjør det lettere å forutse hvordan brukere vil samhandle med systemet, slik at man kan fikse problemer raskt.

Ting man må ta i betraktning når man utvikler interaktive design:

Definer hvordan brukere kan bruke brukergrensesnittet

- Tenk på hvordan brukere kan kommunisere med brukergrensesnittet gjennom styringsinstrumentene de har tilgjengelig.
- Tenk på handlinger brukeren forventer å kunne bruke, som ikke direkte er en del av produktspesifikasjonen. For eksempel, en bruker kan forvente å kopiere innhold ved å trykke Ctrl + C på tastaturet.

Gi brukeren hint om hva en handling vil gjøre, før den blir tatt

- Bruk utseende (farge, form, størrelse o.l.) til å signalisere til brukeren hvordan brukergrensesnittet fungerer og hvordan det skal brukes.
- Presenter nok informasjon til å signalisere til brukeren hva som vil skje om de gjør en gitt handling.

Forutse og reduser mulige feil

- Design rammer og begrensninger som reduserer og forebygger feil som brukeren kan komme til å gjøre.
- Eventuelle feilmeldinger som vises til brukeren skal inneholde nok informasjon slik at de kan fikse problemet selv, eller skal kunne forstå hvorfor feilen skjedde.

Tenk på responssystemer og responstid

- Gi brukeren respons når en handling i applikasjonen blir gjort. Dette gjøres for å signalisere til brukeren at handlingen deres har blitt registrert av systemet.
- Reduser tiden mellom en brukers handling til produktets respons. Produktets reaksjonsevne kan kategoriseres inn i fire nivå: umiddelbar (0-0.1 sek), hakking (0.1-1 sek), avbrudd (1-10 sek) og forstyrrelser (+10 sek).

Tenk strategisk på hvert element

- Bruk fornuftige størrelser for brukergrensesnittene slik at de er enkle å bruke.
- Bruk kanter og hjørner strategisk for interaktive element som menyer og/eller knapper. Fitts lov[38] forteller oss også at tiden det tar å velge et element i et brukergrensesnitt kan formuleres som funksjonen av avstanden(D) til elementet delt på størrelsen(W): $ID = \frac{2D}{W}$.

Fokuser på lærbarhet

- Porsjoner informasjon som blir vist til brukeren inn i 7 ± 2 elementer av gangen. Ifølge George Miller[35] sliter korttidshukommelsen med å huske mer enn 5-9 elementer av gangen.
- Forenkle brukergrensesnittet så mye som mulig. Ifølge Larry Tezzler skal man redusere applikasjonens kompleksitet slik at brukeren lærer programmet raskere[36]. Han nevner også at en gitt handling eller funksjon kun kan bli forenklet til en viss grad, før de mister sin hensikt.
- Bruk kjente formater og designvalg. Hick's lov[37] forteller at beslutningstid er en kombinasjon av hvor velkjent designet i applikasjonen er til brukeren, om brukeren er kjent med valgene som kan tas i applikasjonen, og antall valg brukeren må velge mellom.

2.4 Designmønstre

Designmønstre (eng: design patterns) beskriver generelle problemer som blir møtt i mange forskjellige programmer, samt generelle løsninger på dem[5]. Designmønstre har fire kjennetegn:

1. **Navnet**: Navnet i et designmønster kan beskrive problemet, løsningen til problemet og/eller konsekvenser av designprinsippet i noen få ord.

2. **Problemet**: Problemet forteller oss om bruksområdet til designmønsteret. Det kan fortelle om konteksten til problemet, spesifikke implementasjonsdetaljer og andre vilkår for at løsningen skal gi mening.
3. **Løsningen**: Løsningen beskriver hovedprinsippene som utgjør designmønsteret, hvordan de henger sammen, og deres ansvar.
4. **Konsekvensene**: De positive og negative aspektene ved å bruke designmønsteret. Dette kan innebære størrelse, tid eller problemer i spesifikke språk og/eller implementasjoner.

Designmønstre gjør det lettere å gjenbruke design og arkitekturer som har vist seg å være vellykkede tidligere. Ved å utrykke utprøvde teknikker som designmønstre gjør man dem lettere å implementere i nye system. Designmønstre gjør det lettere å velge designalternativer som gjør systemet gjenbrukbart, og å unngå alternativer med motsatt effekt. Designmønstre kan også forbedre dokumentasjonen og gjøre utvikling av systemet enklere ved å definere eksplisitte spesifikasjoner for hvordan klasser og objekter kommuniserer, samt hensikten bak dem. Designmønstre gjør det både lettere og raskere å oppnå et gunstig design.

2.4.1 Singletonmønsteret [5]

Singletonmønsteret forsikrer oss om at det aldri eksisterer mer enn én instans av en klasse som kan aksesseres via et globalt tilgangspunkt. For enkelte klasser er det viktig at det bare finnes nøyaktig et objekt av dem. Typisk er dette administrative klasser som kontrollerer objekter og funksjoner i systemet. Ofte er singletonklassen selv ansvarlig for oppretting av sin egen instans for å forhindre feilaktige instanser.

Singletonmønsteret kan brukes når:

- Det må eksistere nøyaktig én instans av en gitt klasse, og den må være tilgjengelig i flere klasser som ikke nødvendigvis er sammenkoblet.
- Den eneste instansen skal bli utvidet av subklasser, og klienter skal kunne bruke den utvidede instansen uten å endre sin egen kode.

2.4.1.1 Interaksjoner mellom klassene

Klientene aksesserer den unike instansen kun gjennom singletonklassens tilgangspunkt.

2.4.1.2 Konsekvenser av å bruke singletonmønsteret

1. Kontrollert tilgang til den eneste instansen. Siden singletonklassen innkapsler sin egen unike instans, kan man opprettholde stor kontroll over hvordan klientene bruker instansen.
2. Redusert sløsing av navnerom (eng: name space). Singletonmønsteret er en forbedring fra globale variabler, der man unngår å forurense/blåse opp navnerommet til klassen.
3. Tillater endring og utvidelse av singletonens oppgaver og representasjon gjennom arv, der det er enkelt å konfigurere en applikasjon med en instans av denne utvidede subklassen. Konfigurasjon kan skje i sanntid med en instans av den klassen man trenger.

4. Tillater flere instanser om nødvendig. Om man finner ut at man trenger flere instanser av singletonklassen er det enkelt å tillate, der tilgangspunktet er den eneste operasjonen som må endres.
5. Mer fleksibilitet enn ved å bruke klasseoperasjoner. I stedet for å bruke en singleton kan man innkapsle dens funksjonalitet i globale klasseoperasjoner (statiske metoder og variabler). Dette gjør det vanskelig å skape flere instanser av klassen, samt umulig å bruke arv.

2.4.2 Tilstandsmønsteret (eng: state pattern) [5]

Tilstandsmønsteret gjør det mulig for et objekt å endre sin interne adferd når objektets interne tilstand endrer seg.

Dette designmønsteret kan bli brukt når:

- Et objekts adferd er avhengig av objektets tilstand, og adferden må oppdateres i sanntid.
- Operasjoner har store, flerpartsbetingelser som avhenger av objektets interne tilstand, gjerne representert i form av en eller flere forhåndsdefinerte konstanter.

I dette designet plasserer man hver del av betingelsene i en separat klasse. Dette gjør at man kan referere til objektets tilstand gjennom objekter som kan variere betydelig fra hverandre.

Tilstandsmønsteret trenger tre klasser:

- Kontekst
  – Definerer grensesnittet til brukere av tilstandene.
  – Holder på en instans av tilstandsklassens undertyper som definerer klassens nåværende tilstand.
- Tilstand
  – Definerer et grensesnitt for innkapsling av spesifikk funksjonalitet som er tilknyttet en gitt tilstand i kontekst klassen.
- Konkrete Tilstandsunderklasser
  – Hver underklasse implementerer funksjonalitet tilknyttet den spesifikke tilstanden i kontekstklassen.

2.4.2.1 Interaksjoner mellom klassene

- Kontekstklassen delegerer tilstandsrelaterte ressurser til den aktive tilstanden.
- En kontekst kan bruke seg selv som et argument til tilstandsklassen som håndterer ressursforespørselen, slik at tilstandsobjektet kan referere og bruke kontekstobjektet om nødvendig.
- Kontekstklassen er det primære grensesnittet til eksterne klasser. De kan konfigurere kontekstklassen ved å endre tilstanden, direkte eller indirekte. Når konteksten er satt, behøver ikke eksterne klasser å bry seg om de interne tilstandene direkte.

2.4.2.2 Konsekvenser av å bruke tilstandsmønstret

1. Det lokaliserer og deler opp funksjonalitet mellom forskjellige tilstander. All funksjonalitet tilknyttet en spesifikk tilstand ligger i et objekt. Dette gjør utvidelse av

nye tilstander lettere å implementere gjennom å danne nye tilstandssubklasser. Gjennom å bruke tilstandsmønsteret vil man ha et større antall klasser som gjør løsningen mindre kompakt, men det er fortsatt en fordel om man blir kvitt store betingelsestrær. Man unngår monolittiske klasser og gjør koden og systemet mer eksplisitt og leselig, som igjen gjør det lettere å utvide og modifisere senere.

2. Det gjør tilstandsendringer eksplisitte. Når objekter definerer sin tilstand bare gjennom interne dataverdier, har ikke tilstandsovergangene noen eksplisitt representasjon i koden. Tilstandsmønsteret har som konsekvens at klassen ikke befinner seg i en inkonsistent eller invalid tilstand fordi tilstandsovergangene er atomiske: hver tilstandsovergang omhandler endringer for *én* variabel (kontekstens tilstandsvariabel), ikke flere.
3. Tilstandsobjekter kan deles. Hvis tilstandsobjektene ikke inneholder egne variabler, der tilstanden de representerer er forstått kun gjennom subtypen, kan tilstandsobjektene deles mellom flere kontekstobjekter.

2.4.3 Observatørmønsteret (eng: observer pattern) [5]

Observatørmønsteret definerer en «en-til-mange» avhengighet mellom objekter slik at når et objekt endrer tilstand, vil avhengighetene bli oppdatert automatisk. Mønsteret baserer seg hovedsakelig på to parter: subjektet og observatøren. Subjektet kan ha flere forskjellige avhengige observatører, som vil bli varslet når subjektet endrer tilstand; da vil observatørene synkronisere sin tilstand i respons til subjektets tilstand. Denne typen kommunikasjon er også kjent som "utgiver-abonnent strategien". Subjektet er produsent/utgiver og observatørene er abonnenter, der subjektene ikke trenger å vite om sine abonnenter.

Observatørmønsteret kan bli brukt:

- Når en abstraksjon har to aspekter, det ene avhengig av det andre. Ved å innkapsle disse aspektene i forskjellige objekter gjør man det mulig å variere og gjenbruke dem individuelt.
- Når en endring i et objekt krever oppdatering hos andre, og antallet objekter som må endres er usikkert.
- Når et objekt behøver muligheten til å varsle andre objekter uten å vite hvem objektene er.

Når man implementerer observatørmønsteret trenger man fire klasser:

- Subjekt
  - Vet om sine abonnenter. Kan bli observert av observatører.
  - Har et grensesnitt for å feste og løsne observatørobjekter.
- Observatør
  - Definerer et oppdateringsgrensesnitt for objekter som ønsker å abonnere på endringer i subjektene.
- Konkret Subjekt
  - Oppbevarer en tilstand som observatørobjektene er interessert i.
  - Varsler sine observatører når sin egen tilstand endres.
- Konkret Observatør
  - Har en referanse til et konkret "Subjekt" objekt.

- Holder på en tilstand som skal synkroniseres med subjektets tilstand.
- Implementerer observatørens oppdateringsgrensesnitt for å holde seg oppdatert på subjektets tilstand.

2.4.3.1 Interaksjoner i observatørmønsteret

- Den konkrete "Subjekt" klassen varsler sine observatører når dens interne tilstand endres på en slik måte at observatørenes tilstander ikke blir inkonsistente med dens egen tilstand.
- Etter at en endring har blitt rapportert i det konkrete subjektet, kan observatøren referere tilbake til subjektet for mer informasjon.

2.4.3.2 Konsekvenser av å bruke observatørmønsteret

Mønsteret gjør det mulig å modifisere subjekter og observatører uavhengig av hverandre. Man kan gjenbruke subjekter med nye observatører, og omvendt.

1. Det abstraherer koblingen mellom subjekter og observatører. Alt subjektene vet er at de har en liste med observatører som samsvarer med grensesnittet gitt ved den abstrakte observatørklassen.
   Fordi subjektet og observatøren ikke er tett sammenkoblet, kan de tilhøre forskjellige abstraksjonslag i systemet. Her kan for eksempel et lavt-nivå objekt kommunisere med og informere et objekt på et høyere abstraksjonsnivå.
2. Det gjør kringkastingskommunikasjon mulig. I motsetning til ordinære forespørsler, blir ikke mottakere spesifisert ved kringkastingskommunikasjon. Informasjonen blir kringkastet automatisk til alle objekter som abonnerer. Subjektet bryr seg ikke om hvor mange abonnenter som eksisterer. Alt den trenger å gjøre er å varsle dem. Dette gjør at man kan legge til og trekke fra observatører når som helst. Det er opp til observatøren om den vil håndtere eller ignorere varslingen.
3. Uventede oppdateringer kan oppstå. Observatørene har ingen kunnskap om hverandre, så det er vanskelig å estimere hvor mange maskinressurser som blir brukt til å endre tilstanden til subjektet. En tilsynelatende enkel og ufarlig operasjon kan forårsake en rekke unødvendige oppdateringer hos resten av observatørene og deres avhengige objekter. Om avhengighetskriteriene ikke er veldefinerte og/eller vedlikeholdt kan de ofte forårsake falske oppdateringer, som kan være vanskelig å løse. Dette problemet er alvorlig fordi en oppdatering ikke nødvendigvis kan spesifisere hva ett subjekt har endret. Det sistnevnte problemet kan bli løst via en ekstern protokoll som viser observatørene nøyaktig hva som har endret seg.

2.4.4 Kompositt-/komponentmønsteret [5]

Med komponentmønsteret samler man objekter i en trestruktur for å representere hierarkier. Her kan man håndtere individuelle objekter og samlinger av objekter uniformt.

Hovedsakelig vil man bruke komponentmønsteret:

- Når man ønsker å representere helhetlige objekthierarkier som ett objekt.
- Når man ønsker at klientobjektene skal kunne ignorere forskjellen mellom oppbygningen til individuelle objekter. Altså, at klientene skal håndtere alle objekter i den komponentbaserte strukturen uniformt.

I komponentmønsteret har man typisk fire klasser:

- Komponent baseklassen
    - Fungerer som et grensesnitt for objektene i oppbygningen.
    - Implementerer standardmetoder som er delt mellom alle bladene.
    - Spesifiserer et grensesnitt for å håndtere barn-/underkomponenter.
    - *Kan* definere et grensesnitt for å få tilgang til forelder-/superkomponenten.
- Bladet
    - Representerer blad-/løvobjektene i oppbygningen. De har ingen barn/underkomponenter.
    - Definerer funksjonalitet for primitive objekter i oppbygningen.
    - Man fester vanligvis flere blad til hver kompositt for å gi en kompositt ønsket funksjonalitet.
- Kompositt
    - Grensesnitt for komponenter som har underkomponenter.
    - Inneholder en samling som kan holde på underkomponenter.
    - Implementerer en underkomponents relative funksjonalitet i komponentgrensesnittet.
- Klient
    - Bruker og samhandler med objektene i hierarkiet gjennom Komponentgrensesnittet.

2.4.4.1 Interaksjoner i Komponentmønsteret

Klientene bruker Komponentgrensesnittet for å samhandle med og manipulere objektene i oppbygningen. Om komponenten er et blad, vil forespørselen bli behandlet direkte. Om komponenten er en kompositt, vil forespørselen bli sendt til objektets underkomponenter. Det er også mulig å ha funksjonalitet på kompositttobjektet selv, som kan kjøre før og/eller etter at underkomponentene er ferdige med forespørselen.

2.4.4.2 Konsekvenser av å bruke komponentmønsteret

1. Det definerer klassehierarkier organisert av primitive- og oppbygningsobjekter. Primitive objekter kan bli komponert til mere komplekse objekter, som igjen kan bli komponert osv. Når en klient forventer en primitiv, kan vi gi den et kompositttobjekt istedenfor.
2. Klientene blir enklere. Siden de kan behandle både primitive og kompositttobjekter uniformt, kan vi forenkle koden.
3. Det gjør det lett å legge til nye komponenter. Nye definerte løv- og kompositttobjekter vil fungere automatisk med det eksisterende systemet. Klienter behøver ikke å endre sin driftskode for nye komponenter.

4. Det kan gjøre designet i systemet for generelt. Når det er lett å legge til nye komponenter blir det vanskelig å avgrense en kompositts underkomponenter. Typer kan ikke bli brukt til å forsikre seg om at et kompositttobjekt kun tar imot gyldige underkomponenter før man kjører programmet.

2.5 Arkitektur i Unity3D

Når man utvikler i en tilrettelagt spillmotor vil den påvirke kodearkitekturen grundig fra starten av. Derfor er det viktig at man forstår arkitekturen i Unity3D for å bedre forstå denne applikasjonen sin arkitektur og løsninger.

Unity3D er en spillmotor utviklet av Unity Technologies som kan kjøre på mange populære plattformer. Unity er lagd for å lage både spill og simuleringer i 2 eller 3 dimensjoner. Unity har også innebygd støtte for VR.

Unity3D er en komponentbasert spillmotor[7]. Den bruker *Scener* som representerer verdener/nivåer. Alle objekter i en scene er av typen *GameObject*. De fungerer som en hovedklasse som man kan tilføye egne skripter kalt komponenter. Komponentene må arve fra *MonoBehaviour* klassen for at Unity skal kunne bruke og håndtere skriptene. Barn av GameObject kan også lagres som gjenbruksvennlige ressurser kalt *Prefabs*.

Unity3D har et innebygd brukergrensesnittsystem som heter Unity UI[51], som gjør det lett å designe og implementere brukergrensesnitt visuelt i editoren. Alle elementene i dette systemet må være underobjekter av rotobjektet Canvas, som representerer arealet brukergrensesnittet skal befinne seg. Systemet bruker et EventSystem objekt for å håndtere inndata fra brukeren[49]. EventSystem-objektet kan utvides med moduler slik at man kan definere hvordan systemet oppfatter brukerens handlinger.

2.5.1 Unity Asset Store

Unity har en egen integrert butikk i editoren hvor man kan laste ned digitale ressurser som modeller, skripter og verktøy som andre utviklere har laget. Dette gjør det mulig å laste ned og bruke ressurser som er bygget for Unity3D rammeverket, slik at man slipper å måtte lage all funksjonalitet selv. Ressursene i Unity Asset Store kan koste penger, men mange av dem er gratis.

2.6 OBJ 3D-modeller[25]

OBJ formatet (.obj) er et format som definerer en eller flere geometriske modeller. Formatet ble utviklet av Wavefront Technologies som en del av en animasjonspakke de utviklet. I senere tid har filformatet blitt adoptert av andre 3D-grafikkverktøyselgere. Filformatet har støtte for flere forskjellige attributter som beskriver hvordan 3D-modellen skal vises grafisk i en gitt applikasjon. En av de viktigste attributtene er punktdataen som beskriver strukturen til modellen, med normaler som peker i hvilken retning punktene er vendt. Posisjonene til punktene er representert som vektorer, og er relative til nullvektoren der de er plassert i modellens interne koordinatsystem. Ved å flytte nullvektoren, vil man flytte modellen også.

2.7 Nettverksprotokoller

Nettverksprotokoller er standardiserte måter for datamaskiner å kommunisere over internettet[42]. Det finnes mange nettverksprotokoller, men TCP, UDP og FTP er de mest relevante for programmet vårt, så vi kommer kun til å snakke om de i dette delkapittelet.

FTP er en protokoll som opererer på applikasjonsnivået, mens TCP og UDP er protokoller som opererer på transportnivået i OSI-modellen[43]. Siden UDP og TCP opererer på samme abstraksjonsnivå, må man velge hvilken av dem man vil bruke til forbindelser i hvert system. TCP og UDP har mange forskjeller, som vi kommer til å skrive om i de neste avsnittene[44], i tillegg til hvilke bruksområder hver av dem er best egnet til. På slutten av delkapittelet kommer vi tilbake til FTP.

TCP er en pålitelig, tilkoblingsbasert protokoll som sørger for at alle pakker kommer frem, og i riktig rekkefølge. TCP gjør det lett for en utvikler å få programmet sitt til å fungere uten å måtte tenke på mulige nettverksfeil som kan oppstå hos kundene sine. Påliteligheten til TCP gjør den til en godt egnet protokoll for engangsforespørsler og data som må bli splittet opp i flere pakker. TCP blir ofte brukt innen filoverføring, epost og for nettsider[44].

UDP er en protokoll som ikke tilbyr noe av påliteligheten man finner i TCP. Med UDP er det mulig at pakker kommer frem i feil rekkefølge eller at de ikke kommer frem i det hele tatt. Denne mangelen på pålitelighet gjør UDP-protokollen godt egnet for hyppige, selvstendige beskjeder. UDP blir ofte brukt innen spillservere[44], mediastrømming[47] og torrent-teknologi[46].

Man kan ta for seg et hypotetisk scenario hvor en klient A beveger seg i et rom mens en klient B opplever midlertidige internettproblemer. Hundre bevegelspakker blir sendt over nettet til klient B over et par sekunder. Klient B sine internettproblemer går over mot slutten av vårt scenario, og B mottar den siste av de hundre nettverkspakkene. Hvis TCP ble brukt, måtte de første 99 pakkene ha blitt sendt på nytt igjen før pakke nr. 100 kunne blitt prosessert. Dette ville tatt lang tid fordi B må sende en forespørsel til A for hver av de manglende 99 pakkene før A kan sende dem igjen. Hvis for mange pakker blir borte, kommer "congestion control" inn i bildet og sakker ned sendingen enda mer[45]. Hvis UDP ble brukt hadde de manglende 99 pakkene blitt ignorert, og programmet kunne prosessert den mottatte bevegelsespakken som om ingenting hadde skjedd.

FTP er den vanligste filoverføringsprotokollen i verden og er lagd for å overføre filer mellom en klient og en server over en TCP-forbindelse[48]. FTP er en veldig fleksibel protokoll, men krever at filserveren støtter alle FTPs mange kommandoer. Av denne grunnen er FTP best egnet til dedikerte filservere som ikke brukes til andre oppgaver.

2.8 Blokkerende socketer

Det er ingen forskjell på blokkerende og ikke-blokkerende socketer, med mindre bufferen som leses fra er tom. Om en blokkerende socket leser fra en tom buffer vil den pause tråden til den har noe i bufferen. Hvis en ikke-blokkerende socket prøver å lese fra en tom buffer, vil den gi tilbake enten null eller en feil basert på implementasjonen[50].

Ikke-blokkerende socketer har som fordel at man kan avbryte forespørsler når som helst, og at man generelt har mer kontroll over logikken til nettverksdelen av applikasjonen [50]. Blokkerende socketer vil ikke kunne gi like mye kontroll over forbindelsen, men gjør det mye lettere og raskere å implementere fungerende nettverksprotokoller. Hvis man skal bruke blokkerende socketer må man kjøre de i en egen tråd for å unngå at hovedprogrammet skal fryse. Derfor vil blokkerende socketer kreve flere maskinressurser.

3 MATERIALER OG METODE

3.1 Materialer / Programvare

3.1.1 Datamaskiner

Datamaskinene som ble brukt under utvikling og testing av applikasjonen. De er eid av studentene selv, og er stasjonære.

Eier	Operativsystem	Prosessor	Minne	Grafikkort
Fredrik Foss	Windows 7 64-bit	Intel Core i7-3770K @3.50 GHz	16GB DDR3 @1600 MHz	NVIDIA GTX 1060
Marcus B. Johansson	Windows 10 64-bit	Intel Core i7-9770K @4.80 GHz	32GB DDR4 @3000 MHz	NVIDIA GTX 1080TI

3.1.2 Oculus Rift med Touch kontrollere

Programmet skal kjøre på både Oculus Quest (Android) og Oculus Rift (Windows 10). Siden Oculus Quest ikke har blitt lansert enda, tester vi på Oculus Rift. Ifølge Oculus skal man kunne utvikle til Oculus Rift for å enklere overføre applikasjonen til å støtte Oculus Quest ved et senere tidspunkt[10]. Vi bruker også Oculus Touch kontrollerne, som Oculus Quest også blir levert med. Både VR-brillene og kontrollerne ble lånt fra NTNU Ålesund.

Figur 1: Oculus Rift briller og Oculus Touch kontrollere.

