HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datavetenskap 20p
C-nivå
Vårterminen 2007

Dynamiska gränssnitt för dataspel

Anders Andersson

Dynamiska gränssnitt för dataspel

Examensrapport inlämnad av Anders Andersson till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Mikael Thieme.

2007-06-01

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Dynamiskt GUI för dataspel

Anders Andersson

Sammanfattning

Denna rapport beskriver utformningen, implementationen och utvärderingen av en arkitektur för dynamisk hantering av grafiska gränssnitt. Den dynamiska hanteringen handlar om möjligheten att ladda in och ladda ur gränssnitt som är specificerade i dokument utan att starta om spelet.

Arkitekturen realiserades genom att implementera ett system i C++. Resultatet blev ett fullt fungerande system som klarar att hantera flera gränssnittsdokument dynamiskt. Dokumenten som hanteras av systemet är skrivna i XML format som fungerar väl för beskrivning av gränssnitt.

Arbetet kommer fram till att förutom möjligheten att beskriva gränssnitt i form att kontroller och deras egenskaper behövs ett system för att hanterade händelser i gränssnittet. T.ex. när en knapp trycks så ska något hända. Därför undersöks möjligheterna för specifikation av händelsehantering i dokumentformatet. Utvecklingen av denna hantering når dock inte långt, detta arbete skrapar bara på ytan av problematiken.

Nyckelord: gränssnitt, dynamik, dokumentbaserat

Innehållsförteckning

1 Introduktion

Detta projekt undersöker en arkitektur för att skapa dokumentbaserade system för användargränssnitt, alltså ett system där gränssnitt inte är hårdkodade utan specificeras av dokument, som klarar dynamisk hantering av dokument under körtid. Med ett sådant system ska det alltså gå att ladda in gränssnitt, testa dem, ladda ur dem, modifiera dem och ladda in dem igen helt efter behag under en enda körning av applikationen.

Det traditionella sättet att utveckla gränssnitt är att gränssnittet helt enkelt programmeras precis som vilken annan del av applikationen som helst. Med tiden har behovet av allt mer avancerade gränssnitt vuxit fram och problematiken med att programmera gränssnitt växer naturligtvis också. Idag är det inte längre trivialt att skapa ett bra gränssnitt, det räcker inte längre att bara kunna programmering. Att utveckla ett bra gränssnitt kräver kunskap om människa-dator-interaktion som har blivit en hel vetenskap (Benyon, D., Turner, P. & Turner, S., 2005).

När det inte längre är programmering som är det viktigaste för att utveckla ett bra gränssnitt så kan det tänkas att gränssnitt inte längre borde skapas i programkoden. Under åren har också försök gjorts för att separera gränssnitt från programkod. Ett sätt att göra detta är att skapa ett filformat för beskrivning av gränssnitt. Programmerarens uppgift blir då att bygga ett system som kan ladda in detta filformat och skapa fungerande gränssnitt. Detta kallas för ett dokumentbaserat system. När ett dokumentbaserat system används kan en person som inte har programmeringskunskaper, men väl kunskaper om människa-dator-interaktion, skapa gränssnitt och testa dem i spelet. Dokumentbaserade system för gränssnitt har funnits en längre tid men har hittills inte blivit populärt i större utsträckning (Draheim D., Lutteroth D. & Weber G., 2006).

Att utveckla gränssnitt i dokument istället för att göra det i programkoden har flera fördelar. Framförallt gör det gör att utvecklaren av gränssnitt inte behöver en programmerares kompetens för att ändra på något. Olika gränssnitt kan skapas och testas utan att röra en enda rad programkod. Hårdkodning, med dess buggar och kompileringstider, är en stor bromskloss eftersom utveckling av bra gränssnitt är en iterativ process där utvecklaren kontinuerligt gör ändringar och testar användbarheten. Därmed finns en stor tidsvinst i att använda dokumentbaserade system.

Förutom att ett dokumentbaserat system har fördelar under utvecklingen av en applikation finns även ett intresse bland slutanvändare att förändra existerande spel. Det blir allt vanligare att slutanvändare skapar egna äventyr utan att utveckla en egen spelmotor från grunden. Med åren har spelbranschen givit allt mer stöd för dem som vill göra detta, men vad gäller förändring av gränssnittet finns mycket lite stöd om något alls.

Under senare skeden av utvecklingen av ett spel behövs tester på gränssnitt i fullt fungerande och realistiska scenarion för att få ut största möjliga kunskap om brister som behöver rättning. Dagens spel är tunga att starta, det är inte ovanligt att minuter passerar vid inladdning av en spelvärld. Om denna oproduktiva tid måste genomlidas vid varje liten ändring av gränssnittet går stora mängder arbetstid förlorad. Genom att ta bort behovet av omstart av applikationen sparas alltså tid och därmed pengar.

I denna rapport beskrivs undersökningen av ett förslag till en arkitektur som beskriver ett sätt att hantera dokumentbaserade gränssnitt. I första steget redovisas arkitekturen för att ge en översikt på förslaget. I andra steget redovisas en implementation av

arkitekturen som visar att arkitekturen är praktiskt gångbar och ger en närmare insyn i arkitekturens egenskaper. I tredje och sista steget analyseras arkitekturen för att begrunda dess styrkor, svagheter, möjligheter och begränsningar.

Målet med detta arbete är att utforska möjligheterna för en arkitektur som kan ge stora tidsvinster för utvecklare av dataspel. Tidsvinsten består främst i möjligheten att minska behovet av programmering i applikationen och därmed få bort kompileringstider och buggar från utvecklingen av gränssitt.

2 Bakgrund

Utveckling av gränssnitt går från skisser på papper till fungerande modeller där gränssnitt testas i datorprogram. På vägen från koncept till färdigt gränssnitt används många olika sorters modeller för att visualisera koncept och upptäcka brister (Benyon, Turner & Turner, 2005).

För den utveckling som sker allra närmast den riktiga applikationen behövs idag vanligtvis tillgång till programmeringskompetens medan arbetet egentligen bara borde kräva användbarhetskompetens (Benyon, Turner & Turner, 2005).

Användargränssnitt

Grafiska användargränssnitt, vanligen kallat GUI (Graphical User Interface), gör användandet av datorer mer intuitivt och lättillgängligt än rent textbaserade gränssnitt för att de kommandon som kan utföras blir visualiserade för användaren.

Ett GUI byggs upp av en mängd kontroller som fyller olika funktioner. En kontroll i ett grafiskt gränssnitt kan kallas widget och det ordet används genomgående i denna rapport. Vanliga widgettyper är knappar som fungerar på olika sätt, redigerbara textfält, fönster och mycket mer (Wikipedia, 2007).