3.1.3 Unity3D 2017.4

Den nyeste av Unity sine langtidsstøttede versjoner. Dette er ikke den nyeste versjonen av Unity, men Oculus-biblioteket var bare kompatibel med denne versjonen av Unity da vi startet prosjektet. Unity har innebygd støtte for VR, og fungerer på alle de mest populære plattformene; inkludert Windows, Mac, Linux, Android og IOS.

3.1.4 Unity Collaborate

For å synkronisere prosjektet mellom gruppemedlemmene brukte vi Unity Collaborate, et integrert versjonskontrollsystem som fungerer direkte i Unity3D editoren. Systemet ligner Git, der medlemmer kan laste endringer opp eller ned fra et sky-lager eid av Unity. Collaborate er gratis for grupper med 3 eller færre medlemmer, og tilbyr opp til én gigabyte lagringsplass gratis.

3.1.5 JIRA

JIRA er et planleggingsverktøy utviklet av Atlassian for å gjøre utvikling med smidige metoder lettere[21]. JIRA tilbyr flere digitale verktøy som gjør det lett å holde oversikten over prosjektets fremgang og problemer. Gjennom JIRA er det mulig å generere rapporter med nyttige grafer som gir utviklingslaget en oversikt over prosjektets fremgang. Aspekter med kjente smidige metoder som scrum og kanban finner du også i JIRA, der man kan opprette brukerhistorier, problembeskrivelser, og feilfiksing som kan delegeres til utviklingslaget under sprinter.

NTNU Ålesund har sin egen JIRA server for studentbruk, så vi har fått bruke JIRA gratis mens vi jobbet på dette prosjektet.

3.1.6 JetBrains Rider 2018.3

Rider er en IDE utviklet av JetBrains som er bygget for .NET utvikling i Windows og Mac. Utviklingsplattformen har utallige funksjoner som gjør utvikling lettere og mer produktivt. Rider har innebygd støtte for Unity3D applikasjoner gjennom en modul som blir installert automatisk hvis du har Unity installert. Rider er essensielt hva IntelliJ er for Java, bare for C#. Alle produkter av JetBrains er gratis for studenter.

3.1.7 ObjReader av Starscene Software

Et betalt tredjepartsbibliotek brukt for importering av .obj filer mens programmet kjører. ObjReader støtter også importering av assosierte materialer og teksturer via. mtl-filer. Modulen inkluderte kildekoden som vi brukte til å tilpasse funksjonaliteten til vårt bruksområde.

3.1.8 Doxygen 1.8.15

Programvare for Windows, Mac og Linux lagd for å generere dokumentasjon basert på kildekode[20]. Programmet blir hovedsakelig brukt for å skape annoterte C++ kilder, men har også støtte for andre språk, som C#. Verktøyet kan generere dokumentasjon i html-format, som kan leses i en nettleser; LaTeX-format, som kan brukes til å generere PDF-dokumenter; eller RTF-format, for Microsoft Word-støtte. Ettersom at dokumentasjonen er generert direkte fra kildekoden, er det lett å holde den oppdatert.

3.1.9 Draw.io diagramverktøy

Vi brukte nettsiden draw.io[18] for å tegne diagrammene brukt i denne rapporten. Det er et program med offentlig tilgjengelig kildekode lagd for å designe og produsere diagrammer for applikasjoner og systemer. Draw.io er også ett av verdens mest brukte nettleserbaserte diagramverktøy[19].

3.1.10 Adobe Illustrator CS6

Adobe Illustrator er et redigeringsverktøy for vektorgrafikk som er utviklet av Adobe[54]. Verktøyet gjør det mulig å enkelt designe og lage illustrasjoner med farger, former, effekter, og typografi. Illustrasjonene kan brukes over flere forskjellige områder som trykk, web, applikasjoner, animasjoner og mer. Verktøyet brukes hovedsakelig for vektorgrafikk, men støtter også bildeformat. Illustrator ble brukt der Draw.io ikke var tilstrekkelig.

3.1.11 Clumsy 0.2

Clumsy er et verktøy som kan simulere forskjellige nettverksproblemer[41]. Den behøver ingen installasjon eller konfigurering med prosjektet, men avskjærer nettverkspakker som blir sendt og/eller mottatt av den lokale maskinen. Man kan også bruke et filter, slik at bare pakker fra en gitt IP adresse, port og/eller protokoll blir tuklet med av verktøyet. Clumsy kan simulere forsinkelser, misting av pakker, struping (blokkere alle pakkene i en viss tid, for så å sende alle i en forsendelse), sende pakkene i feil rekkefølge, pakkeduplisering og pakkekorrupsjon (bytte om tilfeldige bit-er i pakken).

3.1.12 DOTween av DemiGiant

DOTween er et raskt og effektivt bibliotek som gjør det lett å interpolere fra en verdi til en annen, over en gitt tidsperiode[56]. Dette ble brukt til å animere kontekstmenyen sine element for å gi visuell respons til brukeren.

3.1.13 Kommunikasjonsverktøy

Gruppemedlemmene kommuniserte hovedsakelig gjennom plattformen Discord. Dette innebar bruk av chatten for spørsmål og svar, samt samtaler for daglige scrum møter (om det var problemer/noe som måtte diskuteres). En av fordelene med å bruke Discord er at det har innbygd støtte for syntaksbelysning som gjør det lettere å diskutere kodespørsmål.

3.2 Metode

3.2.1 Prosjektorganisasjon

Oppdragsgiver: Oppdragsgiver for dette prosjektet var Knut Steinnes, daglig leder for RUFO AS.

Utviklingslag: Scrum-laget besto av to dataingeniørstudenter ved NTNU Ålesund.

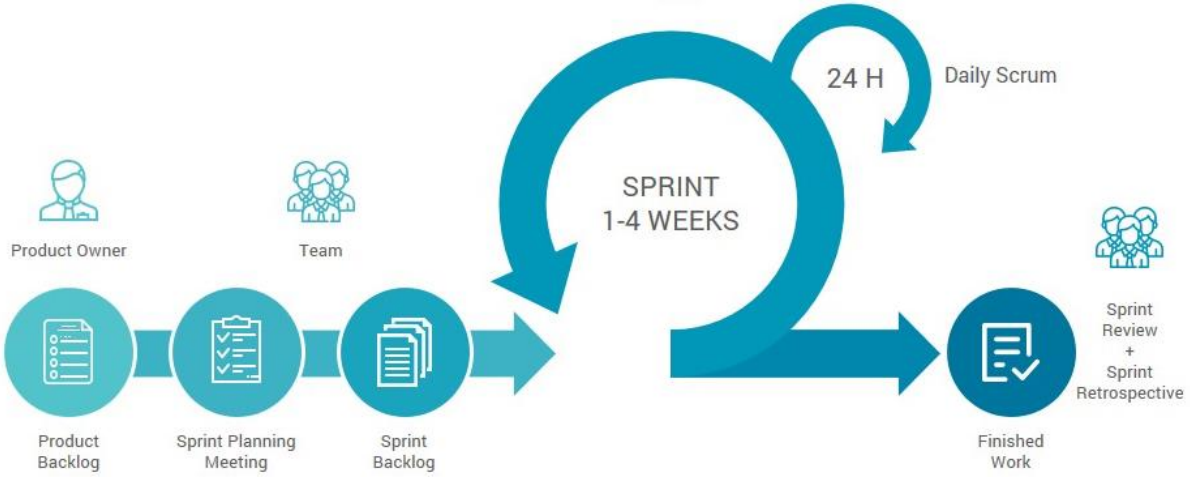
Scrum-mester: Marcus Benjamin Johansson

Veileder: Veilederen for dette prosjektet var Mikael Tollefsen, Universitetslektor, Institutt for IKT og realfag ved NTNU Ålesund.

3.2.2 Utviklingsmetodologi

Dette prosjektet ble utviklet med en scrum-basert metodologi. Dette ble gjort fordi scrum og lignende arbeidsmetoder ofte fungerer bedre for IT-prosjekter i forhold til tradisjonelle utviklingsstrategier som fossefallsmetoden. Om man skal bruke en scrum-basert arbeidsflyt burde man følge arbeidsmetodene nøye og holde seg til rollene[32]. Det ble brukt JIRA i prosjektet for organisering og planlegging, som gjorde det betydelig lettere å opprettholde en scrum-basert arbeidsflyt. Det ble valgt å jobbe i 2-ukers sprinter for dette prosjektet, siden vi

følte at det ga oss god nok tid til å få arbeid gjort hver sprint uten å la kunden vente for lenge på oppdateringer. Tilbakemelding og kommunikasjon med produkteier ble gjort over epost. På slutten av hver sprint ble det sendt en kort oppsummering av hva gruppen hadde oppnådd, bilder/video om aktuelt, og hva som var planlagt å gjøre neste sprint.



Figur 2: Scrum prosessen (figur tatt fra https://brainhub.eu/blog/differences-lean-agile-scrum/)

3.2.3 Testing og problemløsing

Unity sitt komponentsystem gjorde at vi sjeldent trengte å endre på gamle systemer for å få nye systemer til å fungere. Det faktumet, kombinert med at programmet ikke hadde mange presist målbare elementer gjorde at vi valgte å ikke bruke enhetstesting (eng: unit testing). Som et resultat av dette, ble det meget lett å finne ut av hvor og hvorfor nye problemer oppsto. De fleste problemene vi møtte på var enkelt å oppdage og å finne kilden til, som påvirket vår testmetodikk.

Vi lagde ikke en testplan, men istedenfor benyttet vi oss av dynamiske testmetoder som gorillatesting og "white box testing". Integrasjonstesting og systemtesting ble gjort mot slutten av hver sprint. På starten av prosjektet ble integrasjonstesting og systemtesting gjort lokalt, men gikk over til å samarbeide mer og mer mot slutten av prosjektet da nettverksfunksjonaliteten kom på plass.

Gorillatesting[52] er en «negativ» testmetode hvor man prøver så hardt man kan å ødelegge funksjonaliteten til programmet. For eksempel så gorillatestet vi kallenavnsystemet vårt ved å gi oss selv tomme navn, alt for lange navn og ved å bruke spesialkarakterer som "\n".

"White box testing"[53] er en ustrukturert testmetode hvor en utvikler som kjenner et system godt prøver forskjellige ting og ser om alt fungerer som forventet. Det heter "white box testing" fordi utvikleren selv har veldig god oversikt over koden han tester. Ved å vite nøyaktig hvordan en modul fungerer, vet man også nøyaktig hvilke aspekter med modulen som må bli testet og på hvilke måter.

Problemer som oppsto på hovedtråden var lette å finne, da hovedtråden alltid fortalte oss hvilken fil og hvilken linje som krasjet programmet. Prosesser som kjørte utenfor hovedtråden var verre å finne problemer i, da de ikke sa ifra hvis de krasjet, og lot hovedtråden fortsette som om ingenting hadde skjedd. For å finne krasjpunktet i sekundære tråder gjorde vi som så: Først brukte vi store try-catch blokker for å finne ut hvilken generell del av programmet som hadde feilet. Innimellom var feilbeskjeden vi fikk fra try-catch-en nok til å vite hvilken linje

som krasjet tråden, men som oftest måtte vi prøve mindre og mindre try-catch-blokker for å finne frem til den problematiske linjen.

Etter å ha funnet et problem, fikset vi det ved å skrive variabler til Unity sin loggterminal for så å bruke vår kunnskap om modulen til å finne frem til problemet. Fordelen med Unity sin loggterminal over utskrivings-kommandoer i de fleste editorer er at Unity sin lar oss finne problemer ikke bare i editoren, men også i kompilerte versjoner av programmet. Denne oppførselen var veldig hjelpsom da vi jobbet med nettverksfunksjonalitet, fordi det lot oss sjekke og teste variabler hos både mesterklienten og en vanlig klient på samme datamaskin.

4 RESULTATER

4.1 Arkitektur i Unity3D

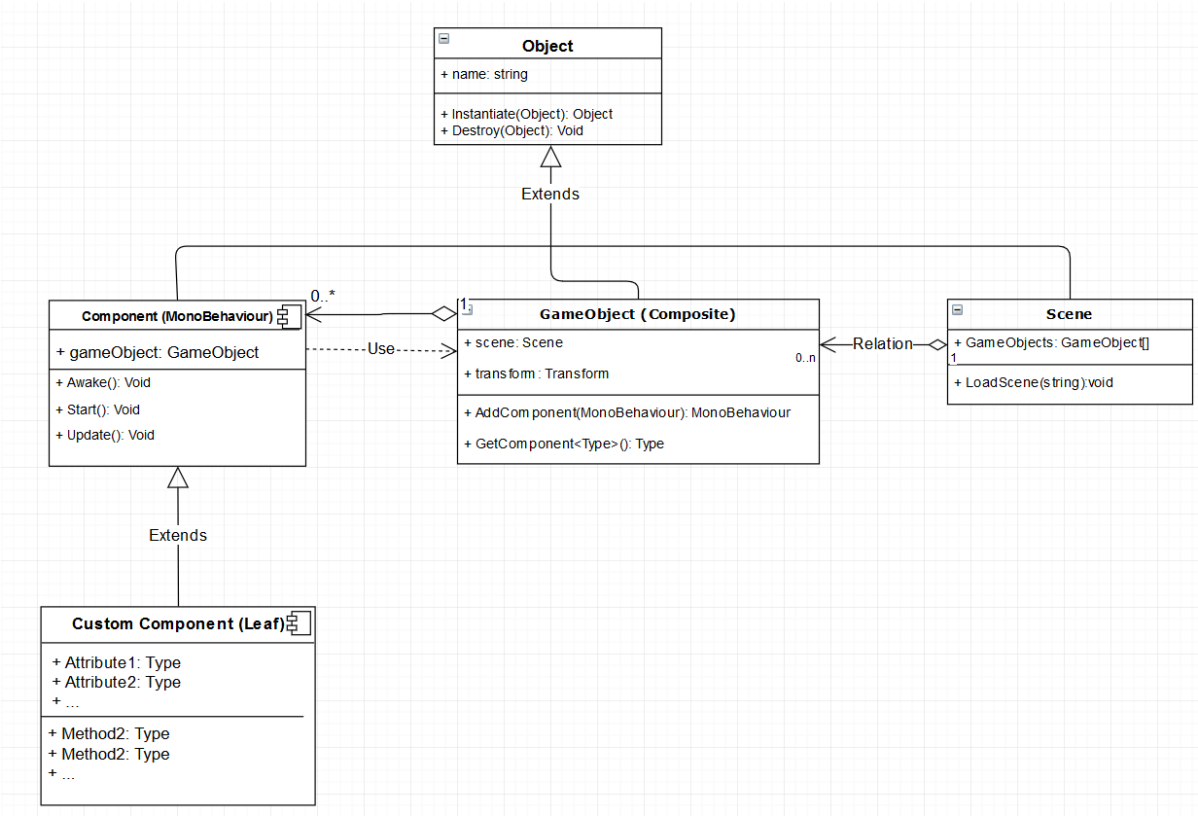
Når man utvikler i en tilrettelagt spillmotor vil det påvirke kodearkitekturen grundig fra starten av. Derfor er det viktig at man forstår arkitekturen i Unity3D for å bedre forstå denne applikasjonen sin arkitektur og løsninger.

Unity3D bruker Scener som representerer verdener/nivåer. Alle objekter i en scene er av typen GameObject, og er en del av et hierarki der scenen er roten. GameObject fungerer som kompositter man kan tilføye egne komponenter. Detter er skript som fungerer likt blader i komponentmønsteret. Komponenter må arve fra MonoBehaviour-klassen, som er Unity sitt komponent-grensesnitt. GameObject-instanser kan også bli lagret som gjenbruksvennlige ferdiglagde objekter kalt Prefabs.

At Unity3D er en komponentbasert spillmotor betyr at man ikke er avhengig av å bruke store og komplekse klassehierarkier for å definere delt funksjonalitet mellom objekter. Man legger til de nødvendige komponentene for å oppnå ønsket funksjonalitet uten å potensielt arve unødvendig kode. Dette gjør det lettere å utvide funksjonalitet ettersom at vi ikke behøver å tenke på konsekvenser i andre subklasser. Gjennom komponenter kan man enklere isolere funksjonalitet mellom klasser, noe som gjør det lettere å designe og produsere kode med lav kobling og høy kohesjon.

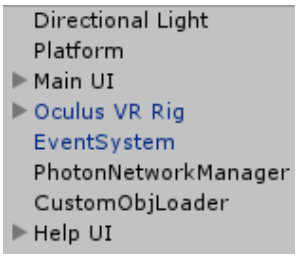
En ting som skiller Unity fra de fleste andre komponentbaserte systemer er at alle komponenter i Unity må ha koordinater, en størrelsesfaktor og en rotasjon. Denne samlingen er representert i «Transform» (transformasjons) komponenten, kan verken legges til eller fjernes. For kompositter som er underlagt andre kompositter vil transformasjonen være relativ til forelderen istedenfor å ha en absolutt transformasjon. Dette gjør det mulig å flytte hele hierarki av kompositter, ved kun å endre på rot-kompositten. Relative koordinater gjør det også lettere å regne ut presise plasseringer av geometriske former for å få ting til å se penere ut. For eksempel er det lettere å skrive inn posisjonene 0 og 5 enn -0.47 og 4.53 hvis man har to objekt som skal plasseres 5 meter unna hverandre.

Figur 3: Eksempel på objektforhold i scenehierearkiet.



Figur 4: UML diagram over arkitekturen i Unity3D

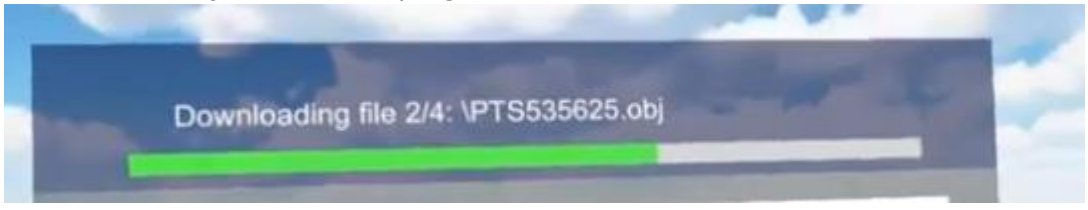
4.1.1 Sceneorganisering



Figur 5: Systemer som kjører i lobbyen.

For å organisere scenene bestemte vi oss for å bruke ett objekt for hvert hovedsystem i applikasjonen. Det betyr at vi f.eks. har ett objekt ved navn "Voice Manager" som holder på alle komponentene relatert til stemmekommunikasjon, og et annet objekt ved navn "EventSystem" som inneholder alle komponenter relatert til håndtering av inndata fra håndkontrollerne. Alternativet til dette hadde vært å ha ett "Ambient systems"-objekt som holdt på alle bakgrunnsprosessene, men det følte vi at hadde vært dårlig design.

Vi har to scener i programmet vårt. Den første, som applikasjonen starter i, er en enkel lobby der brukeren står på en plattform. Scenen inneholder informasjon om hvordan man bruker programmet, og et brukergrensnitt for å velge kallenavn og koble til den neste scenen; møterommet. Brukergrensesnittet for tilkobling viser også fremgang på nedlasting/importering av objekter via en horisontal fremdriftslinje (se figur 6). Vi kommer ikke til å diskutere møterommet i detalj i dette avsnittet, men det er der man får tilgang til resten av funksjonaliteten til programmet.



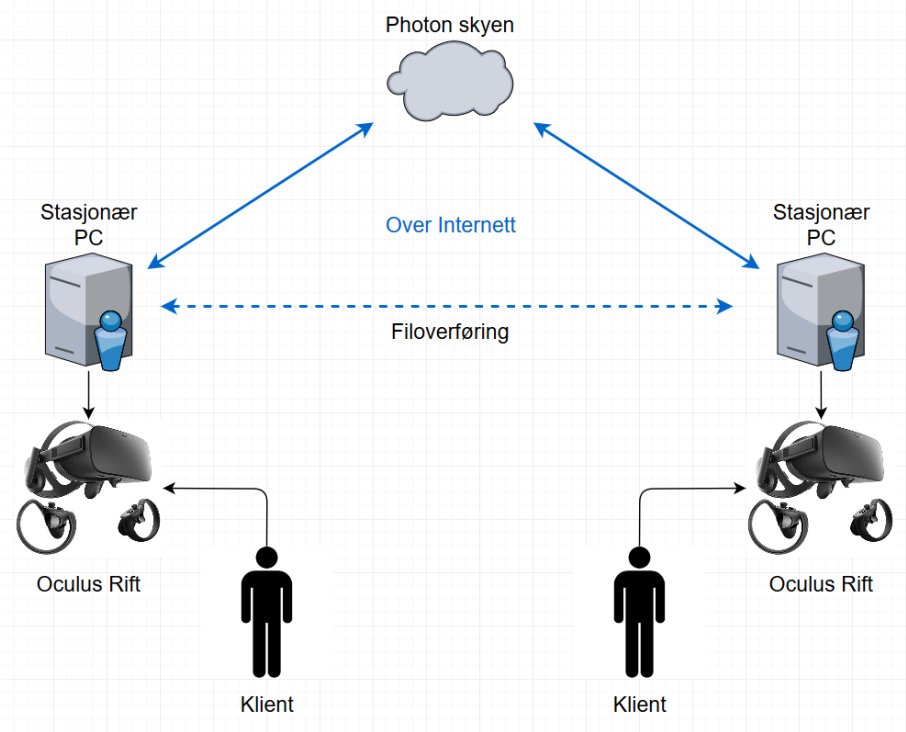
Figur 6: Filoverføringsfremgang

Da vi først implementerte nettverksfunksjonaliteten hadde vi to objekter dedikert til nettverksfunksjonalitet, en i hver scene. Dette var et dårlig design fordi det tvang oss til å håndtere nettverkshendelser to forskjellige steder, som førte til kodeduplikasjon. Vi fjernet nettverksobjektet i møteromscenen og satte et flagg på det første nettverksobjektet som hindret det i å ødelegges ved scenebytte.

4.2 Systemarkitektur

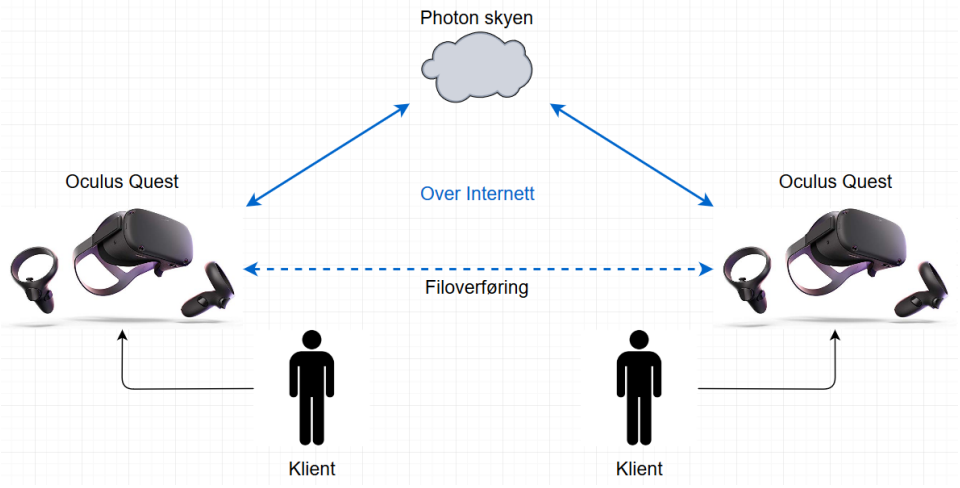
4.2.1 Maskinvare

Oversikt over oppsettet med klassiske VR-briller (brukt i utviklingsfasen):



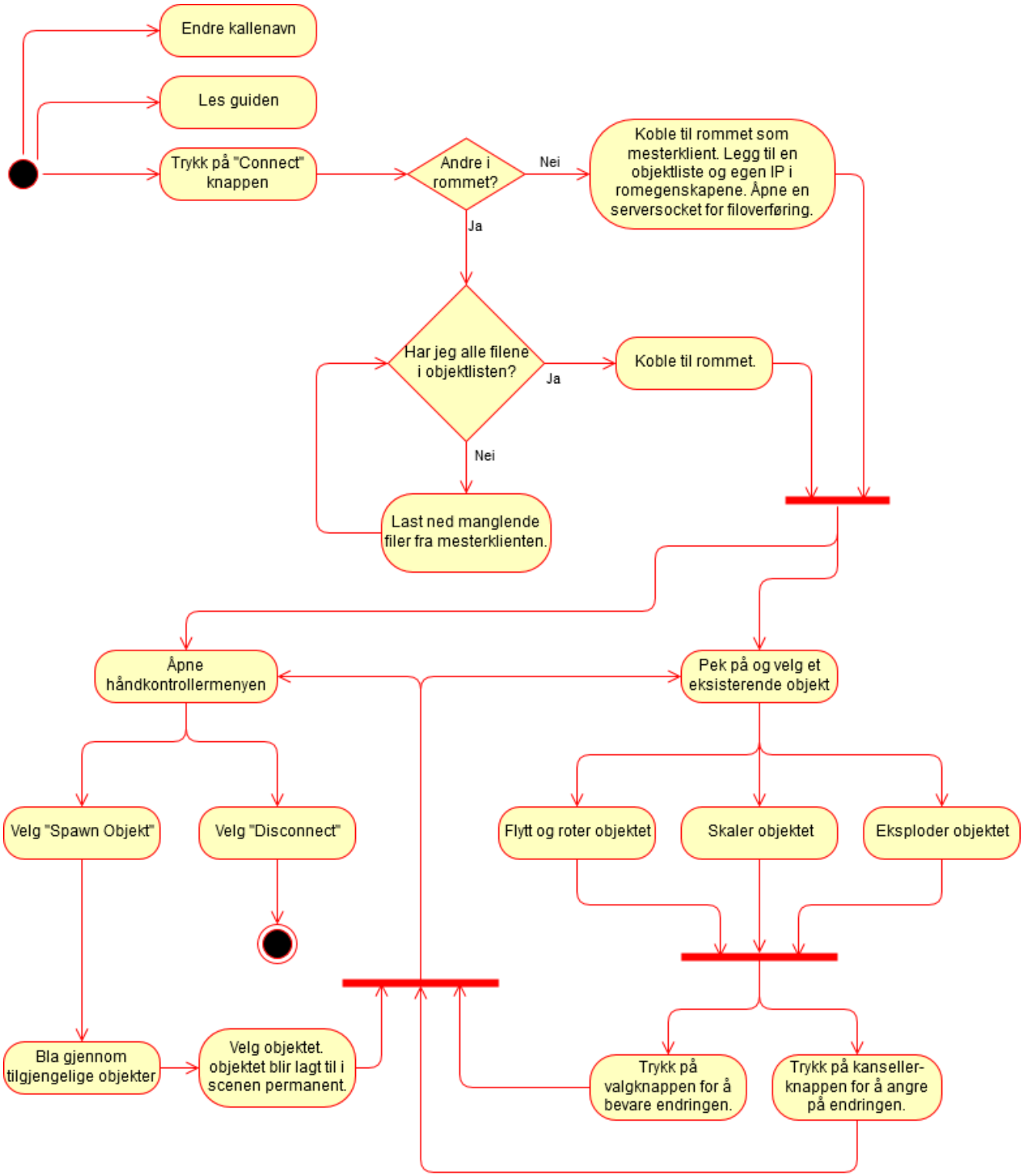
Figur 7: Utstyr brukt under utvikling

Oversikt over oppsettet som vil bli brukt av RUFO når Oculus Quest kommer ut:



Figur 8: Oppsett med Quest-brillene

4.2.2 Aktivitetsdiagram



Figur 9: Aktivitetsdiagram

Mer informasjon om filoverføringsprosessen ligger i 4.6.3.

4.3 PUN som generell nettverksløsning

Siden vi ble bedt om å lage et distribuert møterom med støtte for flere enn 2 personer, trengte vi en måte å synkronisere tilstander på. Vi så etter Unity-vennlige serverløsninger med støtte for stemmekommunikasjon og fant 2 gode alternativer.

Unity3D har en innebygd nettverksløsning kalt UNet[15]. Det er et utdatert system som kommer til å bli erstattet i senere versjoner av Unity3D, men ville fortsatt fungert for vårt bruksområde. UNet støtter to abstraksjonsnivåer: Et skripting-API på høyt nivå ved navn HLAPI, og et API på lavt nivå som lar deg bruke socketer manuelt. HLAPI lar deg enkelt opprette en vert-til-vert-struktur der en av klientene opererer som en autorativ server. UNet har ingen innebygde verktøy for stemmekommunikasjon, men dette finnes som betalte løsninger i Unity Asset store som man kan kjøpe for å spare tid.

Photon Unity Networking (PUN) var også en mulig nettverksløsning i prosjektet vårt[16]. PUN er en tredjepartsmodul som har liknende funksjonalitet som UNet. PUN er en server-klient-nettverksløsning som er gratis for inntil 20 brukere av gangen. Modulen bruker et romsystem, der klienter kobler til forskjellige rom med forskjellige deltakere. PUN er ikke en dedikert serverarkitektur, så applikasjonen kjører/simuleres ikke på serveren; istedenfor blir all synkronisering gjort via hyppige selvstendige oppdateringspakker over UDP eller TCP. Siden serveren ikke kjører programmet selv, har hvert rom en mesterklient som er autorativ ovenfor den synkroniserte tilstanden til de andre klientene. Photon har også tilleggsmodulen Photon Voice tilgjengelig, som gjør det enkelt sette opp stemmekommunikasjon over IP[17]. Denne modulen er også gratis for inntil 20 brukere av gangen.

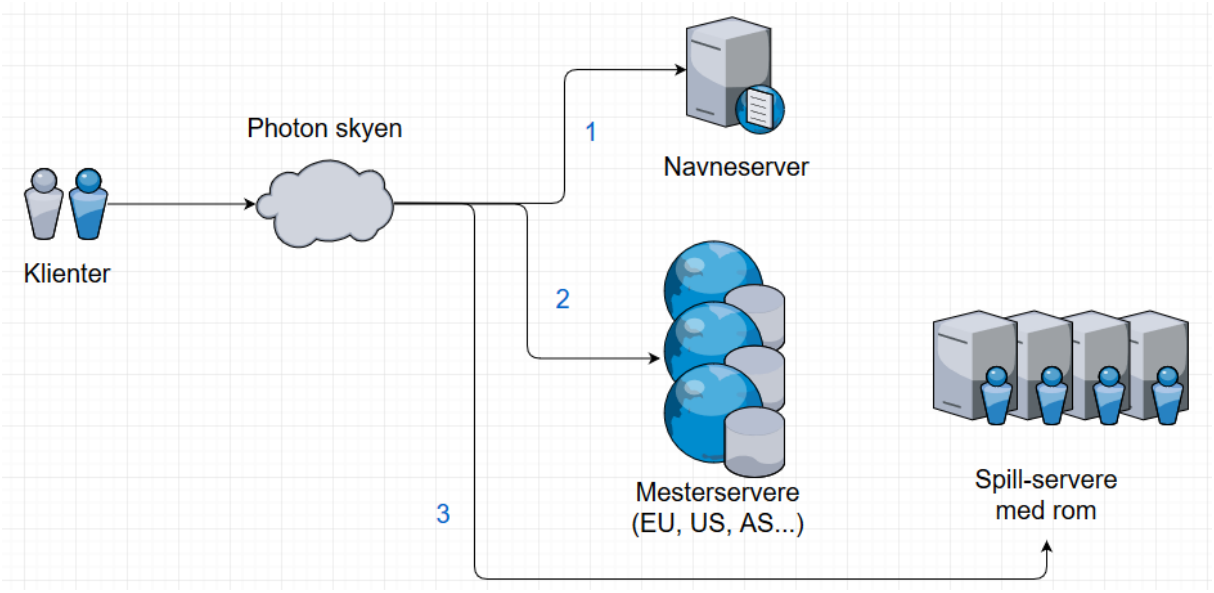
Det ble valgt å bruke Photon-modulen i dette prosjektet hovedsakelig på grunn av dens gratis innebygde støtte for stemmekommunikasjon. Siden UNet holdt på å bli utdatert var dette også en viktig faktor i valget av nettverksmodul. RUFO er en mindre bedrift, så vi antar at de kommer til å ha 2-4 brukere i hvert møte, og trolig ikke ha behov for mer enn ett møte om gangen, spesielt siden denne teknologien er ny. Vi mener derfor at begrensningen på 20 brukere av gangen ikke vil være et problem. Vi valgte derfor å hardkode et romnavn, slik at vi både slapp å legge til romlister og gjorde tilkoblingsprosessen enklere for brukerne.

Det ble valgt å bruke Photon-modulen i dette prosjektet hovedsakelig på grunn av dens gratis innebygde støtte for stemmekommunikasjon. UNet sin kommende pensjonering var også en viktig faktor i dette valget, da det hadde påvirket RUFO sine senere utviklingsmuligheter. Siden teknologien er eksperimentell og RUFO ikke er en gigantisk bedrift, trenger de ikke holde mer enn ett kundemøte over VR til enhver tid. Vi valgte derfor å hardkode et romnavn, som gjorde det lettere for kunder å koble til rommet da de slapp å navigere gjennom en romliste først. Nedsiden med hardkoding av romnavnet er at RUFO må implementere en romliste senere om de begynner å holde flere VR-møter samtidig. Photon er som sagt gratis for inntil 20 brukere av ganger, så med 2-4 deltakere per møterom må RUFO holde minst 5 møter samtidig før mengden brukere hadde blitt et problem, og med så mange samtidige kundemøter burde de ha råd til å betale Photon for tjenestene deres.

4.3.1 Oppkobling til Photon skyen

For å koble til møterommet må en klient først gå gjennom Photons skyarkitektur[24]. Først kobler klienten seg til Photons globale navneserver, som gir en liste over alle tilgjengelige regioner. Deretter pinger klienten alle regionene for å finne ut av hvilken mesterserver (region) den har lavest ping til. Da vi skrev denne rapporten hadde Photon 13 forskjellige

mesterservere[24] spredt over hele verden. Når en region har blitt valgt kan klienten opprette eller koble til eksisterende rom på servere underlagt den valgte regionen. Alle Photon rom blir tilbakestilt etter at alle brukere har forlatt rommet.



Figur 10: Klienters oppkoblingsprosess mot Photon-skyen

Et problem vi møtte på var at hvis en på gruppen hadde dårlig internett og begge koblet til rommet, så havnet de av og til i forskjellige rom. Siden vi hadde hardkodet romnavnet, men ikke regionen; ble regionen valgt basert på hvilken server som hadde lavest ping. Det viste seg at Photons russiske server lå nær nok Ålesund til at en klient med dårlig forbindelse kunne bli sendt dit istedenfor å bli sendt til den europeiske serveren. Dette problemet ble løst ved å presisere at alle klienter skulle koble til den europeiske mesterserveren.

4.3.2 Synkronisering gjennom PUN

For at klienter skal kunne se og samhandle med andre klienter og modeller i rommet, må både scenen og modellene synkroniseres mellom dem i sanntid. Om man ønsker at noe skal oppdateres til et jevnlig intervall, kan man gjøre dette gjennom komponenter kalt PhotonView-er. Disse komponentene inneholder en liste med andre observerte komponenter, som alle implementerer grensesnittet IPUNObservable. Grensesnittet definerer metoden OnSerializeView, som har en referanse til en nettverksstrøm det både kan skrives til og leses fra. Photon har innebygd serialisering og deserialisering av vanlige datatyper som int, float, Object, Collection og lignende[9], men det er også mulig å legge inn serialiseringsregler for egendefinerte klasser og strukturer. Vi trengte ikke den funksjonaliteten i dette prosjektet ettersom at vi kun synkroniserte byte- og float-verdier.

Hvert PhotonView i scenen har en eier som er autorativ over det. Denne eieren er den eneste klienten som kan skrive tilstanden til PhotonView-et til nettverksstrømmen. Eierskap av PhotonView-er kan bli overført mellom klienter og scenen. Hvis scenen eier et PhotonView blir mesterklienten autorativ over den.

I tillegg til å synkronisere tilstanden til modeller, brukte vi også Photon til å synkronisere posisjonen, rotasjonen og håndtilstanden til hver klients avatar. En avatar består av et hode og to hender. Posisjonen og rotasjonen til hodet og hendene ble synkronisert via serialiserte

```
[Flags]
22 usages  7 exposing APIs
private enum HandAnimationFlags
{
    None = 0,
    RightPoint = 1,
    RightGrip = 2,
    RightThumb = 4,
    LeftPoint = 8,
    LeftGrip = 16,
    LeftThumb = 32
}
```

Figur 11: Faste håndtilstands-verdier.

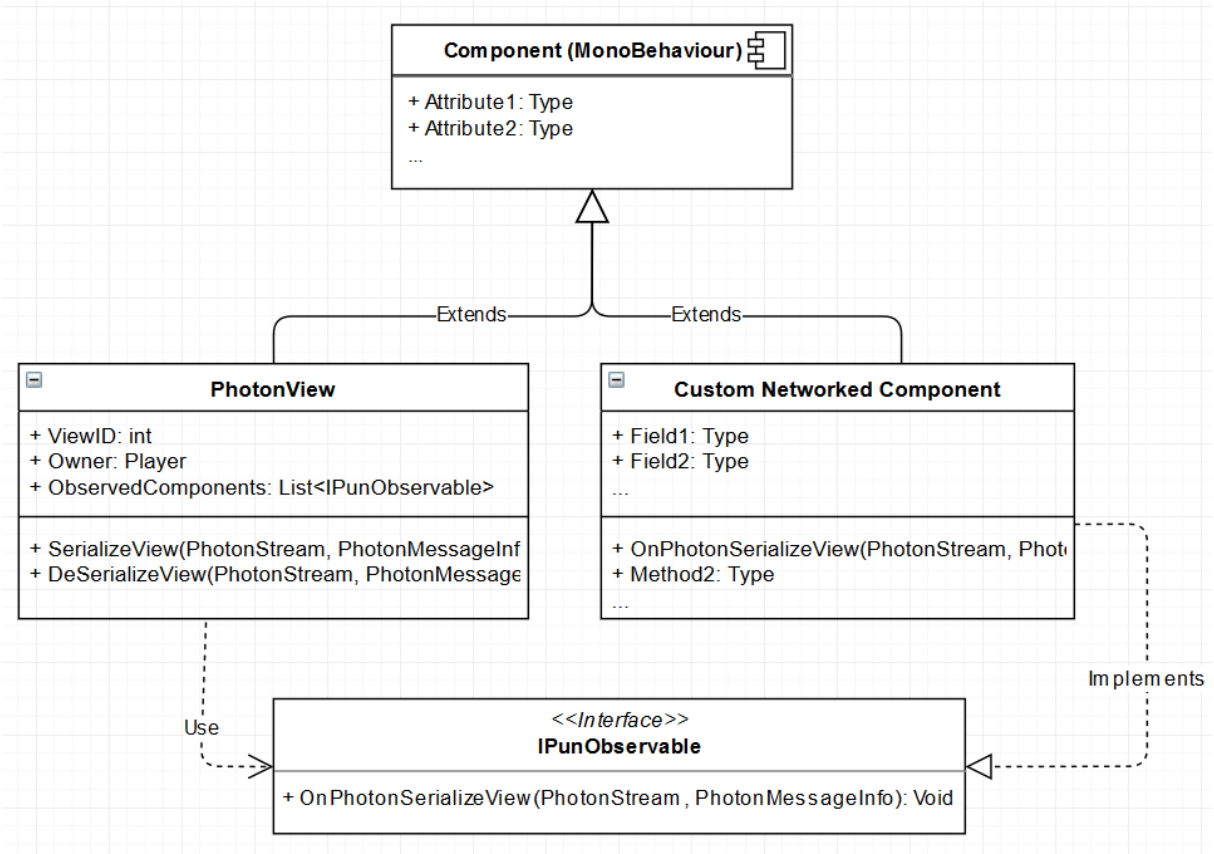
float-verdier. Hver hånd hadde i tillegg 2^3 forskjellige tilstander de kunne være i basert på brukerens fingerposisjoner. For å redusere nettverkstrafikk brukte vi en byte til å holde på tilstanden til begge hendende, dette ga oss 2^4 tilstander å jobbe med per hånd.

Oppdateringspakker blir sendt 20 ganger i sekunder, så for å hindre at avatarene så hakkete ut mens de bevegde seg brukte vi interpolering for å simulere en gradvis oppdatering.

```
private void UpdateRemoteHandAnimations(byte animationFlags)
{
    HandAnimationFlags flags = (HandAnimationFlags) animationFlags;

    targetRightPoint = (flags & HandAnimationFlags.RightPoint) == HandAnimationFlags.RightPoint ? 1 : 0;
    targetRightGrip = (flags & HandAnimationFlags.RightGrip) == HandAnimationFlags.RightGrip ? 1 : 0;
    targetRightThumb = (flags & HandAnimationFlags.RightThumb) == HandAnimationFlags.RightThumb ? 1 : 0;
    targetLeftPoint = (flags & HandAnimationFlags.LeftPoint) == HandAnimationFlags.LeftPoint ? 1 : 0;
    targetLeftGrip = (flags & HandAnimationFlags.LeftGrip) == HandAnimationFlags.LeftGrip ? 1 : 0;
    targetLeftThumb = (flags & HandAnimationFlags.LeftThumb) == HandAnimationFlags.LeftThumb ? 1 : 0;
}
```

Figur 12: Oppdatering av håndtilstander basert på oppdateringsbyte.



Figur 13: PUN synkroniseringsarkitektur

4.3.3 Synkronisering av importerte modeller

Vanligvis er det enkelt å opprette nettverkssynkroniserte objekter via Photon. Om objektet allerede eksisterer i scenen når applikasjonen bygges, behøver man kun en PhotonView-komponent for at den skal kunne bli synkronisert med resten av brukerne. Om man ønsker å opprette et synkronisert objekt som ikke starter i scenen, kan man gjøre dette enkelt med et funksjonskall i PhotonManager-klassen, gitt at objektet eksisterer som en prefab. Begge disse metodene krever at modellene blir lagt inn før programmet blir kompilert, så vi måtte finne en annen løsning.

Løsningen ble å definere en containerklasse ved navn CustomNetworkedBox, som de importerte modeller ble hektet til. Modellenes posisjoner og rotasjoner ble da knyttet til containerobjektet, slik at man kan flytte modellen ved å flytte containerobjektet. Siden containerobjektet inneholdt et PhotonView, kunne containeren synkroniseres over Photon, som implisitt ville oppdatere modellen som containeren holdt på. For at de andre klientene skulle vite hvilken importert modell som ble hektet på et gitt containerobjekt, sendte vi modellens navn via RPC. Klientene kunne så finne den korrekte modellen på sin maskin, for så å oppdatere modellen i sitt containerobjekt.

4.3.4 Synkronisering av nye brukere

Et problem vi kom over var scenarioet der en ny bruker blir med i rommet mens det allerede er modeller i rommet. Photon kan ikke oppdatere disse automatisk siden Photon ikke vet om de importerte modellene, bare containerobjekter som tidligere har blitt oppdatert via RPC.

Det er mulig å hurtiglagre alle RPC-er på Photon-serveren, men da vil alle RPC-ene bli kjørt hos den nye brukeren, selv om de er gamle og/eller unødvendige. Vi ønsker kun informasjon om modellens nåværende tilstand. En annen mulighet er å sende nye RPC-er med den nødvendige informasjonen. Problemet er at *alle* brukerne vil få oppdateringen som resultat av hvordan RPC-er fungerer. Dette medfører sløsing av nettverkstrafikk for brukere som allerede er synkronisert, og vil gjøre løsningen mindre gunstig.

Løsningen som ble valgt var Photon-hendelser. De fungerer noe likt som RPC, men med flere konfigurasjonsmuligheter. Der RPC kaller de samme metodene på de samme objektene hos alle klienter (gjennom å bruke PhotonView-komponentens ID), har ikke Photon-hendelser noen form for innebygd identifikasjon for hvilke objekter som mottar meldingene. Siden vi må oppdatere alle importerte og opprettede modeller, trenger vi å identifisere dem for å synkronisere dem. Vi brukte CustomObjLoader-objektet, en singleton ansvarlig for import av modeller, for håndtering av Photon-hendelsene. Den holder på en liste over alle importerte modeller som har blitt opprettet i scenen. Dette gjør det enkelt å sende en oppdatering til den nye klienten for å synkronisere scenens nåværende tilstand. Siden mengden nyttig informasjon er liten, kan dette innkapsles i en objekt-rekke, altså én for hvert objekt som skal synkroniseres.

Fordi Photon kan serialisere Collection-objekter, valgte vi å bruke et Dictionary<int, object[]> der PhotonView-komponentens ID blir brukt som nøkkel. Når den nye brukeren mottar meldingen, kan den gå gjennom sin liste med opprettede modeller og bruke ViewId-en som nøkkel til å finne den korrekte dataen i Dictionary objektet, for så å synkronisere den lokale modellen.

4.3.5 Stemmekommunikasjon i PUN

Stemmekommunikasjon var lett å implementere med Photon Voice. Alt som måtte gjøres var å fylle et tomt GameObject med tre komponenter som fulgte med modulen. Deretter var det bare å justere innstillingene til alt fungerte slik vi ønsket. Vi satte Photon Voice til å automatisk starte stemmekommunikasjon ved tilkobling til møterommet. For å unngå bakgrunnsstøy, satte vi den til å kun ta opp lyd om stemmens volum var over en gitt grense.

4.4 Håndtering av inndata for brukergrensesnitt

For at brukeren skal ha mulighet til å samhandle med applikasjonen må det både finnes grensesnitt som brukeren kan kommunisere med, og metoder for å fange opp brukerens handlinger fra styreenhetene de bruker. I en VR-applikasjon er det ekstra viktig at man håndterer inndataen fra VR-brillene korrekt. Det vil si at det fysiske synspunktet til brukeren skal bli gjenspeilet i den virtuelle verdenen, slik at brukeren ikke blir kvalm. Siden applikasjonen skal bli brukt av personer med uvisst kjennskap til teknologi var det viktig at vi designet intuitive og engasjerende brukergrensesnitt.

Det ble valgt å bruke Unity3D sitt innebygde brukergrensesnittsystem da det både er kraftig og integrert i Unity-editoren. Unity3D har innebygd støtte for de fleste VR-briller, inkludert Oculus[11]. Det ble også brukt en tilleggsmodul av Oculus[12] som utvider funksjonaliteten til brillene og gjør det lettere å sette opp grunnfunksjonalitet for VR-applikasjoner. Vi brukte Oculus-modulen hovedsakelig for styring/navigering, og til tegning av hender.

4.4.1 Styreenheter i VR

For å kunne samhandle med programmet trenger brukere instrumenter å samhandle med. Et alternativ var å bruke klassiske instrumenter som mus og tastatur. Disse har fordelen av at de fleste brukere har erfaring i å bruke dem fra før, men typiske tastaturoppsett for bevegelse i 3D (som WASD) kan være mindre kjent. En annen mulighet er å bruke en klassisk spillkontroller, noe som fulgte med Oculus Rift brillene. Her vil styring kanskje bli noe mer intuitivt for en ny og uerfaren bruker, der man kan bruke styrespaker for å bestemme retning og rotasjon. Den tredje løsningen var å bruke Oculus Touch kontrollene som ble lansert etter Rift brillene. De inkluderer også styrespaker på kontrollerne, men mest viktig er at de støtter sporing via Oculus sensorene. Dette gjør det mulig å gjenspeile posisjonen og rotasjonen til brukerens hender i den virtuelle verdenen.

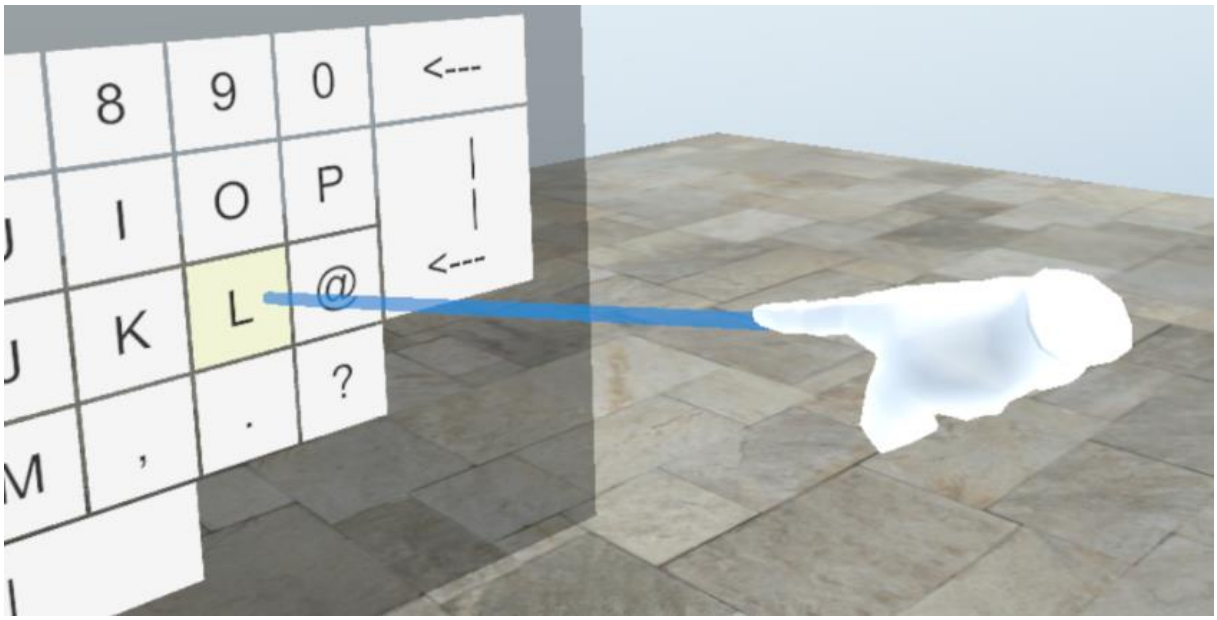
I vår løsning valgte vi Oculus Touch kontrollerne for å styre applikasjonen. Touch kontrollerne var det beste alternativet da de var lagd for VR-opplevelser. De gjør det mulig å gjenspeile brukerens hender i applikasjonen, som gjør interaksjoner mer intuitivt. Den andre og viktigste grunnen til at vi valgte Touch-kontrollerne var fordi Oculus Quest-brillene garanterte støtte til dem[10]. Oculus Quest kom mest sannsynlig til å støtte klassiske spillkontrollere og, men vi hadde ingen garanti om det.