I figur 1 visas ett exempel på ett gränssnitt med några vanliga widgets. Det finns några textobjekt som bara visar en liten text, dessa används ofta för att beskriva en del av gränssnittet så användaren vet vad alla widgets är till för. Textobjektet med texten Filename berättar vad widgeten intill är till för, nämligen att ange ett filnamn. Filnamnet anges i ett editerbart textfält där användaren kan ändra texten till rätt filnamn. Texterna Quality, Low och High beskriver vad sliderwidgeten är till för och vad som händer om den ändras. Den här slidern bestämmer vilken kvalité filen ska sparas med. Till höger syns också ett par knappar som båda har en beskrivande text på sig. Om användaren trycker på Ok kommer filen sparas, om istället Cancel trycks kommer inte filen sparas.

Figur 1 Exempel på ett gränssnitt.

2.1 Gränssnitt i dataspel

Utvecklingsmiljö

En del utvecklingsmiljöer har stöd för att bygga GUI visuellt, t.ex. Microsofts Visual Studio. Utvecklaren får dra kontroller till sin plats och sedan fylla i callbackfunktioner där koden ska manipulera applikation och widgets efter behov. Gränssnitt som utvecklas på detta sätt är hårdkodade, det går inte att ändra på något utan att bygga om applikationen och om kontroller byts ut måste delar av programmet skrivas om (Draheim D., Lutteroth C. & Weber G., 2006).

Bibliotek och motorer

Inom spelbranschen används ibland bibliotek för GUI. Ibland utvecklas egna rutiner för att få ett system som passar spelets unika behov. Men det händer också att det används ett allmänt tillgängligt bibliotek eller en spelmotor som har stöd för GUI. Anledningen till att välja bland bibliotek och spelmotorer är att spel har väldigt olika behov. Alternativen som finns tillgängliga är därför många och erbjuder stor variation i funktionalitet.

Dynamiska gränssnitt

Det finns idag applikationer som erbjuder möjlighet att modifiera gränssnittet. Detta görs vanligtvis med hjälp av ett skriptspråk vilket kan vara svårt för vanliga användare att ta till sig.

Maya(1997), ett program för skapande av 3D grafik, erbjuder ett kraftfullt skriptspråk som låter användare programmera egen funktionalitet. T.ex. kan användaren med hjälp av skriptspråket skapa en exportör för ett eget filformat och lägga till en meny för detta i Maya.

I det massiva onlinespelet World of Warcraft(2004) kan gränssnittet modifieras i mycket stor utsträckning med hjälp av det väletablerade skriptspråket Lua. Detta har blivit mycket populärt bland användarna då vissa spelare skapar något användbart och gör sina script tillgängliga för allmänheten. I figur 2 visas en kalkylator, något som inte finns i spelet från början utan är gjort av spelare som kände ett behov av denna funktionalitet.

Figur 2 Kalkylator i World of Warcraft.

Mozilla har utvecklat ett scriptspråk i XML-format, för sin webbrowser, som används för att göra gränssnitt till webbaserade tjänster som behöver komma närmare applikationer än de hypertextgränssnitt som fortfarande är standard på internet. Detta kan vara av stort intresse för utvecklare av spel som ska köras på en hemsida. Språket, som har fått namnet XUL (uttalas zool), är baserat på en mängd existerande W3C-standarder och gränssnitt som görs med XUL kan fungera både med och utan Internetanslutning (Wikipedia, 2007).

XML (Extensible Markup Language) är ett format definierat för väl strukturerad beskrivning av data. Det har blivit en populär standard för att det låter användaren bestämma hur data ska tolkas genom att lägga till regler som gäller för den egna applikationen (W3C, 2007). För ytterligare stöd till XML finns standarder för att beskriva regler som måste uppfyllas av dokument till en specifik applikation.

2.2 Relaterade arbeten

2.2.1 Growing a GUI from an XML tree (Boshart & Kosa, 2003)

Denna artikel beskriver hur XML använts som verktyg för att introducera hierarkisk programmering för datalogistudenter som ska utveckla grafiska gränssnitt. I artikeln hävdas att studenter som enbart använder en IDE har svårt att förstå hur elementen i ett GUI är relaterade till varandra.

De använde XML just för att formatet är hierarkiskt, precis som kontroller i ett GUI har hierarkiska relationer. Det utvecklades en bestämd struktur för att beskriva gränssnitt i XML samt ett program som konverterade XML till källkod i java.

Studenterna fick alltså skriva sina gränssnitt i XML och konvertera dem till källkod. Sedan måste de också programmera händelsehantering eftersom koden som genererats av konverteraren bara skapar ett grafiskt skal (Boshart & Kosa, 2003).

Detta demonstrerar att XML är ett väl fungerande format för att beskriva gränssnitt tack vare den hierarkiska strukturen. I detta projekt ska gränssnitt också specificeras i ett filformat. Men filerna ska kunna laddas in och köras direkt i en applikation, medan denna rapport bara använder XML för att generera kodskelett.

2.2.2 Graphical User Interfaces as Documents (Draheim D., Lutteroth D. & Weber G., 2006)

Denna rapport tar upp fenomenet dokumentbaserade grafiska gränssnitt, som växt fram under ganska lång tid, och beskriver dess fördelar jämfört med hårdkodade lösningar. Rapporten framhåller att det traditionella sättet att utveckla gränssnitt i programkod har stora nackdelar dels för att programmeringskunskaper krävs, dels för att både layout och funktion blir så hårt bundna och kräver mycket jobb att ändra på.

När gränssnittet skrivs i programkod blir följden i stor utsträckning att gränssnittskoden vävs in vilket gör det svårt att förändra gränssnittet. Risken för att fel uppstår är ständigt överhängande. Om dokumentbaserade gränssnitt används finns istället en klar separation och det blir lättare att göra förändringar.

Vad denna rapport beskriver är precis vad som är drivkraften bakom projektet, att kunna separera gränssnittet från programkoden. Projektet utvecklar en praktisk arkitektur för just detta och utvärderar dess möjligheter.

2.2.3 Developing Principles of GUI Programming Using Views (Bishop J. & Horspool N., 2004)

Denna rapport tar upp frågan om att utveckla principer för programmering av gränssnitt precis som det finns principer för objektorienterad programmering. Det hävdas att gränssnittskoden borde vara klart separerad från den övriga programkoden. Länkning mellan gränssnitt och övrig kod ska göras efter väl definierade principer.

Detta projekt är baserat på precis det som rapporten förespråkar. Arkitekturen som byggs upp i projektet skapar ett system för gränssnitt som står vid sidan av övrig kod. Gränssnitt måste förstås kunna kommunicera med övrig applikationskod, men det får inte bli spagettiprogrammering. För att binda det isolerade gränssnittsystemet till applikationen används ett antal väl definierade broar som byggs med säkerhet och klarhet som högsta prioritet.

3 Problem

Detta projekt fokuserar på att utforma och utvärdera ett system för dynamiska gränssnitt i dataspel.

I det här projektet beskrivs utvecklingen av en arkitektur som gör det möjligt att utveckla och köra gränssnitt utan att starta om applikationen i vilken gränssnitten ska köras. Utvecklingen av denna arkitektur används som ett verktyg för att undersöka möjligheterna till att göra utveckling av gränssitt effektivare.

Projektet är tänkt att vara till nytta för spelutvecklare genom att demonstrera hur behovet av programmeringsfärdigheter kan reduceras. Utvecklingen av gränssnitt är ett omfattande arbete, människa-dator-interaktion är ett helt eget kompetensområde som med fördel görs av personer med specialkunskaper inom området. Då gränssnitt utvecklas måste en mängd utformningar testas och detta görs på allt från pappersprototyper, ända till fullt fungerande prototyper i den färdiga applikationen (Benyon, Turner & Turner, 2005).

Det är min åsikt att de som arbetar med att utveckla gränssnitt inte borde behöva tillgång till programmeringskompetens för att göra sitt jobb. Dessutom blir det en avlastning för programmerare eftersom de slipper skriva om gränssnitt på kommando, vilket tar upp arbetstid som kan användas på annat. Projektet riktar sig även till slutanvändare som vill skapa egna gränssnitt i en existerande applikation. Med åren har spelbranschen givit allt mer stöd för dem som vill göra modifikationer på spel, men vad gäller förändring av gränssnittet finns mycket lite stöd om något alls.

Arkitekturen som utvecklas i detta projekt riktar sig till utveckling av fullt fungerande gränssnitt i en applikation. Systemet som implementeras har funktionalitet för att integreras med en applikation och kan dynamiskt ladda in och ur gränssnitt som definierats i externa filer. Utformning och implementation är gjorda för att få en praktisk inblick i vilka svårigheter och vilka möjligheter som finns för den valda arkitekturen. Resultatet av projektet är erfarenhet och insikter som förhoppningsvis kan hjälpa utvecklingen av system liknande det som utvecklats i detta projekt.

3.1 Delmål

Projektet behandlas i tre steg. Först utformas en arkitektur som löser problemet. Denna arkitektur implementeras sedan för att ge en inblick i hur väl den fungerar i praktiken. Till sist ska arkitekturen utvärderas för att belysa brister och möjligheter.

Dessa tre steg har valts för att ge läsaren möjlighet att se problemet så klart som möjligt. I utformningen får beskrivs en föreslagen lösning ur en översiktsvy. Sedan ges en närmare inblick i lokala problem tack vare implementationen. Efter att läsaren själv har fått se lösningen presenteras en utvärdering som förhoppningsvis leder till eftertanke och värdefulla insikter.

3.1.1 Delmål 1: Utformning

Det första steget mot att bygga ett fungerande system som löser problemet är att utforma en arkitektur. Arkitekturen ska specificera vilka komponenter som ska finnas i systemet och hur dessa ska kommunicera med varandra. Dessutom ska ett filformat för specificering av gränssnitt utformas.

Arkitekturen som ska lösa problemet byggs upp genom att analysera problemet med en objektorienterad metod. Att utforma en arkitektur med den objektorienterade

metoden börjar med att identifiera vilka objekt som ingår. Sedan arbetas relationer fram mellan dessa objekt. Först efter detta börjar objektens användning utformas. Genom hela utformningen måste tidigare steg utvärderas och modifieras eftersom det är mycket osannolikt att utformningen blir rätt från start.

Eftersom projektet handlar om att kunna byta ut gränssnitt under körning måste arkitekturen ha ett bra stöd för hantering av filer som specificeras gränssnitt. Det ska gå att dela upp ett gränssnitt på flera filer och användaren ska kunna ladda in och ur gränssnitt utan att starta om applikationen.

När en del av gränssnittet laddas ur gäller det att objekt som hör till den delen tas bort på ett säkert sätt. Eftersom alla delar av gränssnittet ska kunna kommunicera med varandra finns det mycket fällor vid både utformning och implementation av arkitekturen. Dessa fällor beror på hur objekt läggs till och tas bort samt hur alla delar beror på varandra. Det är stor risk att arkitekturens delar kan skapa komplexa beroenden som är svåra att lösa. I implementationen är risken för svårlösta buggar överhängande.

Kommunikationen är ett område som kräver en hel del eftertanke. Det ska gå att specificera hur widgets i ett gränssnitt ska kommunicera med varandra. Men det måste också gå att kommunicera med applikationen.

Hantering av gränssnittsfiler, widgets, händelser och meddelanden samt kommunikation mellan gränssnitt och applikation är de problem som utreds för att komma fram till en gångbar arkitektur.

3.1.2 Delmål 2: Implementation

För att få ökade insikter om arkitekturen implementeras den. Att bygga upp en implementation av arkitekturen är ett sätt att upptäcka problem som inte upptäcks i utformningen. Det är också en utmaning att få arkitekturen att fungera i praktiken. Utmaningen att få systemet att fungera ger mer värdefull kunskap och insikter i vad som är bra och dåligt i arkitekturen.

Programspråket som använts i detta projekt är C++ för att det är ett väl etablerat språk som har bra stöd för objektorienterad programmering. Hjälpbibliotek som passar projektets skala väljs med enkelhet i fokus för att projektet inte ska hindras p.g.a. långsam inlärning av funktionalitet. Genom att vara trygg i grunderna hålls lättare motivationen uppe och fokus på vad det här projektet går ut på blir tydligare. Problem som inte har med detta projekt att göra hålls till ett minimum.

3.1.3 Delmål 3: Utvärdering

Systemet utvärderas genom att analysera systemets egenskaper genom att välja ett antal kriterier. För varje kriterie diskuteras styrkor och svagheter samt förslag till förbättringar. Här samlas alla insikter om arkitekturens egenskaper i ett samlat och strukturerat format. Till skillnad från de andra två delmålen där insikterna är invävda i den utdragna skildringen av arkitekturen.