4.4.2 Styring og navigering i Unity3D

Mye av den generelle databehandlingen mellom kontrollerne og Unity3D fantes i Oculus-implementasjonen av OVRInput-klassen. Det vil si at vi enkelt kunne finne ut hvilke knapper, avtrekkere eller styrespaker som ble brukt til enhver tid. Bruk av store brukergrensesnitt ble håndtert ved å bruke posisjonen og rotasjonen til hendene for å sende ut laserpekere. Laserpekererne ble kun vist når brukeren fysisk pekte med hendene sine. Pekeren kunne så bli

fanget opp av og brukt av Unity3D sitt brukergrensesnittsystem. Vi valgte dette systemet for store grensesnitt over å bruke Touch-kontrollerne, fordi det gjorde valg av elementer i grensesnittene veldig intuitivt og naturlig, spesielt for brukere som ikke har tidligere erfaring med spillkontrolllere.

For å fange opp peking brukte vi en modifisert versjon av OVRInputManager-klassen, som originalt brukte VR-brillene i stedet for hendene som musepeker. Her oppsto det problemer der det originale komponenten kun håndterte en musepeker om gangen. Klassen ble utvidet til å håndtere flere musepekere, gjennom bruken av et Dictionary<Transform, EventData>. Dette gjorde det lett å hente ut hva en gitt musepeker siktet på, basert på musepekerens Transformkomponent (posisjon og rotasjon) i scenen.



Figur 14: Navigering av brukergrensesnitt med peking

4.4.3 Kontekstmeny

Kravspesifikasjonen sier at en bruker skal kunne flytte på og rotere på modeller som er lagt inn i scenen. For å implementere denne funksjonaliteten trengte vi et enkelt og intuitivt system som lot en kunde både velge og manipulere modeller.

For valg av modeller valgte vi å bruke den samme løsningen som for de store brukergrensesnittene våre, nemlig peking. For at en bruker skulle kunne samhandle med en valgt modell, måtte vi først gi dem informasjon om hvilke handlinger som var tilgjengelige. For å løse dette problemet, valgte vi å konstruere et system rundt grensesnittet "IInteractable". Alle objekter som skulle bli manipulert av brukere måtte implementere dette grensesnittet. Den viktigste metoden grensesnittet definerte var GetActions(), som returnerte en rekke av den abstrakte typen InteractableAction. Dette designet lot oss oppnå lav kobling, da håndmenyen ikke trengte å vite hvilket objekt som ble manipulert; bare om hvilke handlinger som skulle være tilgjengelige. Dette systemet ga oss også veldig høy kohesjon siden hver handling var en egen klasse, og hvert objekt bestemte hvilke handlinger som skulle være lov å bruke på det.

Ideen med InteractableAction-klassen var å ha et grensesnitt med overkjørbare metoder som ville bli kalt ved tilstandsendringer i kontekstmenyen. Dette lar handlingsobjektene definere

hva som skulle skje hver gang en handling ble valgt, kansellert, fremhevet osv. Baseklassens konstruktør krever kun et navn som vises i kontekstmenyen. I denne applikasjonen brukte vi hovedsakelig fem InteractableAction-klasser, som alle manipulerer det valgte IInteractable objektet på forskjellige måter:

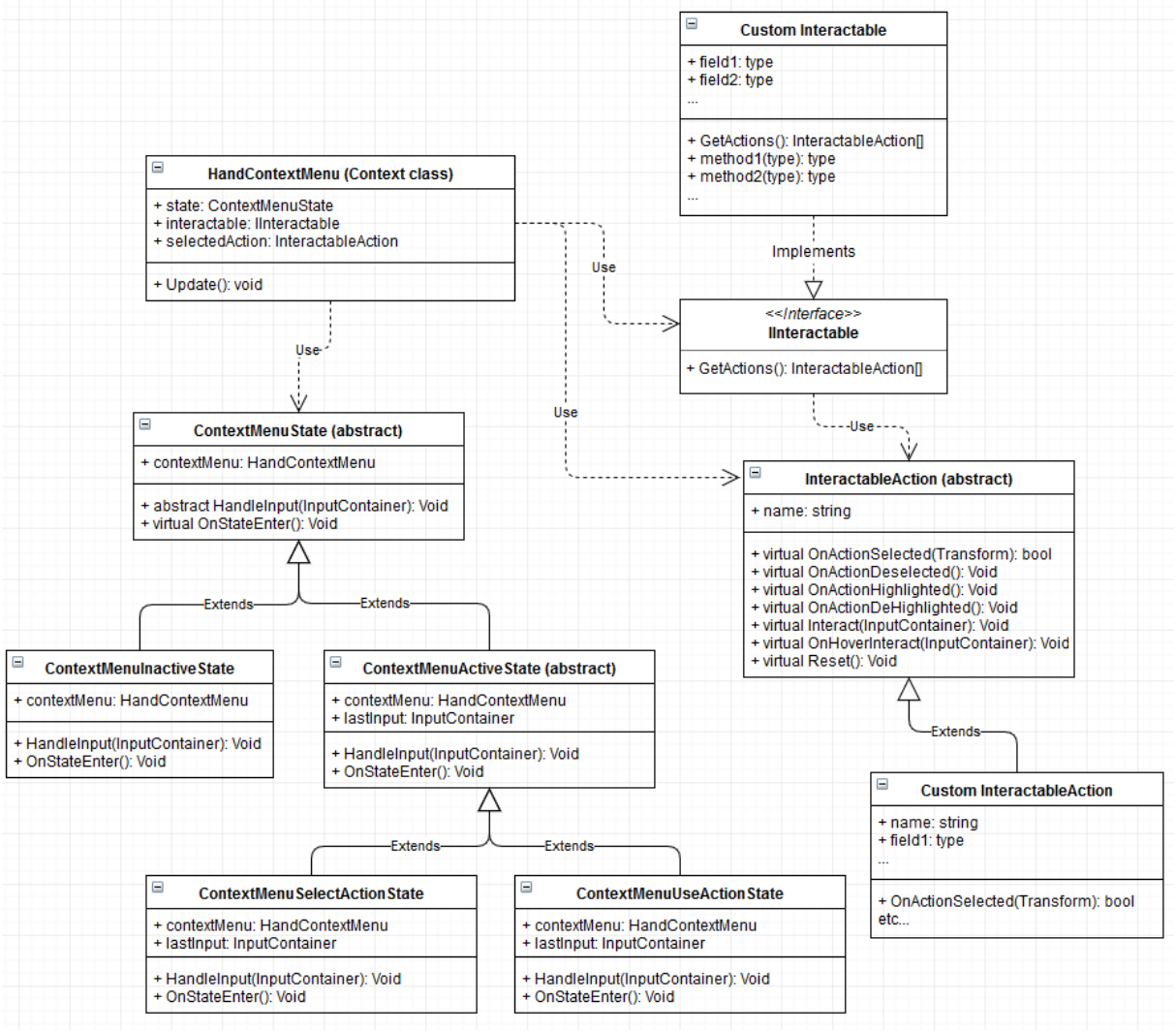
- DeleteAction - krever grensesnittet IDeletable. Sletter objektet.
- TranslateAndRotatePhysicsAction – handling som lar brukeren flytte objektet ved å definere en målposisjon med en gitt distanse fra der brukeren peker. Istedenfor teleportering, bruker denne handlingen hastighet. Dette lar objektet som blir flyttet på bli stoppet av andre objekter med kollisjonsrammer.
- ScaleAction – Lar brukeren endre størrelsesfaktoren til en modell ved å bevege styrespaken til håndkontrolleren opp eller ned.
- ExplodeAction – Lar brukeren endre eksplosjonsfaktoren til en modell ved å bevege styrespaken til håndkontrolleren opp eller ned. Hvordan eksplosjonsfaktoren fungerer kan du lese om i 4.5.5
- SpawnAction – En handling som viser en liste over objekter tilgjengelig for plassering i scenen. Mens handlingen er fremhevet i håndkontrolleren vil det vises et midlertidig objekt i scenen for alle brukere. Det midlertidige objektet vil bli vist der handleren peker. Hvis handleren trykker på valgknappen mens et midlertidig objekt blir vist, blir det lagt til i scenen permanent. For en dypere forklaring av hvordan denne handlingen fungerer, se 4.5.3.1

For at objekter skal kunne bli manipulert av brukere gjennom IInteractable systemet, må vi videreføre inndata fra kontrolleren til det korrekte InteractableAction-objektet som er valgt. Dette viste seg å være et problem siden mye av funksjonaliteten som brukes til å navigere kontekstmenyen, også brukes til å manipulere objekter i InteractableAction objektene. Det gjorde det vanskelig å definere når inndataen skulle håndteres av kontekstmenyen, og når den skulle bli ignorert og sendt videre til en valgt InteractableAction.

Løsningen var å bruke tilstandsmønstret. Ettersom tilstanden til kontekstmenyen var implisitt bestemt av flere forskjellige variabler som måtte sjekkes, kunne vi heller implementere tilstandsmønsteret for å eksplisitt definere hvilken tilstand kontekstmenyen var i til enhver tid. Kontekstmenyklassen «HandContextMenu» fungerer som kontekstobjektet, og holder på en instans av den abstrakte tilstandsklassen ContextMenuState. ContextMenuState-klassen inneholder en referanse tilbake til kontekstobjektet i tillegg til en metode for håndtering av inndata fra kontrolleren og en overstyrbar metode for når tilstanden blir byttet til. Ut ifra denne abstrakte modellen har vi 4 tilstander representert i form av klasser:

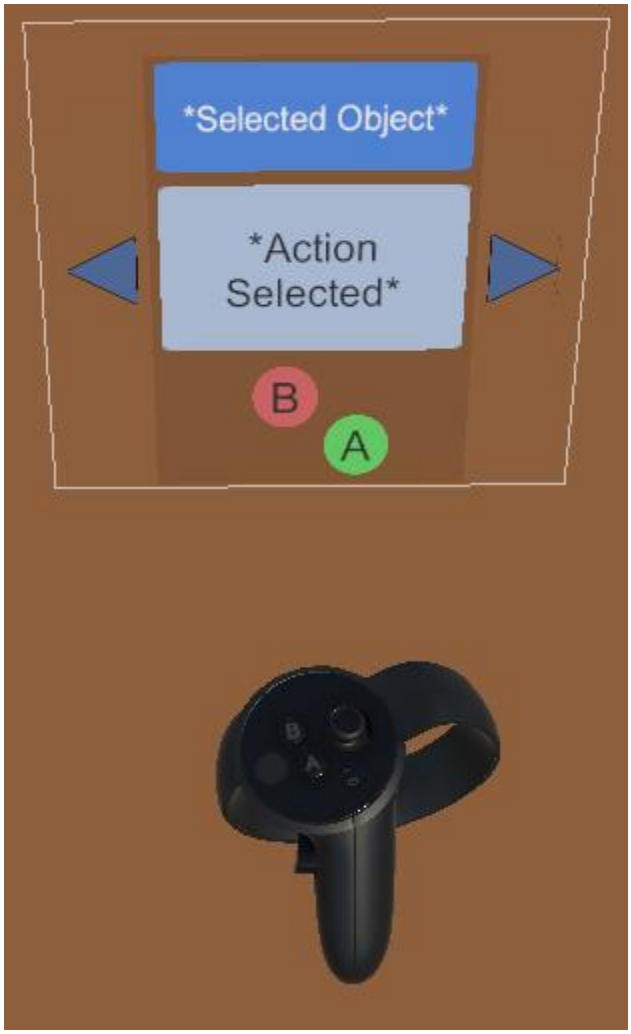
1. ContextMenuInactiveState – Representerer menyens inaktive tilstand. Når menyen er deaktivert.
2. ContextMenuActiveState – En abstrakt klasse som inneholder felles funksjonalitet for aktive tilstander når menyen er aktivert.
3. ContextMenuSelectActionState – Tilstanden der brukeren kan bla mellom og velge ønsket handling fra det valgte IInteractable objektet.
4. ContextMenuUseActionState – Tilstanden der brukeren manipulerer det valgte objektet gjennom den valgte handlingen.

I kontekstobjektet vil den interne tilstandens HandleInput(InputContainer) metode bli kalt hver oppdatering av applikasjonen, der InteractableAction-instansen selv bestemmer hva den vil gjøre med inndataen fra kontrolleren. Denne arkitekturen gjør det meget lett å kontrollere kontekstobjektets adferd ettersom at tilstanden er gitt eksplisitt, og ikke er definert som en samling av spesifikke variabler. Dette gjør også systemet meget lett å utvide ved å definere nye tilstandsklasser, med sin egen funksjonalitet.

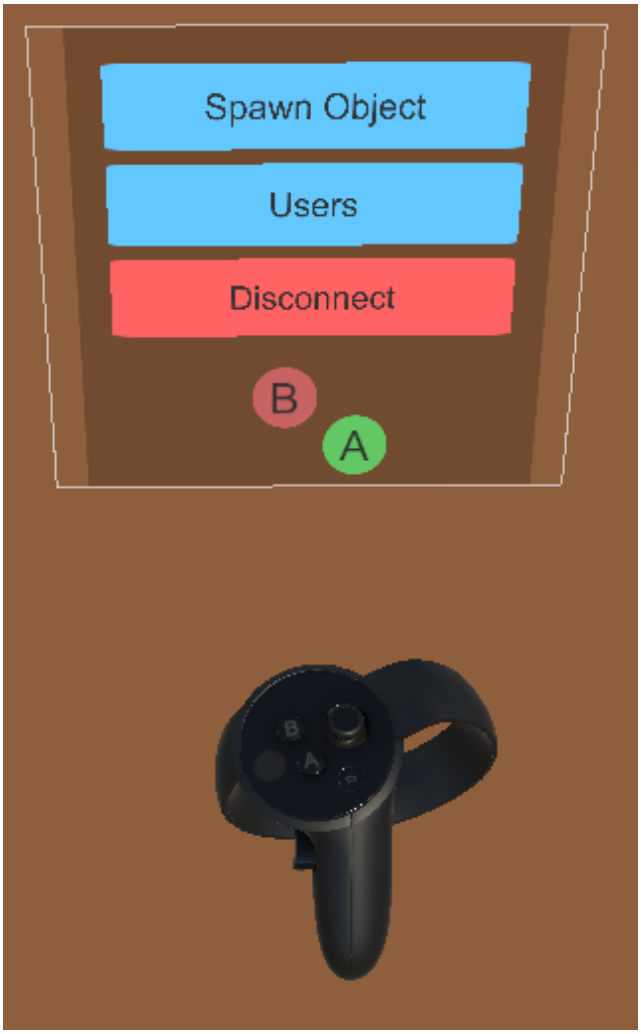


Figur 15: UML diagram for kontekstmenyarkitekturen

Brukeren behøvde også funksjonalitet for å velge den ønskede handlingen til et IInteractable-objekt visuelt. Den første løsningen vi prøvde ut var å lage en form for klokke som befant seg på brukerens hender. Ideen var at siden peking er intuitivt for å velge objekter, må handlingen av å «se på klokken» også være intuitiv for å se på og velge handlinger med det valgte objektet. En versjon av denne løsningen ble designet og implementert, men det viste seg etter utprøving (av utviklerne) at å navigere en meny i en slik positur var noe ukomfortabelt. Det ble heller valgt å lage en virtuell skjerm som blir vist over hånden når et objekt blir valgt. Dette viste seg å være den bedre løsningen siden man kan holde hendene i en mer nøytral stilling. Vi valgte også å gjenspeile kontrolleren i den virtuelle verdenen, slik at brukerne kan se virtuelt hvilke knapper de trykker på. Dette ville vært vanskeligere om menyen hadde fungert som en klokke.



Figur 16: Kontekstmeny. Vises når man velger et objekt.



Figur 17: Hovedmenyen. Åpnes ved å trykke på en knapp på kontrolleren. (B i bildet)

Kontekstmenyen viser brukeren hvilket objekt som er valgt og hvilke handlinger som er tilgjengelige for det objektet, med piler som signaliserer at brukeren kan bla mellom handlingene. Menyen inneholder også to knapper. Disse er ikke funksjonelle knapper, men fargekoder for å signalisere til brukeren hvilken fysiske knapp som velger en handling (grønn), og hvilken som kansellerer en handling (rød). Knappenes bokstaver er også relative til kontrolleren, siden hver kontroller ikke har de samme knappene. Posisjonen på knappene ligger også relativt til hvor de befinner seg på den fysiske kontrolleren.

4.4.4 Modellfremkalling i scenen

Før vi fikk ideen om å bruke en håndmeny, var planen å ha 3D-objekter i scenen med knapper som du trykket på med håndkontrolleren. Ved rommets oppstart ble det skapt ett såkalt "Box summoner"-objekt for hver modell mesterklienten hadde. Hver "Box summoner" hadde flytende tekst som sa navnet på modellen den kunne legge til i scenen, i tillegg til å vise prosentvis importfremgang. (import skjedde ved aktivering av boksen på dette stadiet i prosjektet, istedenfor å skje før man koblet til rommet.)

Siden hver "Box summoner" kun hadde ansvar for en modell måtte vi ha flere av disse spredt rundt i møterommet. Med bevegelsesbegrensningene som kom med VR ble dette kjapt irriterende, så vi trengte en mer brukervennlig løsning.

Istedenfor å prøve å utvide "Box summoner"-objektet til å kunne håndtere flere bokser samtidig ble det bestemt at vi skulle ha en håndmeny istedenfor. Fordelene med en håndmeny var at man slapp å bevege seg til boksen hver gang man skulle legge til en modell i scenen, og at en håndmeny kunne bli utvidet til å støtte flere funksjoner enn bare modellfremkalling, som for eksempel å vise en liste over tilkoblede brukere eller å koble fra møterommet.

Figur 18: "Box summoner". Første forsøk på et system for å legge til modeller i scenen.

4.4.5 Hovedmenyen

For å kunne bruke applikasjonen effektivt trengte brukeren tilgang til enkelte globale handlinger som for eksempel å koble fra og avslutte applikasjonen. Den første ideen vår var å bruke en stor virtuell skjerm som skulle vises foran brukeren der den ble åpnet. Skjermen skulle inneholde nyttige alternativer brukeren kunne velge blant. Denne versjonen ble aldri implementert siden vi kom på å bruke en kontekstmeny før vi hadde tid til det. To fordeler med å bruke håndkontrollerne var at vi kunne gjenbruke mye av funksjonaliteten fra kontekstmenyen, og at brukere slapp å forholde seg til to forskjellige typer brukergrensesnitt. For å åpne menyen må brukeren trykke på kanseller-knappen (Y/B). Når menyen er åpen kan man bruke styrespaken til å flytte markøren, valgknappen (X/A) til å velge, eller kanseller-knappen på nytt for å lukke menyen.

Hovedmenyen har 3 knapper: "Spawn Object", "Users" og "Disconnect". Vi kommer til å skrive om disse knappene og deres funksjoner i de neste avsnittene.

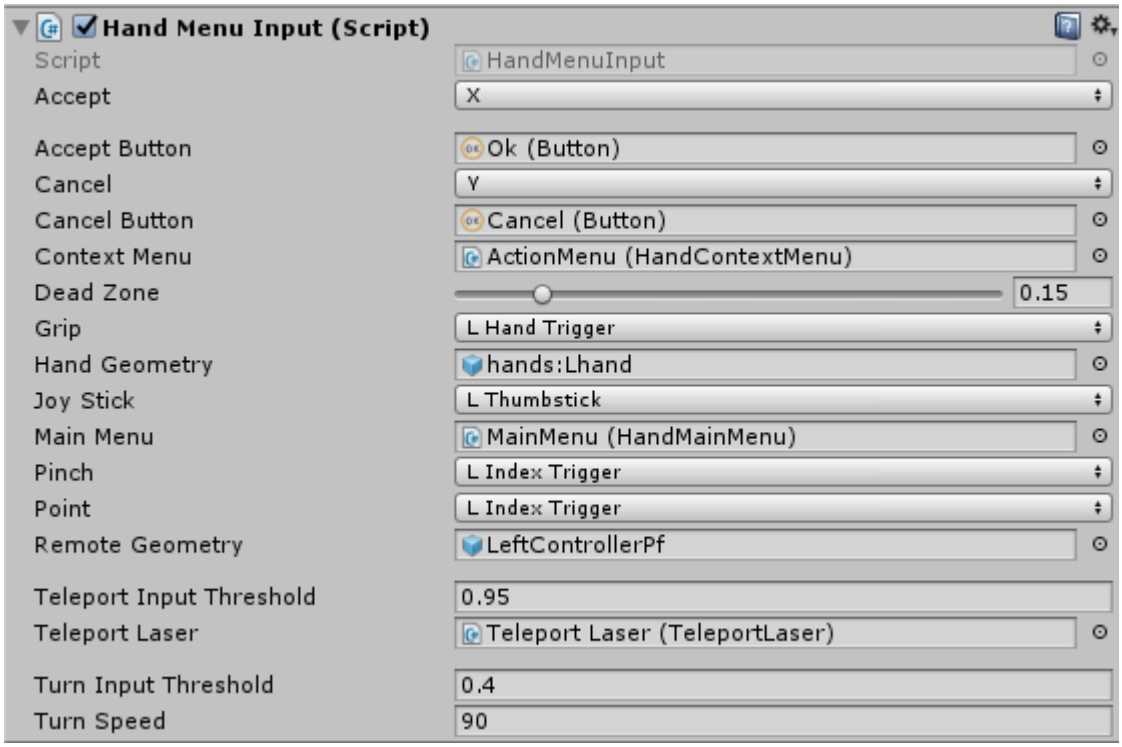
Hvis man velger "Spawn Object" knappen vil en kontekstmeny bli åpnet. Der kan man velge mellom mulige importerte modeller å legge til i scenen. Listen med tilgjengelige modeller er bestemt av mesterklienten, slik at brukere ikke kan opprette modeller andre klientene ikke har lokalt på sin maskin. I kontekstmenyen kan brukeren bla mellom de tilgjengelige modellene. Da vil singleton-instansen av CustomObjImporter bli valgt som IInteractable i HandContextMenu-objektet. Her vil de importerte modellene oppføre seg som handlinger brukeren kan velge mellom i kontekstmenyen. Mens brukeren blar mellom de tilgjengelige modellene, blir brukerens aktive modell forhåndsvist der brukeren peker ved hjelp av strålesjekking. Om brukeren så trykker på valgknappen, vil modellen bli lagt til i scenen permanent, og bli initiert med kollisjonsrammer og en liste over tillatte handlinger. Da blir eierskapet av modellen overført fra klienten som opprettet modellen til scenen, slik at de andre klientene kan samhandle med det.

"Users" knappen la vi til med planen om å få den til å vise en liste over alle møtedeltakere, samt å la brukere endre på volumet til individuelle deltakere. Dette var ikke en del av kravspesifikasjonen, og ble derfor nedprioritert kontinuerlig. Knappen gjør ingenting, men kan enkelt bli implementert senere takket være menysystemets modulære design.

Nederst er "Disconnect" knappen, som vises i rød farge. Denne knappen er meget enkel; den fjerner brukeren fra rommet og avslutter programmet. Andre brukere som er i møterommet vil se at brukeren har koblet fra rommet.

4.4.6 Menyfunksjonalitet i kode

Menyene skulle fungere på både venstre og høyre kontroller, noe som opprinnelig gjorde at menyklassene selv måtte sjekke hvilken kontroller som ble brukt i en gitt menyinstans. Dette var ikke en god løsning da begge menyene behøvde den samme koden. Kodeduplikasjon gjør kode vanskeligere å modifisere senere, så vi valgte heller å bruke en ny klasse, HandMenuInput. HandMenuInput fungerer som et grensesnitt mellom kontrollerne og menyene. Menyklassene har hver sin referanse til et HandMenuInput-objekt, som holder på nødvendig informasjon spesifikk til den gitte kontrolleren. På den måten har menyene tilgang til kontrollerspesifikk informasjon som hvilken knapp som er valgknapp, hva dødsonen på styrespaken er og hvor kontrolleren befinner seg. Om man ønsker, kan man lett endre på disse parameterne visuelt i Unity3D editoren (vist nedenfor). Dette gjør det enkelt å utvide/vedlikeholde hver enkelt håndmeny på et senere tidspunkt.