Anledningen till att detta är ett eget delmål är att under utformning och implementation fokuseras så mycket av min energi på att utveckla en produkt och lösa problemet medan projektets mål är att få ut så mycket kunskap som möjligt. I utvärderingen analyseras arbetet i syfte att gräva fram kunskap ur den hög av erfarenhet som genererats under projektets gång.

4 Metod

Guisystemet som utvecklas ska kunna ladda in dokument och bygga interface från dessa. Det ska också vara möjligt att byta ut ett dokument, d.v.s. deallokera ett interface och ladda in ett annat som uppfyller samma syfte utan att starta om applikationen.

4.1 Metod för delmål 1: Utformning

Att utforma en arkitektur för lösningen görs objektorienterat, däri finns inte mycket att välja på. Frågan är vad utformningen ska baseras på. Det som skiljer på metoderna är alltså hur de önskvärda egenskaperna för arkitekturen ska tas fram.

Metod 1: Intervjuer: En metod för att få klarhet i vilka krav som ska ställas på arkitekturen är att intervjua kunniga personer i spelbranchen om vad som är önskvärda egenskaper i en arkitektur som löser problemet.

Metod 2: Litteraturstudie: Under utformningen görs studier av relaterad litteratur som ger inspiration till arkitekturens utformning.

Val av metod: För detta projekt har metod 2 valts. Insamling av inspiration kommer på så vis från publicerade källor. Troligtvis ligger större eftertanke bakom publicerat arbete än uttalanden i intervjuer.

4.2 Metod för delmål 2: Implementation

Metod 1: Använd existerande system

En möjlighet vore att implementera dynamiska gränssnitt med hjälp av ett existerande system. Detta skulle innebära att det blir en tydlig gräns mellan implementationen för det här projektet och övriga delar i utvecklingen av ett guisystem. Det skulle också sätta begränsningar för vad som kan göras eftersom det då inte går att ändra något i det existerande systemet.

Metod 2: Bygg eget system: Istället för att bygga en arkitektur ovanpå ett existerande guisystem kan alltihop utvecklas från grunden. Det innebär en stor mängd arbete som inte har specifikt med det här problemet att göra, men det ger också mycket större frihet och kontroll eftersom det blir projektet som bestämmer hur varje detalj i arkitekturen ska fungera utan att begränsas av egenskaperna i någon annans system.

Det stora skälet emot att välja den här metoden är att det skulle gå åt mycket mer tid till att bygga grunden innan projektet kommer till den del som det egentligen handlar om. I detta fall finns dock redan en grund som byggts före projekttiden och därmed är det stora förarbetet redan avklarat.

Val av metod: Till detta projekt valdes att använda ett egenutvecklat system. Eftersom detta ger full kontroll över guisystemet finns stora möjligheter att anpassa det till detta projekt. Eftersom utvecklingen av ett guissystem påbörjats innan början av detta projekt är finns en stark motivation att bygga på detta.

Något som är gemensamt för båda metoderna är att en beskrivning av egenskaperna för den grund projektet bygger på måste tillhandahållas för att belysa varför vissa val har gjorts i implementationen. Genom att använda ett egenutvecklat system finns friheten att anpassa systemet efter projektet istället för tvärtom. Detta gör att beslut kan tas i projektets bästa intresse istället för att begränsas när grunden inte kan tillgodose projektets behov.

4.3 Metod för delmål 3: Utvärdering

Metod 1: Blackbox: En metod att utvärdera arkitekturen vore att låta programmerare och designers prova att utveckla ett program med systemet som utvecklats i implementationsfasen. Denna metod kallas blackbox för att systemet utvärderas utan att de som testar det känner till implementationen, fokus ligger på hur systemet används och arkitekturen testas från ett externt perspektiv. Sedan sammanställs deras åsikter i denna rapport.

För att göra detta behövs ett antal personer som har tid och lust att ställa upp och testa systemet. Det är inte heller något som låter sig göras snabbt och lätt eftersom systemet måste förklaras för deltagarna antingen muntligt eller genom dokumentation.

Metod 2: Whitebox: Den andra metoden är att utvärdera systemet på egen hand och frambringa största möjliga värde genom egna tankar om systemet. Denna metod kallas whitebox eftersom utvecklaren av systemet har insikt i implementationsdetaljer och kan resonera kring de val som gjorts i utvecklingen.

Denna utvärdering behöver göras på ett väl strukturerat sätt för att få tydliga och genomgående resultat. Arkitekturens delsystem ska granskas. Först anges vad som fungerar i implementationen samt vad som blivit dåligt. Sedan resoneras det om styrkor och brister i själva arkitekturen för delsystemet. Sist spekuleras om arkitekturens möjligheter och gränser.

Val av metod: För detta projekt har whitebox-metoden valts då den verkar ha störst potential att generera värdefull kunskap. Blackbox-metoden har flertalet nackdelar och risker. Den är beroende av andra människor, den kräver mycket förberedande arbete och inte minst riskerar utfallet bli ytligt och tunt.

Med whitebox-metoden finns alla resurser redan i utvecklarens huvud eftersom de byggts upp under projektet. Den kräver inga extra resurser så det går att analysera systemet utan speciell förberedelse och det finns stora möjligheter att dra fram värdefull kunskap.

5 Resultat

5.1 Delmål 1: Utformning

För att få en klar bild av vad systemet ska klara kommer här en sammanfattning av hur det ska användas.

Gränssnittsutvecklaren ska beskriva sitt gränssnitt i ett lättläst filformat som programmet sedan läser in för att bygga ett fungerande gränssnitt. Filformatet kan inte bara beskriva hur gränssnittet ska se ut utan också hur meddelanden ska hanteras.

Applikationsutvecklaren, d.v.s. den programmerare som använder systemet, ska kunna ladda in och ur gränssnitt. Det ska också gå att binda callbackfunktioner, som kan anropas av systemets meddelandehantering.

5.1.1 Filformat

Filformatet beskrivs först för att syftet med hela arkitekturen är att hantera dessa filer. För att kunna bygga ett system som hanterar filer måste först en specifikation av dessa filers struktur klargöras. Systemet använder XML filer på grund av dess struktur som fungerar väl för specificering av objekt och hierarkier.

Ett korrekt XML-dokument ska ha ett rotelement, i formatet för detta system heter roten layout. Det finns två attribut som kan sättas i elementet layout, root och mother. Attributet root anger vilken widget som ska sättas till rot när layouten blivit inladdad. Attributet mother anger en widget som är förälder till layouten. Detta innebär att en stor layout kan delas upp i flera dokument.

Widgets specificeras med widget-element. För att en widget ska kunna skapas krävs att en typ och ett namn specificeras för den. Attributet type anger vilken sorts widget som ska skapas. Typen måste naturligtvis finnas i systemet för att en widget ska kunna skapas. Attributet name anger ett namn som används för att referera till rätt widget vid meddelandehantering med mera. Namnet måste vara unikt.

Utöver de två obligatoriska attributen kan en widget ha fler attribut, exakt vilka attribut som används av en widget bestäms av typens implementation. Filformatet har dock inga regler beträffande dessa attribut.

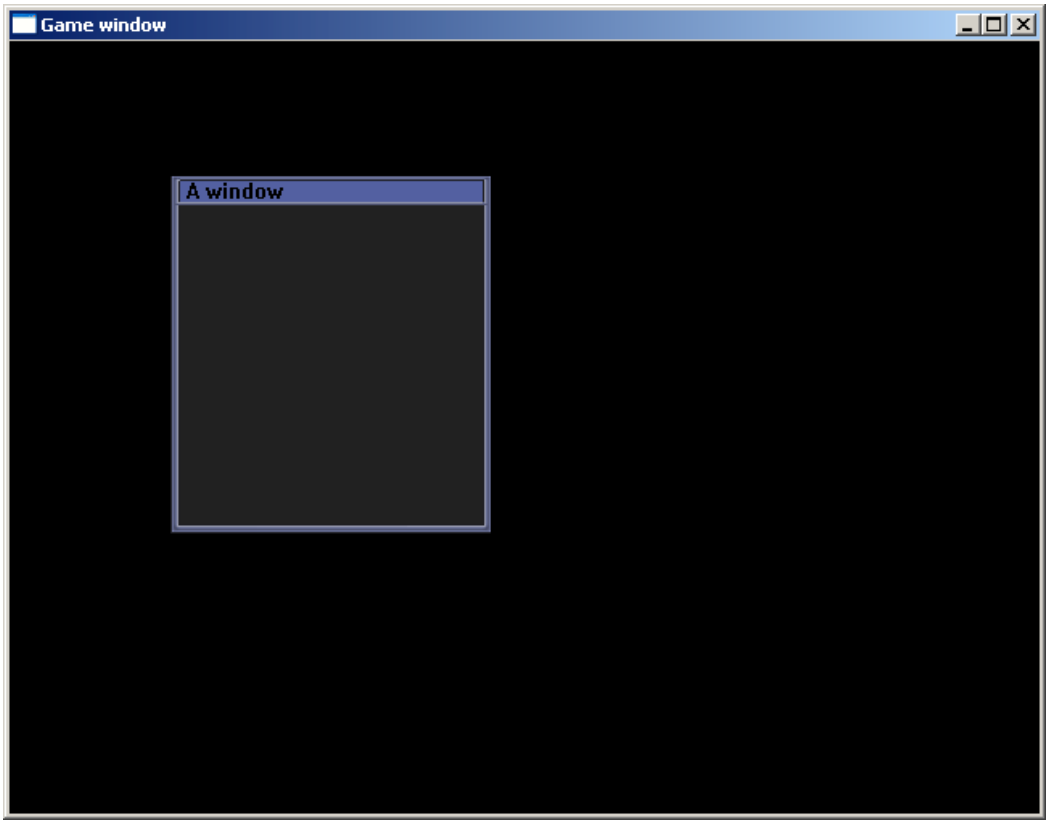
Nedan demonstreras två exempel på hur layoutdokument kan se ut. Det övre exemplet skapar vid inladdning en widget av typen Manager som sedan sätts till rot i systemet. Det nedre exemplet förutsätter att den övre layouten redan laddats in eftersom en widget med namnet Manager ska vara förälder.

```
<layout root=”Manager”>
     <widget type=”Manager” name=”Manager”/>
</layout>
```

```
<layout mother=”Manager”>
     <widget type=”Window” name=”Window” top=”100” left=”100”
width=”200” height=”200” title=”A window”/>
</layout>
```

Naturligtvis kan allt läggas i samma fil. Då läggs helt enkelt barnet under föräldern i XML-hierarkin. Resultatet av båda metoderna är ett tomt fönster vilket illustreras i Figur 3.

```
<layout root=”Manager”>
     <widget type=”Manager” name=”Manager”>
          <widget type=”Window” name=”Window” top=”100” left=”100”
width=”200” height=”200” title=”A window”/>
     </widget>
</layout>
```



Figur 3 Resultatet av en layout som specificerar ett fönster.

Filformatet kan också specificera meddelandehantering med elementen messagerelay och messageoutput. Ett messagerelay specificerar vilken händelse som ska hanteras och har två attribut, händelsens namn och namnet på källan till händelsen. Detta element kan ha en eller flera messageoutput som aktiveras när händelsen inträffar.

En messageoutput har attributen mottagare, meddelande, värdekälla och värde. Mottagaren är dit meddelandet ska skickas. Hur meddelanden hanteras varierar med mottagaren. Hur attributet värde tolkas beror på värdekällan. Om en värdekälla är angiven hämtas värdet från värdekällan, annars tolkas värdet som en konstant. Hur värdet används varierar beroende på meddelande och mottagare, vissa meddelanden använder inte värden alls.

I nedanstående exempel visas en layout med en manager och en knapp samt ett fönster som ligger utanför managern i layouten. Fönstret kommer alltså inte synas direkt efter inladdning. För att frambringa fönstret används ett messagerelay som aktiveras när knappen trycks. I messagerelay finns en messageoutput som skickar meddelandet add widget till Manager och anger värdet Window vilket är namnet på den widget som ska läggas till. I detta exempel används alltså inte värdekälla.

```
<layout root=”Manager”>
     <widget type=”Manager” name=”Manager”>
          <widget type=”Button” name=”Opener” label=”Open window”/>
     </widget>
     <widget type=”Window” name=”Window” top=”100” left=”100”
width=”200” height=”200” title=”A window”/>

     <messagerelay source=”Opener” event=”Activated”>
          <messageoutput receiver="Manager" message="add widget"
value="Window"/>
     </messagerelay>
</layout>
```

I nästa exempel används en inputbox istället för en knapp. I en inputbox kan text skrivas och den aktiveras när användaren trycker enter. I exemplets messageoutput hämtas nu värdet från inputboxen som anges av värdekällan. Attributet value anger då vilket attribut som ska hämtas av värdekällan. Om texten i inputboxen då är namnet på det fönster som ska läggas till i managern kommer fönstret att bli synligt.