Figur 19: HandMenuInput komponenten i editoren.

Objektet fikk også som jobb å både håndtere handlinger som teleportering og rotering av brukeren, og å sjekke om en gitt kontroller/meny var ledig, slik at menyer og handlinger ikke kunne overlappe hverandre.

4.4.7 Teleportering og bevegelse

I en VR-applikasjon befinner brukeren seg i både et virtuelt rom og et fysisk rom. Det vil si at hvor langt du kan gå og bevege deg i møterommet er begrenset av rommet brukeren befinner seg i. Derfor er det viktig å la brukeren utforske og bevege seg gjennom den virtuelle

verdenen på andre måter. Vi valgte en relativt kjent løsning, der brukeren kan teleportere til et nytt punkt som vist av en kurvet bane styrt av hendene.

Figur 20: Visualisering av bane for teleportering.

For å tegne banen brukte vi en LineRenderer, en komponent som følger med Unity3D. Den trenger en liste med punkter for å vise hvor linjen går i tillegg til et materiale for å tegne linjen. For å finne alle punktene brukte vi en for-løkke som bruker Formel 1: veiformelen, med den iterative variabelen (i) ganget med en gitt tidsenhet. Om man endrer tidsenheten og/eller antallet maks iterasjoner vil man kunne endre hvor nøyaktig og hvor lang banen blir. For at man skal teleportere trenger man et endepunkt. Det kan man finne ved å kjøre en strålesjekk mellom hvert punkt på banen. Det er ikke nødvendig å sjekke hvert eneste punkt hvis nøyaktigheten av banen er høy. I stedet kan man bruke en heltallsvariabel som bestemmer hvor ofte strålesjekkingen skal foregå. Her bruker vi 2, altså at det blir strålesjekket mellom hvert andre punkt i banen.

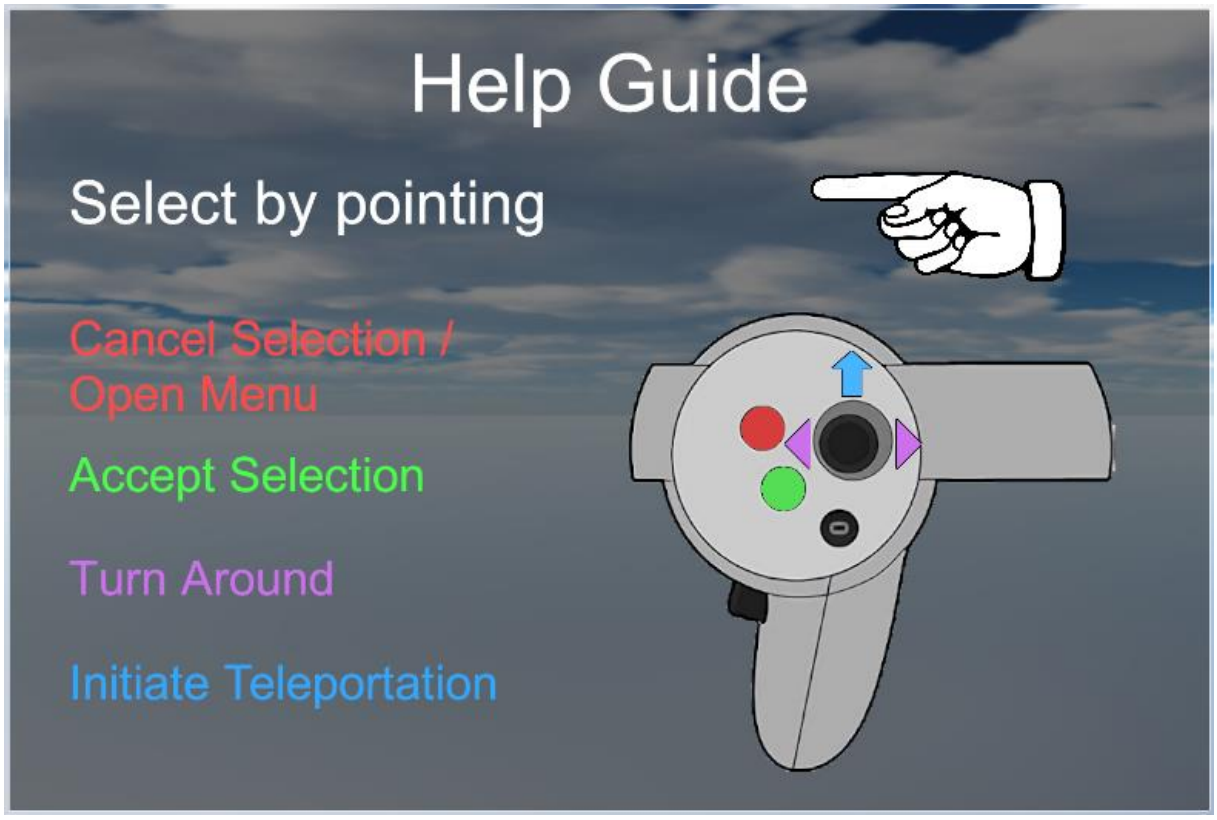
Brukeren burde også ha muligheten til å snu/rotere seg, uten å måtte gjøre det fysisk. Kablene til VR-brillene er den største grunnen til å vi trenger den funksjonaliteten, men sensorsynlighet var også et problem, da sensorene ikke kan se hvor hende er om brukeren dekker dem til med ryggen sin. Disse problemene vil bli forminsket når Oculus Quest-brillene kommer ut, da de er trådløse.

Det ble valgt å la brukeren rotere seg selv ved hjelp av styrespaken på Touch-kontrollerne. Denne funksjonaliteten ble kun gjort tilgjengelig når menytilstanden til HandMenuInput-objektet var ledig, slik at brukere ikke begynte å rotere mens de navigerte en meny.

4.4.8 Kontroller hjelpguide

For at man skal kunne kontrollere applikasjonen gjennom peking, rotering og teleportering, må brukeren vite at disse handlingene eksisterer. Vi trengte en måte for å informere brukerne om hvordan man styrte applikasjonen.

Det ble valgt å implementere en virtuell skjerm som befant seg til venstre for brukeren når applikasjonen startet. Både skjermen og dens elementer er store for å gjøre de enkle å se og lese. Siden applikasjonen bare brukte fem forskjellige handlinger, kunne vi vise dem på samme bilde uten å overvelde brukeren med informasjon. Handlingene er fargekodet for å kartlegge dem til de riktige knappene/styrespakene på kontrolleren. Fargene rød og grønn ble brukt til å signalisere «negative» og «positive» handlinger respektivt (rød for stop/avbryt/nei, grønn for start/gå/ok).



Figur 21: Hjelpeguide / kontrollerguide.

4.5 Import av modeller mens programmet kjører.

En av de største og viktigste utfordringene i dette prosjektet var støtten for import av egne modellfiler, som skulle kunne vises og synkroniseres over nettet med andre brukere. Unity3D har innebygd støtte for de fleste populære 3D-modell formater, men krever at filene blir lagt inn i prosjektet før kompilering. Vi trengte å importere modeller i sanntid, så Unity sin innebygde løsning kunne ikke brukes. Vi fikk tre testfiler av RUFO mens vi jobbet med denne funksjonaliteten, i tillegg til modellenes mål ned til millimeteren. RUFO brukte wavefront-filer (.obj-formatet), så vi måtte finne en løsning som fungerte for .obj-filer.

Det ble valgt relativt tidlig i prosjektets gjennomgang at vi skulle bruke en ressurs fra Unity Asset Store for modellimportering slik at vi kunne fokusere på resten av applikasjonen. Å utvikle vår egen løsning for filimportering hadde vært en veldig tidkrevende prosess og hadde mest sannsynlig resultert i en løsning som hadde vært verre enn det vi fant på Unity Asset Store. Ressursen vi valgte å bruke var det betalte biblioteket ObjReader av Starscene Software[13].

Biblioteket fungerte som reklamert, men dessverre kun på hovedtråden, så programmet frøs i ~10 sekunder hver gang man lastet inn en kasse. Problemet var at Unity ikke er trådsikker, så alle Unity-funksjoner må bli kjørt på hovedtråden. Dette inkluderer operasjoner som Oppretting, manipulering og sletting av GameObjekt-er, så vi måtte modifisere løsningen til å kun gjøre Unity-operasjonene på hovedtråden, og resten av prosesseringen i en bakgrunnstråd.

Løsningen ble å gjøre alle kalkulasjonene som trengs for å tolke en .obj-fil i en bakgrunnstråd, for så å lagre alle resultatene i et containerobjekt. Containerobjektet ble så sendt til

hovedtråden når bakgrunnstråden var ferdig. Da kunne vi bruke Unity-operasjonene til å skape GameObject-instansen og vise den i scenen. Dette reduserte fryseperioden fra ~10s til ~0.2s per modell.

4.5.1 Finne lokale modellfiler

ObjReader trenger å vite hvor filene som skal importeres ligger på den lokale maskinen. Vi valgte å bruke en mappe vi kalte «Custom», som vi plasserte ved siden av applikasjonens .exe fil. Det er i denne mappen applikasjonen vil lete etter filer som ObjReader kan importere.

En annen mulighet hadde vært å designe en filleser, for så å bruke den til å velge modeller å importere. Et brukergrensesnitt måtte blitt designet og implementert for å gjøre dette i VR, som ville vært svært tidkrevende å utvikle. Siden dette ikke var i kravspesifikasjonen, valgte vi å ikke bruke tid på det. Vi mener i tillegg at vår nåværende løsning med å manuelt plassere modellfiler i en mappe er intuitivt nok for RUFO å bruke.

4.5.2 Håndtering av importerte modeller

Etter at programmet hadde funnet filene som skulle importeres, trengte vi en overordnet protokoll for når og hvordan importering skulle skje. Da vi brukte "Box Summoner" løsningen (beskrevet i 4.4.4), startet hver knapp en egen importprosess og la den importerte modellen foran seg selv. Hver "Box Summoner" holdt på en referanse til den importerte modellen, slik at vi senere kunne bruke den referansen til å samhandle med modellen.

For å oppnå denne funksjonaliteten, sendte hver "Box Summoner" med en referanse til seg selv hver gang de kalte importmetoden i importeringskomponenten (ObjReader). Når importeringen ble ferdig, kalte importeringskomponenten en funksjon i en BoxSummoner, som registrerte modellen med den gitte BoxSummoner-instansen. Denne løsningen fungerte, men hadde unødvendig kobling mellom de to klassene. Vi reduserte koblingen senere ved å endre funksjons-kallet importeringskomponenten gjør fra å kalle en metode i BoxSummoner-instansen, til å bruke C# sin "delegate"-type. Med delegate-typen peker man på en metode med en gitt signatur istedenfor en spesifikk funksjon i en spesifikk klasse. Denne endringen var en forbedring da den lot et hvilket som helst objekt bruke importeringskomponenten så lenge de implementerte denne metoden, istedenfor at kun BoxSummoner-typen kunne bruke det.

En restriksjon med den nye løsningen var at kun objektet som startet importeringsjobben kunne motta referansen til modellen. Siden vi skulle gå over til å bruke en håndmeny, bestemte vi oss for å fikse den restriksjonen ved å bruke observatørmønsteret (beskrevet i 2.4.3). Observatørmønsteret har alle de samme fordelene som delegate-typen har, men tillater at flere objekter kan abonnere på hendelsene (eng: events) sendt ut av importkomponenten.

Det ble også valgt å bruke BackgroundWorker-objekter for å importere modeller istedenfor rene tråder. Fordelen med disse objektene var at de inneholdt funksjonalitet som gjorde det lett å håndtere startparametere for tråden, hendelser for fremgangsoppdateringer, og til slutt enkelt returnere resultatet til hovedtråden når bakgrunnstråden ble ferdig.

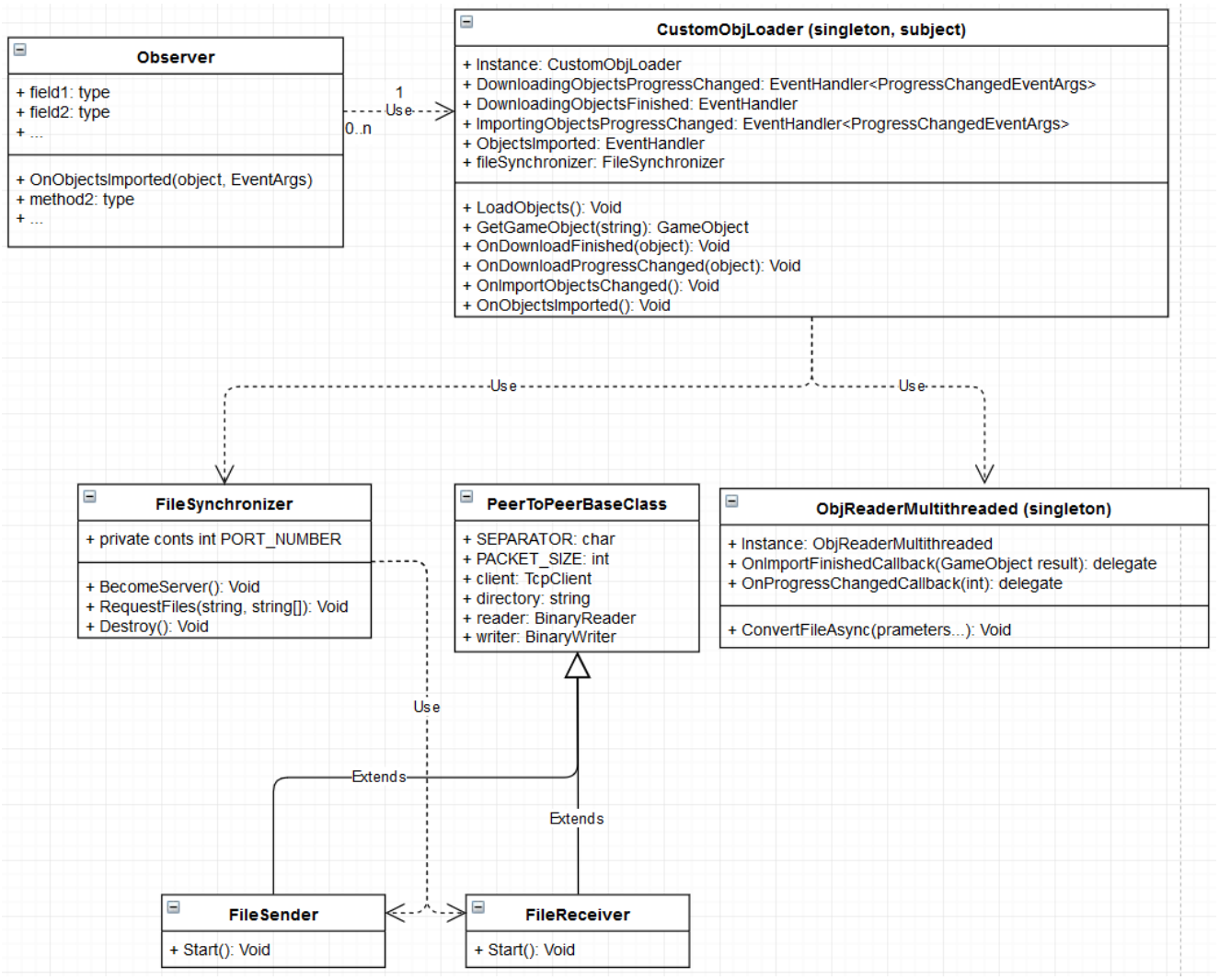
4.5.3 CustomObjLoader Singleton

Siden antallet modeller som blir importert til enhver tid er ukjent, ble klassen CustomObjLoader implementert for å ta ansvar for de importerte modellene. Dette gjorde den også til det perfekte valget for å være subjektet som behandler alle hendelser rundt

modellfiler. Mange objekter var avhengige av disse hendelsene, så det ble valgt å bruke singletonmønstret (se 2.4.1) for denne klassen. Dette gjorde instansen enkel å nå fra andre klasser, da den fikk et globalt tilgangspunkt. Om andre objekter ønsket å bli varslet hver gang en modell ble importert, kunne de enkelt abonnere på hendelser via den globale singleton-instansen.

For å opprette en singleton må man forsikre seg om at det bare finnes én instans av klassen til enhver tid. Unity3D gir deg ikke tilgang til konstruktøren for komponentobjekter, men gir i stedet såkalte Unity metoder som automatisk blir kalt om de finnes i koden. To av Unity metodene: Start() og Awake(), kan brukes til initiering av komponenter. Awake() blir kalt med en gang komponenten er laget, mens Start() først blir kalt når komponenten er aktiv[29]. Vi bruker Awake() til å finne ut når en instans av CustomObjLoader opprettes. Instansen sjekker da om en global variabel er null. Om variabelen er null, så settes variabelen til å peke til denne instansen. Hvis ikke, blir denne instansen ødelagt. Dette forsikrer oss om at bare en instans av klassen kan eksistere til enhver tid.

Unity metoden Start() blir også brukt. Når Start() blir kalt vil instansen sjekke Custom-mappen for .obj-filer å importere. Disse filene blir lagt inn i en liste som blir sendt videre til ObjReader-komponenten, som utfører selve importeringsjobben med BackgroundWorker-instanser. Når et BackgroundWorker-objekt er ferdig, sender den en instans av ObjContainer tilbake til hovedtråden gjennom BackgroundWorkerOnComplete-hendelsen. Med det har Unity3D all nødvendig informasjon for å generere GameObject-instansene i scenen. Siden objektene fortsatt må opprettes på hovedtråden, vil applikasjonen fryse ~0.2s, noe som er en betydelig forbedring fra ~10 sekunder. De genererte GameObject-instansene blir lagret i en inaktiv tilstand slik at brukere kan kopiere disse inn i scenen når de ønsker, uten å måtte importere filene på nytt.



Figur 22: UML diagram for håndtering av modellfiler

4.5.3.1 CustomObjLoader som IInteractable

Siden CustomObjLoader-instansen håndterer all funksjonalitet rundt de importerte modellene, kunne vi igjen bruke dette objektet hver gang brukeren ønsket å legge til nye modeller i scenen. Vi hadde allerede et system for manipulasjon av modeller i form av kontekstmenyen til håndkontrollerne, så ideen var å bruke det samme systemet for også å opprette nye modeller.

Systemet er avhengig av en IInteractable som forsyner kontekstmenyen med mulige valg brukeren kan bla mellom. Det ble derfor valgt å la CustomObjLoader arve fra IInteractable grensesnittet. For at CustomObjLoader skal returnere en rekke med modeller som brukeren kan legge til i scenen, behøver vi en ny type som arver fra InteractableAction klassen; SpawnAction. SpawnAction krever to parametere: navnet på boksen den representerer, og en referanse tilbake til en CustomObjLoader-instans.

I praksis vil det nye handlingsobjektet fungere slikt:

1. Brukeren velger «Spawn Object» fra hovedmenyen.
2. CustomObjLoader instansen vil bli brukt som IInteractable i kontekstmenyen.

3. Hele Rekken som blir returnert via GetActions() metoden vil være av den nye SpawnAction typen.
4. Det første elementet vil bli vist i kontekstmenyen som gjør at OnActionHighlighted() blir kjørt i SpawnAction objektet.
5. I denne metoden blir containerobjektet (CustomNetworkedBox) hentet fra CustomObjLoader-instansen, som bli brukt til å oppbevare en kopi av den valgte importerte modellen. Navnet på modellen blir sendt over RPC slik at de andre klientene kan se og oppdatere modellen som blir vist.
6. Om brukeren blar til neste handling, gjentas trinn 3-5.
7. Om brukeren trykker på valgknappen, vil modellen permanent bli lagt til i scenen, og en initieringsprosess blir startet. Etter at modellen er initiert kan andre klienter velge og samhandle med modellen via kontekstmenyen til håndkontrolleren.
8. Om brukeren trykker på kanseller-knappen, vil handlingen avbrytes, og containerobjektet bli slettet.

4.5.4 Kalkulering av modellens rammer for kollisjonsdeteksjon

Etter at en modell har blitt lagt inn i scenen, skal klienter ha mulighet til å velge og samhandle med det. Vi følte at den mest intuitive metoden å velge et objekt var å peke på det for så å klikke på valgknappen. Dette fungerer i praksis det samme som hvordan pek-og-velg funksjonaliteten fungerer for brukergrensesnitt i 4.4.2.

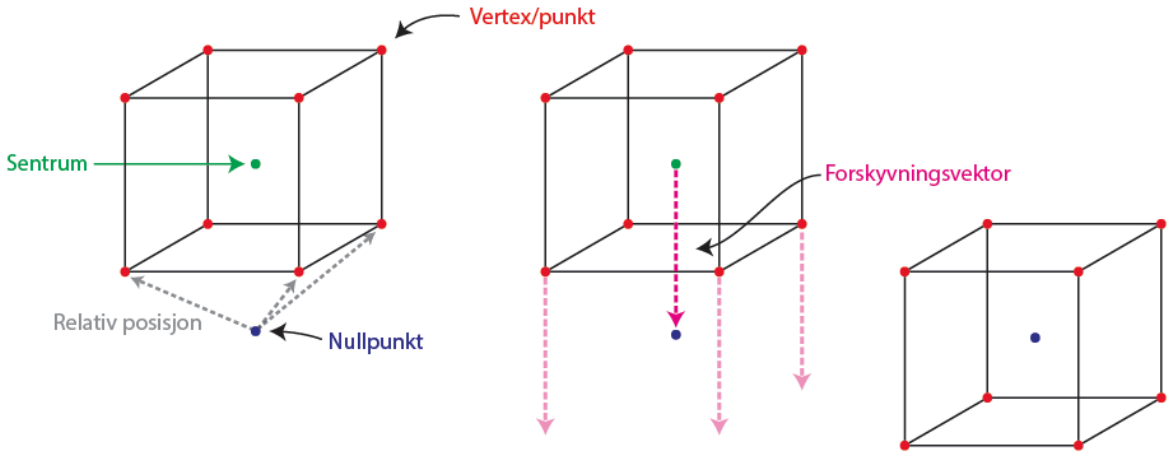
For at strålesjekkingen i pek-og-velg-systemet skulle fungere med modellene måtte vi gi dem kollisjonsrammer, som betyr at man må vite dimensjonene til modellen. For å finne dimensjonene må vi sjekke hvert punkt i 3D-modellen og lagre posisjonene som er lengst unna senteret til modellen for hver av hovedaksene (X, Y, Z). Siden vi allerede går gjennom alle punktene til modellene i importprosessen, valgte vi å utvide den til å også regne ut kollisjonsrammer. Kollisjonsrammer blir lagt til i ObjContainer-datastrukturen sammen med alle de andre kalkulasjonene som har blitt gjort. Når en modell blir lagt til i scenen, kan vi da opprette en BoxCollider-komponent med utregningene for å opprette en kubeformet kollisjonsramme som innkapsler modellen. At modeller har kollisjonsrammer gjør også at de føles mer virkelige for brukere av programmet, da det lar brukeren bevege objekter rundt omkring uten at de går igjennom gulv eller vegger.

4.5.4.1 Forskyvning av modellens nullpunkt

En tilleggsprosess som foregår samtidig som kalkulasjon av kollisjonsrammer, er å finne sentrum av modellen, basert på 3D-modellens punkter. Alle punktene på en OBJ-modell er definert som vektorer, relative til modellens nullpunkt/opprinnelsespunkt (2.6). Av og til vil 3D-modellen ha et nullpunkt som ikke ligger direkte i sentrum i modellen, slik som med modellene vi fikk fra oppdragsgiver. Dette viste seg å være et problem av to grunner:

- det gjorde plassering av importerte modeller inkonsistent siden posisjonen til 3D-modellen var relativ til nullpunktet.
- Når funksjonaliteten for å eksplodere boksene ble implementert, brukte submodellene sin relative posisjon for å oppnå effekten av å eksplodere modellen. Det vil si, eksplosjonens sentrum ligger ved modellens nullpunkt.

Derfor var det viktig å kunne omberegne og justere nullpunktet til modellen slik at vi kunne oppnå konsistent oppførsel for alle importerte modeller. Dette ble oppnådd ved å bruke den geometriske formen vi regnet ut tidligere for kollisjonsrammen. Volumstrukturen har en sentrumskomponent som er representert av en vektor som ligger relativt til alle punktene i modellen. Om man forskyver alle punktene i modellen denne relative avstanden, vil man forskyve nullpunktet til senteret av modellen.