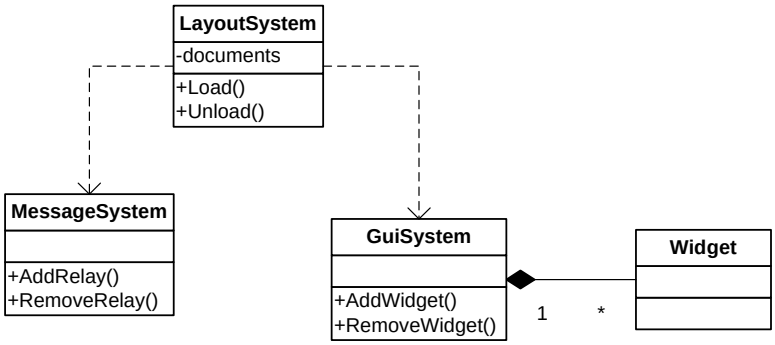
```
<layout root=”Manager”>
     <widget type=”Manager” name=”Manager”>
          <widget type=”InputBox” name=”Opener” text=”Window”/>
     </widget>
     <widget type=”Window” name=”Window” top=”100” left=”100”
width=”200” height=”200” title=”A window”/>

     <messagerelay source=”Opener” event=”Activated”>
          <messageoutput receiver="Manager" message="add widget"
valuesource="Opener" value=”text”/>
     </messagerelay>
</layout>
```

5.1.2 Komponenter i arkitekturen

När filformatet är specificerat är det dags att börja bygga en arkitektur för hanteringen av gränssnitt. För att göra det hela tydligt resoneras varje del fram logisk ordning. Tyvärr ser inte utformningen helt logisk ut i verkligheten p.g.a. att projektet i verkligheten inte följde en linjär process.

För det första, som resonerats förut, som kommer hanteringen av filer främst. Alltså är systemet som hanterar filer den första klassen, den får namnet LayoutSystem. Denna klass ska lagra xmldokument och ha funktioner för att ladda in och ur dessa.

När xmldokument laddas in och ur ska widgets och messagerelays skapas. Alltså behövs dessa två komponenter och de har var sitt system för lagring och hantering. Systemet för messagerelays heter helt naturligt MessageSystem. Dock har widgets hamnat i systemet vid namn GuiSystem. Detta är att det är en kvarleva från tiden innan det här projektet, egentligen skulle det ha gjorts ett WidgetSystem.



Figur 4 Komponenter för hantering av dokument.

Så långt kan alltså gränssnitt byggas upp från filer, likaså riva ner det. Då följer systemet som använder gränssnitten, utan vilket hela projektet vore meningslöst.

För att ett grässnitt ska vara användbart måste det få input från användaren, det ska visualiseras på skärmen och det ska kommunicera med applikationen. Dessa områden beror på vilken applikation arkitekturen ska användas i, men eftersom detta projekt handlar om en generell lösning görs bindningarna till applikationen så generella som möjligt.

Systemet ska inte lägga sig i hur input inskaffas. Beroende på vilken plattform som används ihop med systemet införskaffas input på olika sätt och applikationsutvecklaren ska kunna välja hur det ska hanteras. Systemet har istället en klass som tar emot input och det är applikationens ansvar att få tag på input och ge den till systemet efter behov. Klassen heter GuiSystem, som nu har fått två syften. Som sagt borde inte widgethantering ligga i denna klass. GuiSystem är den klass som vet vilken widget som är rot. När input skickas till GuiSystem kommer det skickas vidare till roten i gränssnittet.

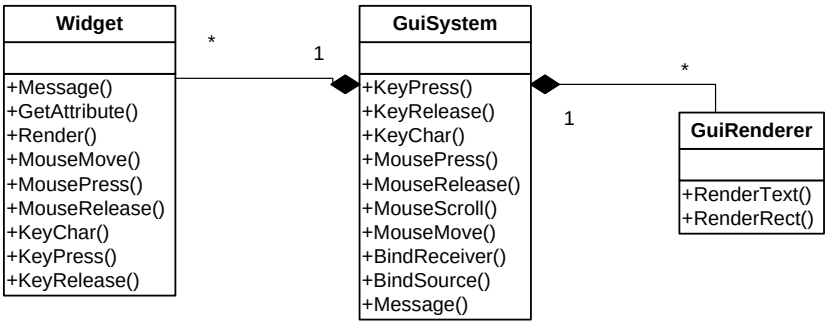
Visualisering är också mycket beroende av vilken plattform som används, mjukvarurenderat, hårdvaruaccelererat, DirectX eller OpenGL ska inte spela någon roll för systemet. Utritning kommer dock inte utifrån, som input gör, utan inifrån de widgets som ska visualiseras. För att göra visualiseringen generell behövs ett interface för de grafiska element som används av widgets. Det mest grundläggande är text som finns med i både textbaserade och grafiska gränssnitt. Sedan behövs rektanglar för att visualisera storleken på widgets och delar av widgets. I gränssnitt för befintliga applikationer syns att i stort sett allt är fyrkantigt. Widgets må ha rundade hörn visuellt men rent funktionellt är det rektangulärt, det går att klicka i det avrundade hörnet på en knapp. Det är ytterst sällsynt att widgets funktionella ytor är något annat än en rektangulära. Rent generellt behövs inte särskilt många grafiska element för att bygga upp ett gränssnitt.

Problemet här är att låta gränssnittet ritas ut inifrån systemet, men applikationen ska bestämma hur det ska gå till. Min lösning är att låta applikationen skriva en egen klass som systemet får använda. I systemet finns därför basklassen GuiRenderer som specificerar ett interface med funktioner för de grafiska element systemet använder. Applikationen skapar alltså en klass som ärver av GuiRenderer och implementerar utritning av de grafiska elementen.

Till sist kommer frågan om kommunikation mellan applikation och gränssnitt. Systemet ska täcka kommunikation i båda riktningar och vad som ska kommuniceras ska kunna specificeras både i applikationen och i xmlfilerna. Hur kommunikation

specificeras i en xml-fil utreds i 5.1.1, det som behandlas här är hur kommunikation i applikationen specificeras.

Händelser i gränssnittet ska kunna aktivera funktioner i applikationer. Gränssnittet ska också kunna hämta data från applikationen för användning i gränssnittet. Detta kan hanteras med callbacks, d.v.s. att systemet anropar applikationen när något ska göras. För att kunna använda funktioner i applikationen måste systemet först få reda på vart dessa funktioner finns. I filformatet specificeras ett mål eller källa beroende på vilken av de två nämnda funktionerna som ska användas. Alltså måste applikationen kunna registrera funktioner med de namn som de ska användas under. Den funktionaliteten har lagts i klassen GuiSystem, som ännu en gång får ett syfte till, det kan tyckas att sådan funktionalitet skulle höra till MessageSystem men som sagt har inte allt blivit perfekt. GuiSystem håller alltså reda på vilka funktioner som hör till vilket namn och kan anropa dessa vid händelser.



Figur 5 UML för input, visualisering och kommunikation

Den minsta komponenten i ett GUI är en enskild Widget. Det finns många typer av widgets men för att de ska kunna hanteras enkelt har de ett gemensamt interface. Systemet ska inte särbehandla någon typ av widget och använder därför bara pekare till basklassen Widget. Basklassen för alla widgets är ett interface med funktioner för att ta emot meddelanden från mus, tangentbord, meddelandesystemet samt eventuell fokushanterare. Den har även en renderingsfunktion för att kunna rita ut sig själv samt en klonfunktion för att widgetfabriken ska kunna skapa kopior av rätt typ.

5.2 Delmål 2: Implementation

Arkitekturen realiserades med hjälp av den spelmotor som varit under utveckling i ungefär ett halvår innan projektet. Implementationen är skriven i standard C++ med hjälp av ett par bibliotek.

För att binda funktioner för användning i meddelandesystemet användes boost-biblioteken bind och signal. Dessutom användes shared_ptr från boost för att hantera pekare på ett säkert och enkelt sätt. Anledningen till att använda boost är att dess bibliotek kraftfullare och lättare att använda än standardbibliotekens motsvarigheter.

För hanteringen av XML-dokument användes TinyXML för att det är ett enkelt lättviktsbibliotek. Sitt namn till trots har det mer än tillfredsställande funktionalitet och fyller sitt syfte för detta projekt alldeles utmärkt.

5.2.1 Widget

Varje implementation av en widgettyp är unik, vilka delar av interfacet som behöver implementeras varierar kraftigt. För att demonstrera implementation av widgets ordentligt används klassen InputBox som använder hela interfacet.

Till att börja med ska det finnas konstruktorer, destruktor mm efter behov. Det tillhör vanlig C++ kodning och dessa delar tas inte upp här. Däremot kloningsfunktion är inte lika vanlig C++ kod utan något som används av widgetfabriken i systemet. I fabriken lagras prototyper av widgets och i vissa fall kan det finnas behov av flera olika prototyper som använder samma klass. För att fabriken ska kunna skapa rätt typ av widget måste widgetklasserna ha en kloningsfunktion som returnerar en kopia av instansen.

```
Widget *InputBox::Clone()
{
     return new InputBox(*this);
}
```

Utöver de defaultvärden som satts i prototypen ska det gå att sätta attribut från layoutfilens specifikation av instansen. Därför används vid inladdning funktionen SetAttribute för att instansen ska initialiseras till det specificerade utseendet.

```
void InputBox::SetAttribute(const std::string &attribute, const
std::string &value)
{
     if(attribute=="text")
           m_text=value;
     if(attribute=="left")
           m_rect.SetX(StringToType<int>(value));
     if(attribute=="top")
           m_rect.SetY(StringToType<int>(value));
     if(attribute=="width")
           m_rect.SetW(StringToType<int>(value));
     if(attribute=="height")
           m_rect.SetH(StringToType<int>(value));
}
```

Funktionen Message tar emot meddelanden från systemet. InputBox hanterar här bara meddelandet insert som lägger in en sträng i inputboxens text. Men den skickar också vidare meddelandet till SetAttribute så att det går att sätta attribut genom meddelandesystemet.

```
void InputBox::Message
           (
                 const std::string &message,
                 const std::string &value
           )
{
     if(message=="insert")
     {
           m_text.insert(m_position, value);
           m_position+=value.length();
     }
     SetAttribute(message, value);
}
```

Funktionen GetAttribute används av meddelandesystemet för att hämta värden som ska skickas ut i meddelanden. Här finns bara inputboxens text med vilket är det mest intressanta attributet för denna klass. Naturligtvis kan det vara av intresse att hämta andra attribut men för att spara tid har detta inte implementerats.

```
std::string InputBox::GetAttribute(const std::string &attribute)
{
	if(attribute=="text")
		return m_text;
	return "";
}
```

Funktionen Render anropas när instansen ska ritas ut. Då hämtas systemets renderingsklass och ritar ut instansen med hjälp av dess funktioner. Inputbox har en ganska stor Render-funktion och has därför blivit reducerad för att inte ta så stor plats i rapporten.

```
void InputBox::Render()
{
	GuiRenderer &r=GuiSystem::GetSingleton().GetRenderer();
	r.PushClipRect(m_rect.GetX(), m_rect.GetY(), m_rect.GetW(),
m_rect.GetH());
	// Utritningskod här mycket reducerad.
	r.SetColour(std::string("text"));
	r.RenderText(m_text, m_rect.GetX()-m_scroll, m_rect.GetY());
	r.PopClipRect();
}
```

Funktionerna för mus- och tangentbordsinput används till markering och manipulering av texten. De flesta av dessa funktioner visas inte här för att de inte tillför något av intresse. Även i de funktioner som redovisas är koden kraftigt reducerad. Det som är av störst intresse i dessa funktioner är deras returvärden. Defaultvärdet är OK vilket innebär att ingen speciellt åtgärd begärs. I funktionen MousePress returneras värdet FOCUS om musklicket sker inom instansens area vilket innebär att instansen tar inputfokus. Inputfokus har betydelse för dess förälder som bestämmer hur input delas ut bland barnen.

```
Widget::Response InputBox::MousePress(int button)
{
	if(m_rect.PointIsInside(ws.GetMouseX(), ws.GetMouseY()))
	{
		m_gotFocus=true;
		return FOCUS;
	}
	return OK;
}
```

Funktionen KeyChar tar emot tecken från tangentbordet. För en inputbox innebär det i första hand att tecknet ska läggas till i texten. Funktionen returnerar värdet CONSUMED vilket innebär att meddelandet är förbrukat och inte ska ges till någon annan.

```
Widget::Response InputBox::KeyChar(unsigned int key)
{
	m_text.insert(m_position, 1, key);
	m_position++;
	return CONSUMED;
}
```