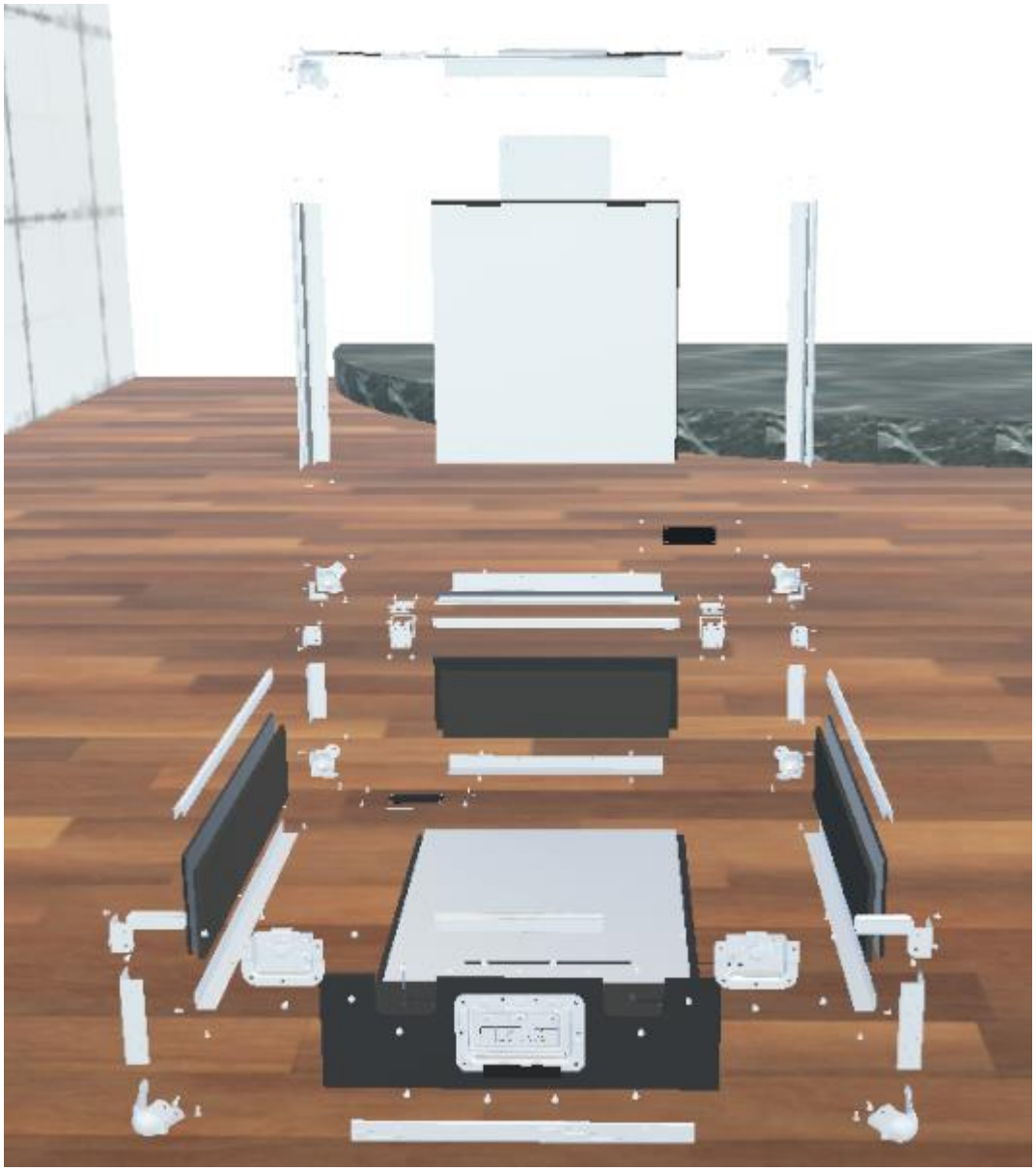


Figur 23: Forskyvning av nullpunkt i en 3D-modell

4.5.5 Eksplodering av modellen

Ifølge kravspesifikasjonen skal brukere kunne eksplodere de importerte kassene, slik at man enkelt kan inspisere hver individuell del en modell består av. Ettersom at vi kan velge og manipulere modeller gjennom IInteractable systemet, valgte vi å implementere denne funksjonaliteten som en InteractableAction, som brukeren kan velge gjennom kontekstmenyen. Siden systemet ble bygd opp med designavgjørelser som ga lav kobling og høy kohesjon, var det lettvint å utvide systemet med en ny handling. Det eneste som måtte legges til var et nytt grensesnitt som lot handlingen fortelle det implementerende objektet om å eksplodere en bestemt mengde.

For å oppnå effekten der modellen eksploderer kan vi bruke det faktum at modellens nullpunkt nå, alltid vil være i sentrum av modellen. Dette antyder at posisjonsvektorene til submodellene i 3D-modellen vil peke fra nullpunktet, til sin posisjon. For å eksplodere modellen behøver vi bare å multiplisere den relative posisjonsvektoren med en faktor over 1, som spesifiserer eksplosjonsmengden. For å finne de relative posisjonene til hver submodell i 3D-modellen, må vi først spesifisere at ObjReader-instansen skal lage nye GameObject-instanser for hver submodell. Da blir den relative posisjonen enklere å endre senere, fordi vi kan iterere over alle submodellene som GameObject-instanser. Når modellen har blitt importert kan man så gå gjennom denne listen med GameObject-instanser og hente ut og lagre den relative posisjonen til hver submodell i en ny containerstruktur. Denne strukturen holder på en referanse til submodellen og submodellens relative posisjon. Dette blir så lagret i hovedobjektet som enkelt kan traversere gjennom i en for-løkke, når eksplosjonsmengden skal endres.



Figur 24: En eksplodert modell

4.6 Filoverføring

For at brukere skal kunne importere modeller inn i scenen, må de ha dem tilgjengelig lokalt. Om en bruker ikke har de samme filene som mesterklienten, kan de derfor ikke bli med. Vi behøvde en filoverføringsløsning som lot nye klienter laste ned manglende filer.

En mulighet som ble vurdert var å sende filene over Photon. Dette er teknisk mulig, men ikke anbefalt der Photon er optimalisert for synkronisering via mindre pakker på rundt 1-20 kilobyte[26]. Siden testmodellene vi fikk var opptil 80 megabyte store, bestemt vi oss for å innføre vår egen filoverføringsløsning. Her ble det valgt å gå for en vert-til-vert-løsning siden de ikke behøver en ekstern server til å fungere. Vert-til-vert løsninger har noen ulemper, men fungerer bra for vårt formål.

Hvis RUFO hadde en egen server vi kunne bruke, hadde vi brukt den som en midlertidig filserver istedenfor å bruke en vert-til-vert-løsning. Fordelen med å bruke en server er at

mesterklienten ikke må sende de samme filene mer enn en gang, og at IP adressen til mesterklienten ikke trenger å være kjent hos de andre klientene.

Det er 3 ting man trenger for å overføre filer uten bruk av en server:

1. Klienten trenger IP-adressen til mesterklienten for å kunne etablere en forbindelse.
2. Mesterklienten må være åpen for innkommende forespørsler.
3. Man trenger en protokoll for selve filoverføringen.

4.6.1 Henting av ekstern IP

Photon lot oss ikke hente den eksterne IP-adressen til brukerne, så vi prøvde først å hente IP-adressen direkte fra PC-en lokalt. Det virket først som om det fungerte, men det viste seg at en av utviklerne hadde intern IP-adresse lik sin eksterne IP-adressen, så løsningen fungerte kun for dem. Dette kom antageligvis av at de var koblet til NTNU sitt eduroam-nettverk.

Vårt neste forøk på å hente IP-adressen til en klient ble gjort via en ekstern nettside, nemlig "https://api.ipify.org ". Denne nettsiden gir en klient sin eksterne IP-adresse som en string uten noe ekstra HTML. Mangelen på HTML gjorde denne løsningen meget rask.

En fordel med denne nettsiden over andre liknende nettsider er at man kan bruke den uten å bekymre seg for at den går ut av drift, siden infrastrukturen er bygget opp gjennom «Heroku»[55]. En annen løsning hadde vært å utvikle et verktøy som tilbyr liknende funksjonalitet som ipify.org og kjørt det på en RUFO-eid server. Dette hadde gitt RUFO full kontroll over verktøyet for loggføringsformål. RUFO hadde ingen server vi kunne bruke, så det ble ikke gjort i dette prosjektet.

4.6.2 Etablering av vert-til-vert TCP forbindelse

Etter å ha fått henting av egen ekstern IP-adresse til å fungere, var vårt neste trinn å opprette en forbindelse mellom nye klienter og mesterklienten. Det første forsøket vårt fungerte som følger:

Mesterklienten startet en serversocket i en ny tråd som andre klienter kunne koble til. Den brukte TCP og hørte på port 7836. TCP ble valgt for å garantere at filer ikke ble korrupte under sending. Port 7836 ble valgt ettersom at det er slik man staver RUFO på et t9 tastatur (eldre mobiltastatur). Ifølge Admin subnet[23] blir ikke denne TCP-porten brukt til noe. UDP-porten blir derimot brukt til Apple inc. sin "QuickTime Streaming Server", en applikasjon vi antar RUFO ikke har kjørende på samme maskin. Hver gang en klient koblet til serversocketen ble forespørselen deres håndtert i en ny tråd mens serversocketen fortsatte å høre etter nye forbindelser. Dette designet lot flere klienter be om manglende filer samtidig. Siden filene blir åpnet i skrivebeskyttet modus, kan mesterklienten ha flere filoverføringsprosesser som sender samme fil uten å bekymre seg over problemer som kappløpssituasjoner.

Hver gang en ny klient koblet til rommet ble de sendt IP-adressen til mesterklienten. Den nye klienten koblet så til mesterklienten og mottok alle filene som lå i mappen der mesterklienten oppbevarte modellfilene sine. Mens overføringen skjedde kunne man ikke legge til flere modeller i scenen, og hvis en ny klient koblet til etter at en modell hadde blitt skapt i scenen ville den vært usynlig for den nye klienten. Vi følte derfor at dette var et dårlig design, så det ble gjort noen endringer i tilkoblingsprosessen.

Det største hinderet til å forbedre løsningen var at mesterklienten ikke kunne sende IP-en sin til en ny klient uten at klienten allerede hadde koblet seg til rommet. Dette ble løst ved å la mesterklienten feste både IP-adressen sin og en liste over modellfiler som blir brukt i rommet til rommets egenskapsliste. Egenskapslisten et datalager som også er synlig for klienter utenfor det gitte rommet. På denne måten kunne nye klienter koble seg til mesterklienten og be om modellfilene de manglet uten å måtte koble seg til rommet først. Det siste problemet var hvordan vi skulle håndtere klienter som koblet til rommet etter at en eller flere modeller hadde blitt lagt til i scenen. Hvordan dette ble løst står i 4.3.4.

At IP-adressen til mesterklienten er offentlig tilgjengelig gjennom romegenskapene er en liten sikkerhetsrisiko, men ikke betydelig nok til å bekymre seg over i en prototype. Som vi skrev i 4.6, kan man unngå å vise IP-adressen til mesterklienten ved å bruke en server.

4.6.3 Filoverføringsprotokollen

For selve filoverføringen vurderte vi å bruke FTP, men det viste seg å være en unødvendig tung og overkomplisert løsning i forhold til hva vi hadde behov for. Vi valgte istedenfor å lage vår egen filoverføringsprotokoll. I dette avsnittet kommer vi til å referere til mesterklienten som "server" (ikke Photon-serveren), og klienten som ber om filer som "klient". Protokollen vår fungerer som følger:

1. Klient sender en liste med modellfiler den mangler.
2. Server finner alle filer relatert til disse modellfilene og plasserer dem i en liste. Den originale modellfilen er inkludert i denne listen. Alle filnavnene i listen er relative til modellmappen («Custom» som ligger i installasjonsmappen).
   For eksempel: "boks.obj", "boks.mtl" og "boks/tekstur.png".
3. Server sender hvor mange filer som kommer til å bli overført. (antallet filer som ble funnet i trinn 2)
4. Server sender det relative filnavnet til en fil den skal sende.
5. Server sender hvor mange pakker filen er splittet opp i. Hver pakke er 1024 Byte (1 kB) stor.
6. Server sender alle pakkene for den gitte filen.
7. Klient lagrer filen midlertidig i minnet som en byte-rekke. Denne rekken blir plassert i en ny liste som representerer alle filene som er mottatt. Dette blir gjort slik at filoverføringen ikke skal ta unødvendig lang tid, som skjer hvis vi må vente på at klienten skriver hver fil til disk etter mottakelse.
8. Trinn 4-7 blir gjentatt til alle filer har blitt sendt. Server lukker så forbindelsen og avslutter sendetråden.
9. Klienten lagrer filene den mottok til modellmappen under samme navn som serveren sendte tidligere. Dette gjør filnavnene konsistente mellom klientene, slik at vi kan referere til modellene med navn når de opprettes i scenen gjennom RPC.
10. Klienten lukker mottakertråden og kobler til rommet.

I vår originale løsning kunne ikke klienten spørre om spesifikke modeller; den fikk istedenfor alle filene mesterklienten hadde. Dette var ikke gunstig hvis klienten hadde

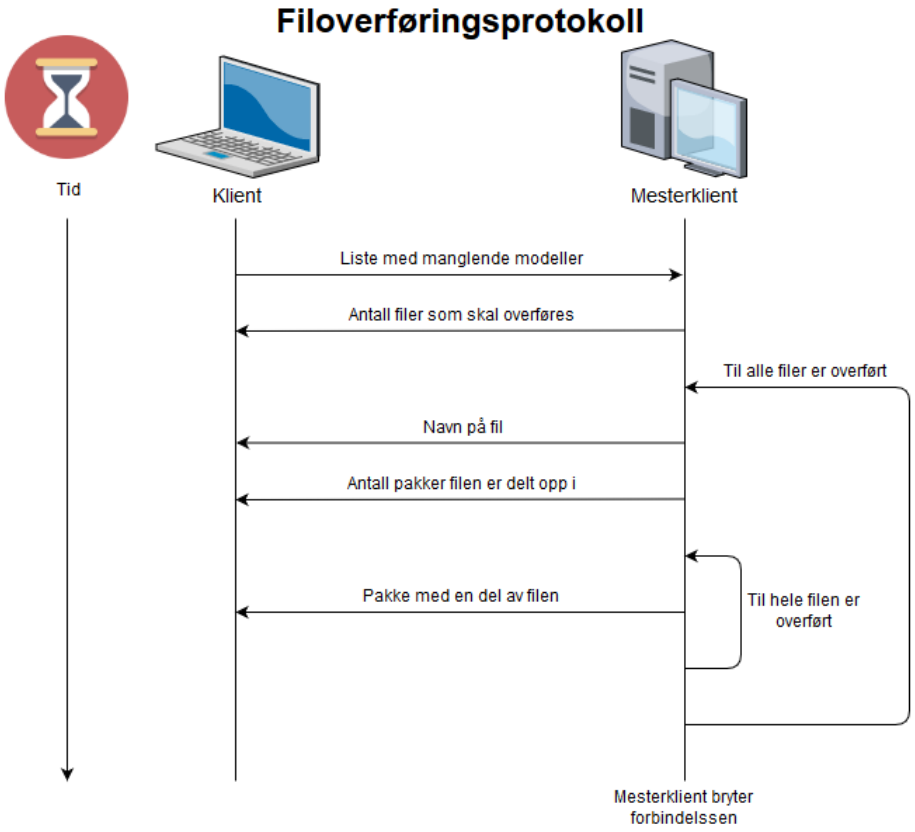
nettverksproblemer eller måtte koble seg fra rommet midlertidig av en eller annen grunn. Om de allerede har modellene, skal de ikke behøve å laste dem ned på nytt.

Den første iterasjonen av protokollen brukte også en overkomplisert løsning for å kode og dekode datastrømmen. Denne løsningen var inspirert av løsningen til "Lolmewn" på Stackoverflow[1]. Løsningen ble senere endret til å pakke inn nettverksstrømmen i en BinaryReader og en BinaryWriter, som koder og dekoder datatypene på en mer effektiv måte.

Mot slutten av utviklingsperioden fant vi ut at filer ble korrupte hvis de ble sendt over en ustabil forbindelse. For å prøve å fikse feilen ble verktøyet Clumsy brukt for å teste om pakkene ble korrupte utenfor applikasjonen. Det viste seg at ingen form for tukling hadde noen betydning for om filene ble korrupte eller ikke, så feilen måtte komme fra applikasjonen. Det viste seg at den originale måten filene ble lagret på var problemet.

I første versjon ble hver fil lagret som en liste med byte-rekker. Ved å endre løsningen til at hver fil ble lagret som en stor byte-rekke istedenfor en liste med mindre byte-rekker, ble problemet fikset. Vi er fortsatt usikre på hvordan denne endringen løste problemet siden den originale versjonen fungerte lokalt selv med Clumsy sin tukling.

Her er en graf som beskriver filoverføringsprotokollen visuelt:



Figur 25: Filoverføringsprotokollen

4.7 Møteromdesignet

Med tanke på det visuelle designet bestemte vi oss for å bruke en enkel stil med trygge fargevalg og vanlige byggematerialer. Resultatet ble et firkantet rom med hvite vegger, parkettgulv i tre, og en scene med RUFO-logoen for å vise frem modeller. Belysning i møterommet var veldig viktig siden prosjektets formål var å vise kunder hvordan modeller så ut i detalj. Vi prøvde først lampebelysning, men fant ut at møterommet så bedre ut og føltes mer åpent ut uten et tak. Vi bestemte oss derfor for å la Unity sitt virtuelle dagslys for å ta seg av belysningen. I tillegg til å se bedre ut, lot denne endringen brukere plukke opp og kaste modeller ut av møterommet, som viste seg å være en underholdende og intuitiv måte å fjerne modeller på i midten av et møte. Photon-rom blir tilbakestilt etter at alle har forlatt det, så rydding er ikke noe RUFO må gjøre etter hvert møte. I fig. 28 kan du se møterommet fra utviklingsmiljøet. De svarte områdene blir som i fig. 27 når applikasjonen kjører, blått med skyer i himmelen.

Figur 26: Lobbyscenen brukeren starter i.

Figur 27: Møterommet sett fra Unity3D editoren.

5 DRØFTING

5.1 Valg av VR-briller

Ifølge kravspesifikasjonen skulle det utvikles en VR-applikasjon. VR er en relativt ny teknologi som lar brukeren oppleve en virtuell verden. Utvikling av VR applikasjoner er noe standardisert, med verktøy og utviklingspakker for de fleste populære VR-brillene. De to største merkene innen VR var HTC Vive med SteamVR og Facebook sin Oculus Rift da vi startet dette prosjektet. NTNU Ålesund hadde begge typene tilgjengelig. SteamVR fungerer som et slags grensesnitt til VR-briller, som vil si at de fleste VR-briller vil fungere, uansett merke. Det eneste kravet SteamVR stiller av plattformen den kjører på er at Steam er installert. Steam er en digital distribusjonsplattform som kjører på Windows, Mac og Linux operativsystemer.

Oppdragsgiver hadde nevnt i sin visjon at man skulle kunne sende et par VR-briller til en kunde som var interessert i et sett med produkter. RUFO og kunden skulle så kunne møtes i det virtuelle rommet for å diskutere og inspisere produktene i en 1:1 skala. Siden vi ikke kan anta at klientene til RUFO er teknisk begavet, må produktet være intuitivt. Både HTC Vive og Oculus Rift må kobles til en PC for å fungere, og den oppkoblingsprosessen kan ta ganske lang tid om man aldri har gjort det før. Et annet problem med disse VR-brillene er at datamaskinen til klienten også må være relativt kraftig for å kunne kjøre VR-applikasjoner. Siden vi ikke kan garantere dette, ville en selvstendig VR-løsning som har en konsistent ytelse og er enkel å sette opp fungere best.

Oculus har annonsert en ny VR-modell som lanseres våren 2019 ved navn Oculus Quest[10]. Det er en helt trådløs og uavhengig modell som er kompatibel med programvare utgitt for den tidligere modellen, Oculus Rift. Quest-brillene krever ikke å bli koblet til en PC, men har i gjengjeld redusert ytelse sammenliknet med de klassiske VR-brillene på en relativt god PC. Alt vi vet om Quest-en er at prosessoren dens er en Snapdragon 835[28], som er en relativt populær prosessor for mobiltelefoner. Siden Quest-brillene bruker Android operativsystemet, kan de ikke ta nytte av Steam og OpenVR, men må utvikles med Oculus sine egne utviklingsverktøy.

Det ble valgt å fokusere på Oculus Quest-plattformen, ettersom at den kom ønskene til oppdragsgiver mer i møte enn de andre alternativene. Siden Quest-brillene enda ikke hadde blitt utgitt under prosjektets gjennomgang, brukte vi Oculus Rift som utviklingsplattform. Dette vil gjøre overføring til Quest-brillene en lett jobb ifølge Oculus[10]. Utviklingsverktøyene fra Oculus finnes også til Unity3D, som vil hjelpe med oppsett av grunnleggende VR-funksjonalitet.

5.2 Unity3D som grafikkmotor

For å utvikle en VR-applikasjon må man skape en digital verden som kan vises gjennom VR-briller. Grafikk-/spillmotorer gjør det lett å lage virtuelle verdener der mye av grunnfunksjonaliteten som tegning av verdenen, fysikk og mye av rammeverkene allerede er implementert. Fordi prosjektet skulle utvikles på 3-4 måneder var bruken av et tredjepartsrammeverk nødvendig for å kunne nå målene satt av arbeidsgiver. Siden det ble valgt å bruke Oculus Rift som utviklingsplattform, kunne vi bruke førsteparti utviklingsverktøy skrevet i C++, eller tilleggsmoduler til to av de største spillmotorene: Unreal Engine 4 av Epic games, og Unity3D av Unity Technologies. Unity3D bruker C#, som likner Java; et språk begge gruppemedlemmene har erfaring i. Unity3D er også generelt mye brukt for utvikling av

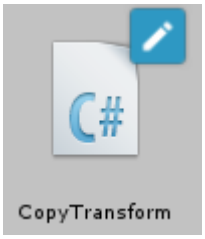
interaktive VR-applikasjoner[6]. Unreal Engine 4 bruker C++, og er også mye brukt for interaktive VR-applikasjoner. I tillegg til C++, kan man også bruke et visuelt programmeringsspråk kalt blueprints som lar utviklere modellere funksjonalitet ved å koble sammen noder[40]. Dette er også mulig i Unity3D om man kjøper en tilleggspakke fra Unity sin Asset Store[39].

Det ble valgt å bruke Unity3D i dette prosjektet, hovedsakelig fordi en av gruppemedlemmene allerede hadde erfaring med utviklings-miljøet, men også siden programmeringsspråket ligner Java, og derfor er lettere å venne seg til for de i gruppen som er nye til utvikling i Unity3D rammeverket.

5.3 Versjonskontrollsystem i Unity: Collaborate

En viktig del av programvareutvikling er å forsikre seg om at endinger i dokumenter og filer ikke bare blir lagret lokalt, men også synkronisert med de andre utviklerne. Versjonskontrollsystemer gjør det lettere å arbeide sammen, og gir utviklerne muligheten til å rulle tilbake prosjektet til en tidligere versjon om noe går galt. Endringer som blir gjort i prosjektet blir lagret lokalt og/eller over nettet for å enkelt kunne synkroniseres med andre utviklere. Med Unity3D hadde vi to hovedvalg: Git og Unity Collaborate.

Git behøver ekstra konfigurasjon for å fungere, for store deler av Unity-prosjektet består av ressurser i binærformat som Git ikke kan tolke. Dette kan løses ved å tvinge Unity til å bruke tekstfiler for ressursserialisering, men siden ressurser ofte blir rekompilert og endret på av Unity automatisk kan dette gjøre det vanskelig å finne relevante endringer som er gjort mellom opplastninger. Sammenslåing av disse filene kan også bli vanskelig hvis de genererte filene er forskjellige for forskjellige utviklere. Heldigvis blir Unity3D levert med et verktøy kalt Smart Merge[14], som kan automatisk sammenslå disse ressursfilene til semantisk korrekte filer.



Figur 28: Fil markert som endret.

Unity Collaborate er et versjonskontrollsystem som er integrert i Unity3D editoren. Verktøyet er gratis for inntil tre brukere med én gigabyte skylagring. Siden Collaborate er integrert i Unity3D kan den markere filer som er modifisert, flyttet eller slettet, direkte i editoren (se figur 8). Denne markeringen vil også bli vist for de andre utviklerne, bare med et annet symbol. Collaborate støtter ikke grener, så alle utviklere må jobbe på hva som ville vært mestergrenen i Git. Mangelen på grener gjør hyppige opplastinger til skyen farlige, da de kan påvirke arbeidet til de andre utviklerne. Om det oppstår konflikter må man velge mellom egen versjon av en fil, den nedlastede versjonen, eller å løse konflikten manuelt med et sammenslåingsverktøy. Hverken Unity eller Rider har innebygd sammenslåingsfunksjonalitet, så man må bruke et tredjepartsverktøy hvis dette ønskes.

Vi valgte å bruke Unity Collaborate i dette prosjektet. En av gruppemedlemmene hadde allerede erfaring i å bruke både Unity og Git sammen, og så derfor muligheten for en enklere og mer integrert løsning som mer attraktiv enn Git. Gruppen så på mangelen på grenfunksjonalitet som et kompromiss for enkeltheten av å bruke en integrert løsning som fungerer uten mye konfigurasjon. Det viste seg senere at grener er meget nyttige, da vi ofte klarte å overskrive endringene til hverandre med opplastinger.

Vi opplevde flere feil i Collaborate der systemet kunne låse seg om versjonssystemet mistet oversikten over lokale endringer. Problemet ble løst av å slette den korrupte oversiktsfilen, for så å la Unity generere den på nytt; men det hadde den uheldige bivirkningen av at Unity3D

deretter trodde at endringene hadde blitt lastet ned og integrert i den lokale versjonen, som forkastet de siste endringene gjort av de andre gruppemedlemmene.

I tillegg fant vi ut at hvis man endret navnet til en fil, ville Unity merke den som flyttet. Den ville ikke registrere eventuelle endringer som også ble gjort i filen, så vi måtte gjøre to opplastninger om vi endret både navn og innhold. Alle disse problemene førte til at mye tid som kunne ha blitt brukt på produktivt arbeid ble brukt på å jobbe rundt Collaborate. Om vi hadde gjort prosjektet om igjen hadde vi valgt å bruke Git, selv om konfigurasjonen hadde vært noe slitsom. Git er mer stabil, har grenfunksjonalitet og alle i gruppen har erfaring med det.

5.4 Valg av verktøy

Vi mener generelt at verktøyene vi valgte å bruke under utviklingsprosessen var gode valg da de økte produktiviteten vår betydelig. De fleste verktøyene vi valgte kan vi ikke tenke oss noen bedre erstatninger for. Det var dessverre et par unntak til dette, men heldigvis forstyrret de ikke utviklingen for mye.

Spesielt har utviklingsmiljøet Rider (3.1.6) vært meget hjelpsomt. Rider, lik andre utviklingsmiljø som Visual Studio, har kodefullføring som hjelper med å skrive kode gjennom hint og forslag basert på konteksten koden befinner seg i. Rider gjør det lett å finne dokumentasjon om en gitt klasse eller metode, i tillegg til alternative løsninger som den tror er bedre praksis enn det du har skrevet (med begrunnelse). Rider inneholder også kraftige reformateringsverktøy som gjør det lett å endre navn og signaturer over hele prosjektet. Verktøyet kan også vise hvor ofte klasser, grensesnitt og metoder blir brukt, som gir en god indikasjon til hvor sammenkoblet en gitt løsning er.

Det eneste verktøyet vi har vært misfornøyd med har vært Unity Collaborate, en Git-erstatning som er integrert i Unity. Vi opplevde mange problemer med Collaborate, som står i kapittelet over (5.3).

Vi brukte kun grunnfunksjonaliteten i JIRA som sprintorganisering, men med kun 2 personer på utviklingslaget følte vi ikke at det var nødvendig å bruke gruppesamarbeidsverktøy så mye, siden vi når som helst kunne sende hverandre en melding og dermed ha kontaktet alle utviklerne.

Det finnes sikkert bedre grafverktøy enn draw.io, men vi er begge vant til å bruke det, i tillegg til at det kjører i en nettleser istedenfor å måtte bli installert på våre egne PC-er. Det støtter også integrasjon med Microsoft OneDrive som gjorde det lett å sende og endre grafer om det var nødvendig.

5.5 Programstruktur og designmønstre

Det ble valgt som mål å bli bedre kjent med diverse designmønstre ved å bruke de under utviklingsprosessen.

Siden vi utviklet applikasjonen med Unity3D, måtte vi bruke komponentstrukturen som lå til grunn i Unity3D-rammeverket. For den som var uerfaren med Unity3D var dette en tung overgang fra tradisjonell objektorientert programmering, da Unity3D brukte en komponentbasert arkitektur som måtte settes opp grafisk i editoren istedenfor å skrives inn som kode. Hvis man ønsket å bruke et skript, måtte dette legges til i ett GameObject, som hadde egne koordinater i scenen. Dette aspektet med Unity3D-arkitekturen var vanskelig å

overkomme, men det hjalp at gruppen hadde en utvikler som allerede var erfaren og komfortabel med Unity3D-arkitekturen. En fordel med at Unity tvinger alle til å bruke deres komponentbaserte arkitektur er at de kan tilby bedre innebygd støtte for vanlige arbeidsoppgaver innenfor den arkitekturen. Dette lot oss raskt bli ferdige med grunnleggende funksjonalitet, slik at vi kunne begynne å jobbe på funksjonalitet spesifikk til vår applikasjon. En annen fordel med å bli tvunget til å bruke komponentsystemet var at vi fikk verdifull erfaring i å bruke komponentmønsteret og hvordan det påvirker programvare generelt.

CustomObjLoader er klassen ansvarlig for importering av modeller, funksjonalitet som er sentral i denne applikasjonen. Flere modeller er avhengige av denne klassen, så det var nyttig å gi den ett globalt tilgangspunkt. Det er 3 måter å gjøre en klasses metoder lett tilgjengelige i andre klasser. Den første er å la alle klasser som trenger en instans bruke konstruktøren til å lage sin egen instans. Denne løsningen hadde gjort det vanskeligere å bruke trådbassenger og gjort implementasjon av observatørmønsteret mer komplisert. Den andre måten er ved å kun bruke statiske klassemetoder og klassevariabler. Problemet med den metoden er at vi mister mye av fleksibiliteten man får av å bruke instanser. Den siste metoden var å bruke singletonmønsteret. Dette fungerte utmerket for hva vi trengte, og inneholdt ingen ulemper som påvirket dette programmet.

Siden CustomObjLoader ble brukt av flere klasser og skulle være en singleton, valgte vi å bruke observatørmønsteret. Dette lot oss danne et en-til-mange forhold mellom klassene, som reduserte kobling. Ikke overaskende, forenklet denne designavgjørelsen også alle senere situasjoner hvor vi måtte hente informasjon om importstatus, som i grafiske brukergrensesnitt.

Tilstandsmønsteret ble også brukt effektivt i dette prosjektet. For å håndtere interaksjon mellom de importerte modellene, ble systemet rundt IInteractable grensesnittet utviklet. Dette gjorde det lett for implementerende klasser å definere mulige handlinger klienten kunne bruke til å manipulere det valgte modellen. Dette blir vist gjennom kontekstmenyen, som lar brukeren bla mellom de mulige handlingene. Siden brukeren bruker kontrolleren både til å navigere kontekstmenyen, og til å bruke den valgte handlingen, måtte vi finne ut av hvordan inndataen fra kontrolleren blir behandlet, basert på hvilken tilstand kontekstmenyen befant seg i. Her brukte vi tilstandsmønsteret for å eksplisitt definere tilstanden hos kontekstmenyen, som gjorde det lett å presisere hvordan inndataen fra kontrolleren skulle bli tolket til enhver tilstand.

5.6 Ytelse

Med all VR-teknologi er det viktig å ha konsistent høy FPS for å unngå kvalme og desorientering[27]. 90 FPS er den anbefalte nedre grensen, så det er viktig at applikasjonen når denne grensen. VR-brillene låser FPS-en til applikasjonen til denne grensen, slik at man ikke bruker maskinressurser på å vise bilder som ikke kan vises til brukeren.

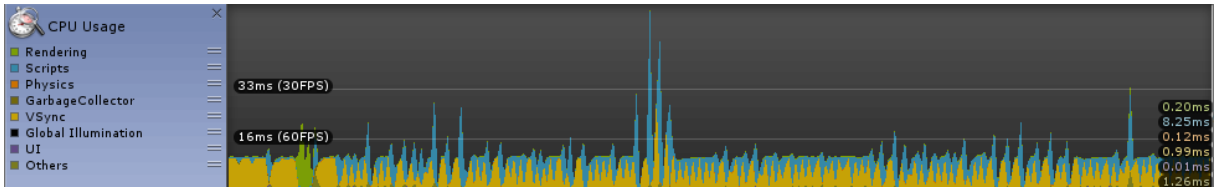
Da denne rapporten ble skrevet var det eneste vi visste om Oculus Quest at den skulle ha to 1600x1440 skjermer og at prosessoren skulle være en Qualcomm Snapdragon 835[28]. Å prøve å regne ut hvor høy FPS vi hadde fått med den prosessoren hadde vært meningsløst, så det beste vi kan gjøre er å dokumentere hvor bra programmet kjører på våre egne PC-er.

Datamaskin nr. 1:
GPU: NVIDIA GTX 1060
CPU: Intel Core i7-3770K @3.50GHz
RAM: 16GB DDR3 @1600 MHz
OS: Windows 7 64-bit.

Datamaskin nr. 1 hadde I snitt 98 FPS etter at programmet hadde stabilisert seg. Stabilisering skjedde ca. 20 sekunder etter at scenebytte ble gjort. Før stabilisering var FPS-en veldig variabel, og programmet frøs flere ganger i ~50ms av gangen. Grunnen til at FPS-en gikk over 90 var fordi programmet ble kjørt i Unity editoren istedenfor en kompilert versjon.

Datamaskin nr. 2:
GPU: NVIDIA GTX 1080TI
CPU: Intel Core i7-9700k @4.80GHz
RAM: 32GB DDR4 @3000 MHz
OS: Windows 10 64-bit

Under normale forhold, lå FPS på rundt 90. Under oppstart frøs applikasjonen i ~240 ms. ~50ms ved scenebytte, og holdt seg rundt 60 FPS under importeringen av de importerte boksene, med noen bunnpunkt ned i 30-15 FPS som varte i ~50ms:



Figur 29: Ytelsesgraf for datamaskin nr. 2

At applikasjonen fryser under oppstart føler vi er akseptabelt, siden det er så kortvarig. Dårligere og inkonsistent ytelse under importering av modellfiler kan også aksepteres, siden det kun skjer på starten av møtet.

Basert på hva vi har observert på egne PC-er tror vi programmet kommer til å klare å kjøre i 90 FPS på en Oculus Quest etter at programmet har stabilisert seg. Hvis noen spesielt kompliserte modeller blir lastet inn i scenen kan det endre seg, da kompliserte modeller øker arbeidsmengden til GPU-en betydelig.

5.7 Nettverksløsning

Nettverksløsningen er stabil, men ikke uten begrensninger. I dette delkapittelet kommer vi til å liste opp alle restriksjonene med løsningen vår i tillegg til mulige løsninger for dem.

5.7.1 Førstemann til å koble til blir mesterklient

Restriksjon:
En RUFO-ansatt må koble til rommet først for å bli mesterklient og å kunne sende modellfilene sine til de andre klientene. Hvis kunden kobler til først blir deres lokale modeller brukt istedenfor.

Mulig løsning:
Etter å ha implementert "Users" knappen i hovedmenyen (se 5.9.3), kunne vi implementert en måte å overføre mesterklient-statusen til en annen klient.

En annen mulighet hadde vært å sjekke hvilke klienter som bruker Oculus Quest og hvilke som bruker en vanlig Oculus Rift, for så å gi mesterklient-statusen til den som bruker Oculus Rift. Problemet med denne løsningen er at den krever at Oculus Quest briller blir sendt til alle møtedeltakere, uansett om de har VR-briller fra før av eller ikke. I tillegg hadde ikke RUFO kunne brukt Oculus Quest på deres side om de ønsket det.

5.7.2 Oppdatering av modelliste mens møtet er i gang.

Restriksjon:
Alle må gå ut og inn i rommet igjen hvis en ny modell skal legges til. Hvis en gammel fil blir endret på må klienten manuelt gå inn i modellmappen og slette/overskrive den gamle versjonen.

Mulig løsning:
Filoverføringsløsningen vår spør allerede kun om modeller den mangler, så hvis en ny fil måtte bli synkronisert kunne vi ha sendt ut en RPC til alle tilkoblede klienter og fått dem til å kjøre filoverføringsprosessen på nytt igjen. Klientene hadde da kun måtte laste ned den nye filen.

Re-synkronisering av modifiserte filer hadde krevd nye systemer, men kunne blitt gjort ved å lagre en hash-verdi for alle modellfiler for så å se etter endringer i hash-verdien med jevne mellomrom. Hvis en endring ble oppdaget kunne vi ha sendt ut en RPC som ba alle klienter slette den gamle versjonen av modellen og alle dens relaterte filer, for så å kjøre filoverføringsprotokollen på nytt.

Hvis mesterklient-statusen blir overført til en annen klient under et møte kan vi forsikre oss om at de eksisterende modellene fortsatt finnes hos den nye mesterklienten, men det er usikkert om den nye mesterklienten har modeller den forrige mesterklienten ikke hadde. Løsningen hadde vært å kjøre samme protokoll som når en ny modell blir lagt inn.

5.8 Utviklingsprosess

Det ble valgt å bruke scrum under utviklingen av dette prosjektet. Det ble planlagt at utviklerne skulle jobbe alle hverdager på prosjektet i ca. 4 timer hver dag. JIRA hjalp betydelig med å planlegge og organisere utviklingsprosessen over 2 ukers sprinter. Siden verktøyet kunne brukes gratis gjennom universitetet, og begge utviklerne hadde erfaring med JIRA fra faget *systemutvikling og modellering* (ID202712) var det lett å adoptere verktøyet inn i utviklingslagets arbeidsflyt.

NTNU Ålesund tilbød også Atlassian sitt samarbeidsverktøy Confluence. Det var planlagt å bruke denne programvaren sammen med JIRA for å samle all dokumentasjon og alle rapporter på ett sted, spesielt siden Confluence er integrert i JIRA. Dette ble ikke gjort på grunn av noen misforståelser, slik at det ikke ble opprettet en side i Confluence for prosjektet. Vi brukte istedenfor Microsoft OneDrive til å lagre alle dokumenter og relatert innhold, som vi følte fungerte bra nok for dette prosjektet. Siden prosjektets omfang var relativt liten, og utviklingslaget var besto av kun to utviklere, følte vi at enkelte verktøy og generell metodikk innenfor scrum ble overflødig. Scrum sin anbefalte lagstørrelse er på minst tre personer, så dette var ikke overraskende.

Det ble heller brukt kommunikasjonsverktøyet Discord for både kodediskusjoner og designvalg som foretok over tekst eller tale, basert på viktigheten og kompleksiteten av det som ble diskutert. Siden Discord også har syntaksfremhevning, ble kodediskusjoner enklere å gjennomgå. En ting vi savnet var støtte for genererte rapporter som kommer gjennom

integrasjonen mellom Confluence og JIRA. Om vi hadde gjort prosjektet på nytt hadde vi brukt Confluence.

Møtene våre ble ikke loggført, og standup-møtene ble noen ganger ikke gjennomført da vi følte det fungerte like bra å bare varsle den andre utvikleren om problemer oppsto eller noe var uklart. Denne løsningen ville antagelig ikke fungerte med et scrum-lag på en større skala, men for dette prosjektet viste det seg å fungere bra. Ellers har vi fulgt scrum-reglene ganske nøye. Kunden vår var ikke interessert i å endre på oppgaveprioriteringene våre etter å ha gitt oss kravspesifikasjonen, så vi diskuterte med oss selv for å finne ut hvilke oppgaver vi ville gjøre hver sprint.

5.8.1 Kommunikasjon med Produkteier

Siden vi brukte scrum-metodikken, var kommunikasjon med produkteier/oppdragsgiver viktig for å levere et optimalt produkt. All kommunikasjon med produkteier ble gjort via email, med unntak av noen telefonsamtaler på starten av prosjektet da vi diskuterte kravspesifikasjonen. Mot slutten av hver sprint ble det sendt en kort fremdriftsmelding som fortalte hva som ble gjort, med bilder eller video om det var relevant. Epostene inkluderte også våre planer for neste sprint i tilfelle RUFO ville omprioritere. Hvis vi hadde møtt på problemer som gjorde en del av kravspesifikasjonen ekstra tidkrevende, hadde vi informert produkteier om dette, men det skjedde heldigvis ikke under dette prosjektet.

Produkteier gav oss kun positive tilbakemeldinger, og ønsket aldri å endre på prioriteten til oppgavene våre underveis. Siden både testing og kjøring av dette programmet behøver et sett med Oculus Rift briller, ble det valgt å kun sende oppdateringer i form av tekst, bilde og video til produkteier. Vi vurderte å organisere for at produkteier skulle teste applikasjonen selv, slik at han kunne gi oss bedre og mer konstruktive tilbakemeldinger under utviklingsprosessen. Dette hadde krevd at de setter opp en datamaskin for å teste hver kjørbar iterasjon av applikasjonen, som ifølge kravspesifikasjonen også skulle kunne kjøre i de uavhengige Quest brillene senere. Vi mener at siden applikasjonen er en prototype som er ment til å utforske muligheten av modellvisualisering i VR, var tiden bedre brukt på å undersøke og utvikle løsningen i stedet for.

5.9 Videre arbeid

Her er forslag og ideer som kan utforskes for videreutvikling av prosjektet.

5.9.1 Kompatibilitet med Oculus Quest

Siden det viste seg at Oculus Quest-brillene som prosjektet var siktet oppimot ikke hadde blitt lansert i løpet av utviklingstiden, ville kompatibilitet med de nye brillene vært den største prioriteringen om vi skulle fortsette med prosjektet. Dette burde ikke være et stort problem siden Oculus nevner at overgangen fra Oculus Rift til Oculus Quest skal være enkel[10].

5.9.2 Støtte for flere filformat

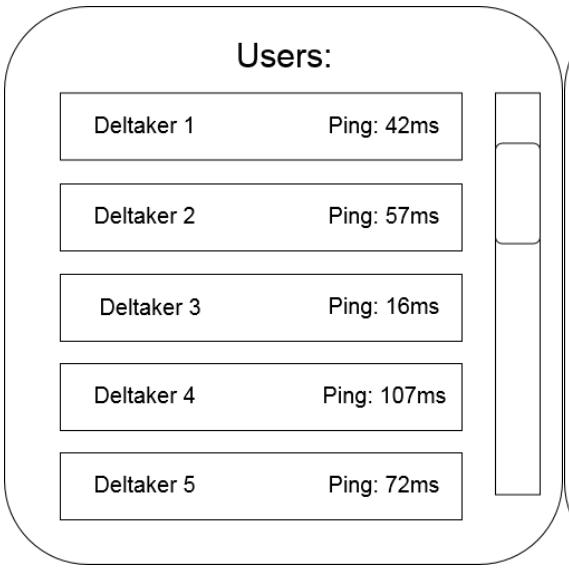
I denne oppgaven fokuserte vi på en type filformat: Wavefront (.obj). Dette ble gjort basert på testfilene som vi fikk under utviklingen av produkteier. Wavefront formatet er også støttet i de fleste 3D-applikasjoner, slik at de kan eksporteres og importeres i denne applikasjonen. Noe av grunnen til at dette er at formatet er gammelt og ikke har støtte for ekstra funksjonalitet som skjelett og animasjoner. Om vi hadde fortsatt å utvikle applikasjonen, kunne vi lagt til

støtte for flere filtyper, som hadde gjort det mulig å importere og vise animasjoner av modellene til brukerne. Møterommet hadde være mere imponerende om du f.eks. kunne åpne og lukke en boks.

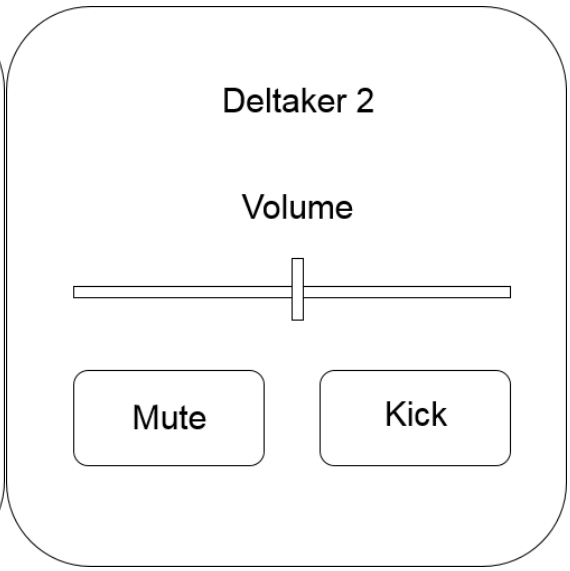
5.9.3 "Users"-menyen

Som nevnt i 4.4.6, ble det lagt til en "Users" knapp i håndkontrollermenyen uten at vi implementere funksjonaliteten til den. "Users" menyen er grunnleggende for å implementere funksjonalitet som volumkontroll for hver enkelt møtedeltaker, endring av mesterklient og evnen til å fjerne folk fra møterommet; så "Users" menyen hadde definitivt vært høyt prioritert om vi skulle fortsette med prosjektet.

Her er skissene vi lagde for "Users" menyen og menyen som skulle komme opp om man trykket på en enkelt deltaker i "Users" menyen.



Figur 30: "Users" menyen



Figur 31: Individuell deltakermeny.

5.9.4 Fargeblindstøtte

Videre kunne vi ha lagt inn støtte for en fargeblindmodus. Vi har brukt både grønn, rød og lilla i forskjellige brukergrensesnitt, og de er blant de vanligste fargene fargeblinde personer klager på. Det er to hovedmåter å fikse problemer som kommer av fargeblindhet. Den første er å redesigne brukergrensesnittene våre til å bruke farger fargeblinde personer ikke sliter med å se. Problemet med denne løsningen er at det påvirker personer som ikke er fargeblinde. Den andre løsningen er å la brukere konfigurere fargebruken selv. Denne løsningen fungerer best, men krever mye arbeid da man ikke bare må lage et grensesnitt for det, men også å implementere dynamiske farger basert på brukervalgene.

5.9.5 Hjelpeguide for Handlinger

Applikasjonen inneholder allerede en hjelpeguide som informerer brukeren om hvordan de styrer og navigerer applikasjonen generelt. En videre mulighet er å gjøre det samme i kontekstmenyen når en handling har blitt valgt av brukeren. Siden hver handling håndterer inndata fra kontrolleren forskjellig, hadde det vært nyttig å innføre en måte å informere brukeren om hvordan en gitt handling i kontekstmenyen brukes.

En mulig løsning hadde vært å utvide InteractableAction-baseklassen med en metode som returnerer en formatert tekststreng eller en egen struktur, som kan fortelle et brukergrensesnitt hvordan det skal vise informasjon til brukeren. Denne informasjonen skal kunne hjelpe dem med å forstå hvordan en gitt handlingen brukes. Dette kunne blitt vist i en ny virtuell skjerm som ligger ved siden av den originale kontekstmenyen, som kun vises når brukeren har valgt en handling. Her igjen kunne man brukt farger, størrelse og bilder for å gjøre brukergrensesnittet så intuitivt som mulig.

6 KONKLUSJON

Målet med prosjektet var å utvikle en prototype for å utforske et distribuert, virtuelt møterom i VR. Her skal RUFO og kunder kunne møtes og snakke sammen, samt inspisere og diskutere digitale representasjoner av bestilte produkter i et 1:1 forhold. Kunder skal kunne manipulere og inspisere modellene slik at de kan forsikre seg om alt er riktig før modellen produseres, og dermed slippe å bruke penger på feilaktige produkt. Vi valgte å løse oppgaven med Unity3D rammeverket sammen med Oculus Rift VR-brillene. Vi mener vi har klart å møte oppdragsgivers mål og ønsker i dette prosjektet, med ekstra funksjonalitet som gjør det lettere for både RUFO og kunder å sette opp og bruke.

Problemstillingen var klar og tillot fleksibilitet under utviklingen, siden oppgaven gikk ut på å utforske og utvikle en prototype. Dette gjorde det mulig å planlegge og utforske løsninger både for funksjonaliteten og selve designet til applikasjonen. Vi tok denne muligheten til å utforske designmønstre med fokus på ansvarsdrevet design, og mener resultatet reflekterer dette. Resultatet har veldig lav kobling og skal være lett for RUFO å utvide.

Mange av aspektene ved prosjektutviklingen har vært krevende, men svært lærerike. VR-utvikling (gjennom Unity3D) er noe vi har måttet lære fra grunnen av, som både har vært en interessant og utfordrende prosess. Vi har i tillegg lært utrolig mye om distribuert programvare, forskjellige designmønstre og deres effekt på systems sammenkobling. Spesielt kunne vi se fordelene ved å bruke komponentsystemet og tilstandsmønsteret, og hvordan de positivt påvirket applikasjonens arkitektur.

Vi er begge veldig fornøyde med resultatet og tror definitivt at applikasjonen vi har laget kommer til å hjelpe RUFO med å tilfredsstille kundene sine, og at RUFO kommer til å ta det i bruk når Oculus Quest blir lansert. Valget vi gjorde med å fokusere på Oculus Quest føler vi begge at var en utmerket avgjørelse som kommer til å sette RUFO på framfoten innen VR-integrasjon i business.

7 REFERANSER

[1] Lolmewn. (2012, januar 8). How to send a string over a socket in C#. Hentet 2. mai 2019, fra Stack Overflow: https://stackoverflow.com/questions/8773721/how-to-send-a-string-over-a-socket-in-c-sharp

[2] KrzysztofCwalina. (n.d.). Naming Guidelines. Retrieved April 30, 2019, from https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

[3] BillWagner. (n.d.). C# Coding Conventions - C# Programming Guide. Retrieved April 30, 2019, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

[4] David J. Barnes, & Michael Kölling. (2017). *Objects First With Java* (Sixth Global Edition). Pearson Education.

[5] Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st Edition). In *Computing Series* (1st Edition). Addison-Wesley Professional.

[6] Unity Public Relations Fact Page. (n.d.). Retrieved April 29, 2019, from Unity website: https://unity3d.com/public-relations

[7] Technologies, U. (n.d.). Unity - Manual: Creating and Using Scripts. Retrieved April 22, 2019, from https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

[8] Technologies, U. (n.d.). Unity - Manual: Order of Execution for Event Functions. Retrieved April 23, 2019, from https://docs.unity3d.com/Manual/ExecutionOrder.html

[9] Serialization in Photon | Photon Engine. (n.d.). Retrieved April 19, 2019, from https://doc.photonengine.com/en-us/realtime/current/reference/serialization-in-photon

[10] Oculus Developer Center | Quest Development. (n.d.). Retrieved March 29, 2019, from https://developer.oculus.com/quest/

[11] Technologies, U. (n.d.). Unity - Manual: VR overview. Retrieved April 3, 2019, from https://docs.unity3d.com/Manual/VROverview.html

[12] Oculus Integration - Asset Store. (n.d.). Retrieved April 3, 2019, from https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022

[13] ObjReader - Asset Store. (n.d.). Retrieved April 4, 2019, from https://assetstore.unity.com/packages/tools/input-management/objreader-152

[14] Technologies, U. (n.d.). Unity - Manual: Smart Merge. Retrieved April 30, 2019, from https://docs.unity3d.com/Manual/SmartMerge.html

[15] Technologies, U. (n.d.). Unity - Manual: Multiplayer Overview. Retrieved April 27, 2019, from https://docs.unity3d.com/Manual/UNetOverview.html

[16] Introduction | Photon Engine. (n.d.). Retrieved May 1, 2019, from https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro

[17] International Cross Platform Voice Chat as a Service | Photon Engine. (n.d.). Retrieved May 1, 2019, from https://www.photonengine.com/en-us/Voice

[18] draw.io. (n.d.). Retrieved May 1, 2019, from https://www.draw.io/

[19] Schweitzer, D. (n.d.). About us. Retrieved May 1, 2019, from draw.io website: https://about.draw.io/about-us/

[20] Doxygen: Main Page. (n.d.). Retrieved April 29, 2019, from http://www.doxygen.nl/index.html

[21] Atlassian. (n.d.). Jira Software - Features. Retrieved May 2, 2019, from Atlassian website: https://www.atlassian.com/software/jira/features

[22] Scrum Reference Card | Scrum Reference Card. (n.d.). Retrieved May 2, 2019, from http://scrumreferencecard.com/scrum-reference-card/

[23] Port 7836. (2016, mars 30). Hentet 2. mai 2019, fra Admin Subnet: https://www.adminsub.net/tcp-udp-port-finder/7836

[24] Regions | Photon Engine. (n.d.). Retrieved May 3, 2019, from https://doc.photonengine.com/en-us/realtime/current/connection-and-authentication/regions

[25] Wavefront OBJ: Summary from the Encyclopedia of Graphics File Formats. (n.d.). Retrieved May 5, 2019, from http://www.fileformat.info/format/wavefrontobj/egff.htm

[26] Frequently Asked Questions | Photon Engine. (n.d.). Retrieved May 6, 2019, from https://doc.photonengine.com/en-us/realtime/current/troubleshooting/faq

[27] The Importance of Frame Rates. (udatert). Hentet 6. mai 2019, fra IrisVR: http://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates

[28] Kyoto, K. (2018, September 27). Oculus Quest Specifications. Hentet 6. mai 2019, fra SizeScreens.com: https://www.sizescreens.com/oculus-quest-specifications/

[29] Delivering large-scale IT projects on time, on budget, and on value | McKinsey. (udatert). Hentet 7. mai 2019, fra https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/delivering-large-scale-it-projects-on-time-on-budget-and-on-value

[30] Arne Styve | Prosjektledelse i IT Prosjekter (24 jan. 2019). IF300114 Ingeniørfaglig systemteknikk og systemutvikling

[31] Manifesto for Agile Software Development. (udatert). Hentet 7. mai 2019, fra http://agilemanifesto.org/

[32] Scrum Guide | Scrum Guides. (udatert). Hentet 7. mai 2019, fra https://www.scrumguides.org/scrum-guide.html

[33] Interaction Design Basics. (2014, februar 19). Hentet 8. mai 2019, fra /what-and-why/interaction-design.html

[34] PROFESSIONAL CASING SINCE 1983 - Rufo.no - Professional Casing Since 1983. (udatert). Hentet 8. mai 2019, fra https://www.rufo.no/categories/bildearkivinspirasjon

[35] Short-term memory capacity: Magic number or magic spell? (udatert). Hentet 9. mai 2019, fra https://psycnet.apa.org/fulltext/1986-29151-001.html

[36] Tesler's Law. (udatert). Hentet 9. mai 2019, fra Laws of UX website: https://lawsofux.com/

[37] Hick's Law. (udatert). Hentet 9. mai 2019, fra Laws of UX website: https://lawsofux.com/

[38] Fitts's Law. (udatert). Hentet 9. mai 2019, fra Laws of UX website: https://lawsofux.com/

[39] Bolt - Asset Store. (udatert). Hentet 9. mai 2019, fra https://assetstore.unity.com/packages/tools/visual-scripting/bolt-87491

[40] Unreal Engine | Features. (udatert). Hentet 9. mai 2019, fra http://www.unrealengine.com/features

[41] clumsy, a utility for simulating broken network for Windows Vista / Windows 7 and above. (udatert). Hentet 10. mai 2019, fra https://jagt.github.io/clumsy/index.html

[42] Objective 1.1: Common Protocols. (2019, januar 26). Hentet 10. mai 2019, fra https://en.wikibooks.org/wiki/Network_Plus_Certification/Technologies/Common_Protocols

[43] April 23, U., September 24, 2019 / Posted, & Beal, 1999By Vangie. (2019, april 23). The 7 Layers of the OSI Model. Hentet 10. mai 2019, fra https://www.webopedia.com/quick_ref/OSI_Layers.asp

[44] TCP vs UDP. (udatert). Hentet 10. mai 2019, fra Diffen.com: https://www.diffen.com/difference/TCP_vs_UDP

[45] Kurose, Ross: Computer Networking - A Top-Down Apporach 6th edition, Pearson (2013), ISBN: 978-0-273-76896-8

[46] Norberg, A. (2012, august 20). uTorrent Transport Protocol. Hentet 10. mai 2019, fra BitTorrent.org nettside: http://www.bittorrent.org/beps/bep_0029.html

[47] McGath, G. (2013, mai 29). Basics of Streaming Protocols. Hentet 10. mai 2019, fra http://www.garymcgath.com/streamingprotocols.html

[48] Understanding How FTP Works. (udatert). Hentet 10. mai 2019, fra DeskShare nettside: https://www.deskshare.com/resources/articles/ftp-how-to.aspx

[49] Technologies, U. (udatert). Unity - Manual: Event System. Hentet 10. mai 2019, fra https://docs.unity3d.com/Manual/EventSystem.html

[50] Klement, S. (udatert). Blocking vs. non-blocking sockets. Hentet 10. mai 2019, fra https://www.scottklement.com/rpg/socktut/nonblocking.html

[51] Technologies, U. (udatert). Unity - Manual: Unity UI: Unity User Interface. Hentet 10. mai 2019, fra https://docs.unity3d.com/Manual/UISystem.html

[52] tutorialspoint.com. (udatert). Gorilla Testing. Hentet 15. mai 2019, fra www.tutorialspoint.com website: https://www.tutorialspoint.com/software_testing_dictionary/gorilla_testing.htm

[53] White Box Testing. (2010, desember 19). Hentet 15. mai 2019, fra Software Testing Fundamentals: http://softwaretestingfundamentals.com/white-box-testing/

[54] What is Adobe Illustrator? (udatert). Hentet 17. mai 2019, fra https://helpx.adobe.com/no/illustrator/how-to/what-is-illustrator.html

[55] ipify - A Simple Public IP Address API. (udatert). Hentet 19. mai 2019, fra https://www.ipify.org/

[56] DOTween (HOTween v2) - Asset Store. (udatert). Hentet 20. mai 2019, fra https://assetstore.unity.com/packages/tools/animation/dotween-hotween-v2-27676

VEDLEGG

Vedlegg 1	Forprosjektrapporten (nedenfor).
Vedlegg 2	Hele prosjektet (krever Unity 2017.4).
Vedlegg 3	Kjørbart program.
Vedlegg 4	Doxygen sin genererte dokumentasjon av kildekoden.
Vedlegg 5	Video som demonstrerer produktet (03:58)

Med unntak av forprosjektrapporten, ligger resten av vedleggene i en separat zip-fil.

Vi har ikke fått tillatelse til å distribuere modellfilene vi fikk av RUFO til å teste programmet med, så hvis leseren vil teste programmet selv må de laste ned modellfiler i .obj formatet selv. En lengdeenhet i modellfilen skal representere en millimeter i virkeligheten. .mtl filer er støttet.

Forprosjektsoppgave

BACHELOROPPGAVE

TITTEL:
VR Kassekonferanse - Forprosjekt

KANDIDATNUMMER(E):			
DATO: **22.01.19**	EMNEKODE: * **IE303612**	EMNE: **Bacheloroppgave Data**	DOKUMENT TILGANG: - Åpen
STUDIUM: **INGENIØR - DATA**		ANT SIDER/VEDLEGG: 11/0	BIBL. NR: - Ikke i bruk -

OPPDRAGSGIVER(E)/VEILEDER(E):
Knut Steinnes (Oppdragsgiver) Mikael Tollefsen (Veileder)

OPPGAVE/SAMMENDRAG:
Vi fikk i oppgave å lage et møterom i VR for RUFO, slik at de kan diskutere transportkassa med kunden før de begynner produksjon. Man skal kunne vise frem genererte 3D modeller i møterommet og manipulere modellen på forskjellige måter. Hovedmålet med oppgaven er å gjøre det lett for RUFO å snakke med kunder om produktet, slik at man unngår dyre produksjonsfeil.

INNHOLD

1 INNLEDNING

Forprosjektet er en del av ingeniørfaglig systemteknikk og systemutvikling, som vil beskrive hvordan gruppen vil planlegge og definere oppgaven.

Oppdragsgiver, RUFO, produserer transportkasser i alle størrelser som klienten kan bestemme. Av og til vil klienten bli misfornøyd med produktet de har bestilt. En mulig løsning er å sette opp et virtuelt møterom med klienten og en representant fra RUFO for å se og diskutere modellen i et 1:1 forhold før den blir produsert.

Målet er at kunden skal være enda mer sikker på at produktet de bestiller faktisk ser riktig ut, og det er lettere å visualisere i VR.
Denne rapporten er et forprosjekt som vil beskrive hvordan vi har planlagt å gjennomføre prosjektet.

2 BEGREPER

- **VR** – Virtual Reality. En virtuell 3D verden som kan bli opplevd gjennom et par med VR-briller.
- **SDK** – Software Development Kit, et digitalt verktøy som brukes til å skape programvare i ulike miljø. (Oculus SDK)
- **User-story** - noe klienten/prosjekt-interessenten/e vil ha inkludert (funksjonalitet) i prosjektet.

3 PROSJEKTORGANISASJON

3.1 Prosjektgruppe

Studentnummer(e)
Marcus Benjamin Johansson - 476514
Fredrik Foss – 460314
Disse er studentnummere, ikke kandidatnummere.

3.1.1 Oppgaver for prosjektgruppen – organisering

Gruppen kommer til å bestå av to utviklere. De skal ha like stort ansvar, slik at arbeidet med oppgaven blir så effektiv som mulig.

Vi har planlagt å bruke Unity sin innebygde versjonskontroll verktøy; Collaborate, og JIRA for å organisere utviklingen av prosjektet og problemer som må fikses o.l.

Prosjektet skal utvikles med "agile" metoden som JIRA er bygget rundt.

3.1.2 Oppgaver for prosjektleder

Prosjektleder tar seg av oppretting av sprinter og delegerer user-stories til gruppemedlemmene. Prosjektleder er hovedansvarlig for kontakt med arbeidsgiver.

3.1.3 Oppgaver for utviklere (alle)

Ansvar for å implementere funksjonalitet gitt ved delegerte user-stories/funksjonalitet innen gitte tidsrammer (sprinter-).

3.2 Styringsgruppe (veileder og kontaktperson oppdragsgiver)

Prosjektgruppe:

Fredrik Foss

Marcus Benjamin Johansson

Veiledere fra NTNU Ålesund:

Mikael Tollefsen

Oppdragsgiver (RUFO):

Knut Steinnes

4 AVTALER

4.1 Avtale med oppdragsgiver

Vi skal jobbe med prosjektet med tanke på kravspesifikasjon og sende fremskrittsrapport til kontaktpersonen hver sprint. Planen er å bli ferdig med prosjektet innen semesteret slik at oppdragsgiver blir fornøyd.

4.2 Arbeidssted og ressurser

Gruppen har en kontaktperson i bedriften som vil svare på alt vi lurer på angående prosjektet, ellers vil gruppen komme til å jobbe selvstendig. Vi har tilgang til VR lab og VR briller fra universitetet. Brillene kan også bli lånt og brukt hjemme.
Datasikkerhet/taushetsavtale er ikke en del av prosjektet. På slutten av hver sprint (to uker), vil vi sende et referat om hva vi har gjort til oppdragsgiver.

4.3 Gruppenormer – samarbeidsregler – holdninger

Gruppemedlemmene skal ha en god arbeidsmoral som vil gjøre arbeidet med prosjektet mer effektivt, samtidig som å pushe kvaliteten til sitt ypperste.

- Kvalitet: Gruppemedlemmene vil gjøre sitt beste for å oppnå et best mulig resultat, gitt prosjektets rammer.
- Punktlighet: Gruppemedlemmene skal respektere avtalte tidsrom, og gjøre sitt beste for å møte dem. Om sykdom eller andre hendelser kolliderer med gitte avtaler, skal dette gis beskjed om snarest mulig.
- Miljø: Gruppemedlemmene viser respekt mellom seg, og prøver å bidra til et godt arbeidsmiljø.

Oppgaven kan gi god innsikt i hvordan arbeidslivet vil utføre prosjekt. Vi som dataingeniører kommer til å jobbe sammen med andre, innenfor og utenfor vårt fagfelt, som vil bruke andre arbeidsmetoder enn vi er vant til. Det er viktig å respektere ulikheter og samhandle om problemer på en profesjonell måte.
Om konfidensielle resurser eller sensitiv informasjon blir delt med gruppen er det viktig å ikke bryte eventuelle avtaler man har med oppdragsgiver.

5 PROSJEKTBESKRIVELSE

5.1 Problemstilling - målsetting - hensikt

Når en klient bestiller en transportkasse fra bedriften kan det være vanskelig for kunder å se og finne alle detaljer av det bestilte produktet ved å se på en 3D-modell på en PC skjerm. Dette kan løses ved at bedriften og klienten møtes i et virtuelt møterom, der de kan se, diskutere og ikke minst samhandle med det bestilte produktet i et 1:1 forhold. Dette gjør det enklere for klienten å se eventuelle feil eller kortsiktigheter som er vanskelig å se, som vil eventuelt komme frem *etter* produksjon av kassene.

Gruppen vil undersøke mulige løsninger for å skape et VR-møterom som gjør det lettere for en eventuell klient å se og forstå hva de har bestilt, slik at de enklere kan se feil før produktet har blitt produsert. Oppdragsgiver har gitt ønske for at løsningen skal kunne bli brukt uten en dedikert datamaskin. Dette vil gi firmaet friheten til å sende et par med VR briller til klienten, slik at man kan holde et virtuelt møte om det produktet de har bestilt, hvor som helst i verden (med internettilgang).

Prosessmål

- Få mer erfaring og forståelse for hvordan arbeidslivet utfører et prosjekt.
- Bli flinkere til å samarbeide sammen i gruppen og med arbeidsgiver.
- Bli bedre i en "scrum" basert arbeidsmetode.
- Forbedre evnen til å skrive profesjonelle rapporter.

Effektmål

- Øke tilfredsheten til klienter av bestilte produkt.

Resultatmål

- En analyserende rapport om mulige løsninger angående problemstillingen.
- En prototype av det virtuelle møterommet.

5.2 Krav til løsning eller prosjektresultat – spesifikasjon

Arbeidsgiver krever en analyserende rapport om den gitte problemstillingen og en fungerende prototype.

Krav til prototype:

- Møte andre brukere i møterommet.
- Stemmekommunikasjon.
- Se, flytte og rotere en transport boks i et 1:1 forhold.
- Programvaren skal være rask og responsiv.

Ønsket funksjonalitet:

- Eksplodere kassemodellen slik at man se alle delene.
- Muligheten til å endre på kassen uten å gå ut av programmet.

5.3 Planlagt framgangsmåte(r) for utviklingsarbeidet – metode(r)

Vi vil først studere de forskjellige VR teknologiene og resursene vi har for å velge den mest egnende plattformen. Når det er tid for utvikling av prototype vil gruppen jobbe iterativt og "agile", gjennom en "scrum" basert arbeidsmetode. Dette gjør at produkteier/oppdragsgiver har innsikt inn i hvordan prosjektet utvikler seg, og kan komme med endringer fortløpende. IT prosjekter er uforutsigbar fordi vi mangler standardisering, noe som gjør planlegging av tid og ressurser vanskelig. Vi vil ikke komme unna uforutsigbarheten med agile, men det gjør prosjektet fleksibel nokk til å endre planene om prosjektet ikke rekker å bli ferdig.
Agile er en kjent og testet produksjonsmetode mye brukt i arbeidslivet i IT bransjen.

5.4 Informasjonsinnsamling – utført og planlagt

Med det første vil gruppen undersøke teknologi og verktøy som blir brukt til å løse liknende problemer i dag. I hovedsak vil vi vurdere de mest kjente teknologiene: Oculus og HTC Vive fordi de er mest kjent og at universitetet har disse tilgjengelig. Vi vet at grafikkmotoren Unity3D kan brukes av begge teknologiene, samtidig som at en av gruppemedlemmene har erfaring i grafikkmotoren. Ellers er VR nytt for medlemmene slik at gruppen må lære seg teknologien gjennom hele prosjektets utvikling.

5.5 Vurdering – analyse av risiko

For hvert prosjekt som blir startet er det en viss sjanse for at prosjektet mislykkes, uansett størrelse eller erfaringsnivå hos prosjektmedlemmene. Risikofaktorer inkluderer menneskefeil og tekniske feil. I dette prosjektet vil vi lagre alt av arbeid på skyen (gjennom github/unity cloud og onedrive/dropbox), for å forminske sjansen for datatap. Et annet problem kan være om medlemmene ikke følger normene og holdningene som avtalt. Fravær er også en risiko som kan ramme prosjektet. Fravær skyldes for det meste uforutsette hendelser som er vanskelig å forebygge. Om permanent fravær skulle inntreffe en av medlemmene, vil prosjektet fortsette med en utvikler og reduserte rammer som vil bli diskutert med veileder og arbeidsgiver.

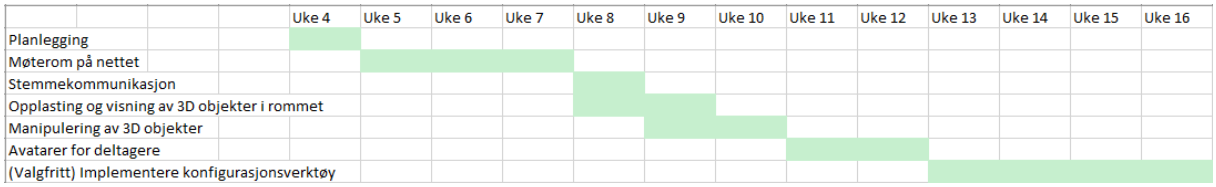
Om gruppen ikke har kunnskapen tilstrekkelig for å fullføre prosjektet, vil vi studere løsninger som igjen vil kutte ned på utviklingstid. Om gitte tidsrammer viser seg å bli for små, vil vi kontakte oppdragsgiver og veileder om en eventuell endret problemstilling og/eller kravspesifikasjon.

Vi antar at sjansen er høy for at vi klarer å realisere prosjektet innen tidsrammene.

5.6 Framdriftsplan – styring av prosjektet

5.6.1 Hovedplan

Dette er det vi har planlagt hittil. Arbeidsgiver kan selvfølgelig endre på prioriteringer og komme med andre ting han vil vi skal legge til, men dette er planen vi starter med.



Planlegging: Skriving av denne rapporten og oppsett av verktøy vi skal bruke.

Møterom på nett: Ett enkelt rom i VR som du kan koble til via TCP. Det er ikke planlagt noen grense på hvor mange som kan være i et rom, men vi antar det aldri kommer til å være flere enn 4 tilstede.

Stemmekommunikasjon: Mulighet for å snakke med hverandre. Man skal individuelt kunne endre volum på hver enkelt deltager.

Opplasting og visning av 3D objekter: Man skal kunne laste opp .obj filer til serveren og vise dem frem i møterommet.
Manipulering av 3D objekter: Funksjonalitet for å rotere, skalere og eksplodere 3D objekter i rommet.

Avatarer for deltagere: En figur som representerer en deltager slik at man kan lett se forskjell på hverandre. Man skal kunne sette denne figuren når man går inn i møterommet.
Implementere konfigurasjonsverktøy: Bedriften har ett konfigurasjonsverktøy som genererer .obj filer basert på verdier du setter inn. Hvis vi kan få dette verktøyet inn i VR rommet hadde det vært mye lettere å komme frem til det kunden vil ha.

5.6.2 Styringshjelpemidler

Vi skal bruke JIRA for alt som har med "agile" å gjøre.
Vi skal bruke Unity sitt "Collab" verktøy for versjonskontroll.

5.6.3 Utviklingshjelpemidler

Vi kommer til å bruke Unity3D til å implementere både klienten og serveren.
For programmering vil vi bruke Visual Studio 2017 og/eller Jetbrains Rider.
For rapportskriving vil vi bruke Atlassian Confluence (brukes sammen med JIRA for rapporter angående prosjektet) og Microsoft Word.

5.6.4 Intern kontroll – evaluering

Gruppen vil bruke en scrum basert arbeidsmåte, der vi på starten av hver arbeidsdag skal ha et kjapt møte for å fortelle hvordan arbeidet har gått. Etter hver sprint skal vi ha et "sprint retrospective" møte for diskuterer hva som gikk bra/dårlig den siste sprinten, og et referat til oppdragsgiver om hva som ble gjort.

Når er et mål/funksjonalitet ferdig?

- Det fungerer, og gjør hva målet spesifiserer
- Programvaren krasjer ikke
- Programvaren skal være regresjonstestet, slik at vi er sikker på at vi ikke har ødelagt tidligere fungerende kode.

5.7 Beslutninger – beslutningsprosess

Hvis det er usikkerhet kommer vi til å diskutere det mellom oss. Hvis nødvendig kontakter vi arbeidsgiver. Marcus kan mest om Unity, så han får siste ordet når det kommer til systemarkitektur.

6 DOKUMENTASJON

6.1 Rapporter og tekniske dokumenter

Vi skal rapportere til arbeidsgiver på slutten av hver sprint (annenhver uke). Vi kommer også til å skrive ned hver stor endring vi gjør slik at vi kan se tilbake på prosjektet når vi skriver selve bachelorrapporten.

7 PLANLAGTE MØTER OG RAPPORTER

7.1 Møter

7.1.1 Møter med styringsgruppen

Gruppen skal møte veileder hver andre uke (etter hver sprint) der vi diskuterer prosjektet. Dette vil foregå på torsdager. Andre møter kan arrangeres om nødvendig.

7.2 Periodiske rapporter

7.2.1 Framdriftsrapporter (inkl. milepæl)

Etter hver sprint vil vi sende et referat med interessante endringer gjort siden siste sprint til oppdragsgiver.

8 PLANLAGT AVVIKSBEHANDLING

Om prosjektet i går som planlagt, skal utvikleren med den delegerte oppgaven ta det opp med resten av gruppen. Løsninger vil bli diskutert først inniblant gruppen, deretter med veileder og så med oppdragsgiver om nødvendig. Dette avhenger på hva som er problemet som skal løses.

9 UTSTYRSBEHOV/FORUTSETNINGER FOR GJENNOMFØRING

VR briller – Oculus rift. 2 stk.