I funktionen KeyPress sker en händelse. När en händelse inträffar ska detta meddelas till systemet så att meddelandehanteringen, som specificeras i layoutfilen, kan hanteras. Widgetinstansen själv känner inte till sitt namn utan skickar sin pekare till systemet. Pekaren översätts där till instansens namn som sedan används i meddelandesystemet.

```
Widget::Response InputBox::KeyPress(int key)
{
     GuiSystem &ws=GuiSystem::GetSingleton();
     case Keyboard::RETURN:
     {
           ws.Relay("Activated", this);
           break;
     }
     return OK;
}
```

Som nämndes tidigare kan en widget ta fokus. Detta innebär att den widget som hade fokus innan måste förlora fokus. Funktionen FocusLost informerar instansen om denna händelse så att det kan hanteras korrekt.

```
Widget::Response InputBox::FocusLost()
{
     m_gotFocus=false;
     return OK;
}
```

Det kan också hända att en widget blir tilldelad fokus. Till detta används funktionen GotFocus som helt enkelt informerar instansen om att den fått fokus.

```
void InputBox::GotFocus()
{
     m_gotFocus=true;
}
```

Till sist finns också funktionen Move som används för att kunna flytta på en widget. Denna funktion finns för att det ska gå att ha t.ex. fönster. När fönstret flyttar behöver det dra med sig sina barn vilket det åstadkommer med hjälp av denna funktion.

```
void InputBox::Move(int dx, int dy)
{
     m_rect.SetPosition(m_rect.GetX()+dx, m_rect.GetY()+dy);
}
```

5.2.2 WidgetFactory

För att kunna skapa widgets av rätt typ från namnet på typen så finns i systemet en widgetfabrik. När systemet initialiseras registreras de widgettyper som ska kunna skapas som prototyper i fabriken. Vid inladdning av layouts skapas kopior av dessa prototyper. Vilka sorters widgets som kan skapas beror alltså på vilka typer som registrerats i fabriken.

Eftersom widgetfabriken är platsen där widgets skapas är det här ansvaret börjar för att pekarna till alla widgets hanteras säkert i systemet. Därför används boost::shared_ptr som håller reda på när det är säkert att ta bort instansen. Enligt dokumentationen för shared_ptr är det bra att alltid använda shared_ptr överallt där new används. Dock används inte detta konsistent igenom hela systemet eftersom shared_ptr kom in sent i projektet. De används i det här projektet bara för att lösa ett specifikt problem och det fanns inte tid till att konvertera hela systemet från att använda vanliga pekare till att använda shared_ptr. Just vad problemet bestod i finns närmare beskrivet under 5.2.4 som handlar om hierarkier i gränssnittet.

I de kodexempel där shared_ptr används skrivs typen som typens namn plus ändelsen Ptr eftersom en typedef används av bekvämlighetsskäl.

```
typedef boost::shared_ptr<Widget> WidgetPtr;
```

Att lägga till en prototyp i fabriken är ganska trivialt. En instans av en specifik widgetimplementation skapas och lagras i en shared_ptr. Sedan kan de attribut sättas som ska ha defaultvärden, det är frivilligt och i de flesta fall görs det inte. Till sist läggs prototypen till i fabriken med funktionen AddPrototype och då specificeras typnamnet. Det kan finnas intresse av att skapa flera prototyper av samma widgetimplementation. Som exempel kan en implementation av en knapp användas både som checkbox och radioknapp genom att olika defaultvärden sätts.

```
WidgetPtr button=WidgetPtr(new Button);
button->SetAttribute("normaltexture", "button");
button->SetAttribute("pressedtexture", "buttonpressed");
WidgetFactory::GetSingleton().AddPrototype("Button", button);
```

Kärnfunktionen för en fabrik är naturligtvis funktionen som returnerar användbara widgets. Funktionen GetPrototype tar ett namn som argument och returnerar en widget av den typ som namnet anger. Om en prototyp med givet namn inte existerar kastas ett fel som får hanteras av den som anropade funktionen. Vanligtvis ska det vara en LayoutHandler som skapar widgets och där finns korrekt hantering av felet.

```
WidgetPtr WidgetFactory::GetPrototype(const std::string &name)
{
     if(m_prototypes.find(name)==m_prototypes.end())
     {
          std::string msg=std::string("Widget type does not
     exist:")+name;
           throw exception(msg.c_str());
     }
     WidgetPtr widget(m_prototypes[name]->Clone());
     return widget;
}
```

5.2.3 GuiSystem

I utformningen konstateras att GuiSystemet har fått flera syften och att dess funktionalitet borde delas upp mellan flera klasser för att göra utformningen tydligare. Detta beror på att mycket av implementationen gjordes innan projekttiden och då fanns ingen formell utformningsplan som skulle följas. Under projekttiden har GuiSystem gått igenom stora förändringar och brutits isär ganska mycket. De delar som finns kvar i GuiSystem ligger bara kvar för att det inte kändes tillräckligt nödvändigt att bryta ut dem och tiden för projektet led mot sitt slut.

5.2.3.1 Toppkontroll

I GuiSystem hanteras den högsta nivån av kontroll av gränssnittet. Detta är egentligen den enda av de tre delarna som borde finnas här. Samtidigt är det den enklaste delen även om antalet funktioner är mycket större än de andra två tillsammans.

Denna del tar emot all input från mus och tangentbord för att skicka till rotwidgeten. Den har också en funktion som säger till rotwidgeten att rita ut sig. All övrig kontroll av gränssnittet lämnas till hierarkiska system inom gränssnittet.

Eftersom alla dessa funktioner är så enkla ges här ett litet exempel som får stå för allting. Roten till gränssnittet lagras i GuiSystem som en weak_ptr, vilket innebär att innan roten kan användas måste den låsas genom att skapa en shared_ptr. Vidare måste det göras kontroll som säkerställer att instansen fortfarande existerar. Sedan är det bara att anropa funktioner precis som på en vanlig pekare. Funktioner som tar emot input returnerar true ifall input har använts, annars returneras false.

```
bool GuiSystem::KeyPress(unsigned int key)
{
     WidgetPtr root=m_root.lock();
     if(root!=NULL)
           return root->KeyPress(key)==Widget::CONSUMED;
     return false;
}
```

5.2.3.2 Widgethantering

Till att börja med sköter GuiSystem lagringen av widgets. Det behövs inte så mycket funktionalitet i detta, det går att lägga till och ta bort widgets samt hämta en widget. En widget lagras i en map som mappar widgetnamn mot widgetinstanser. Instanserna lagras med hjälp av boosts shared_ptr, anledningen till detta förklaras i 5.2.4 som behandlar hierarkier i gränssnitt.

```
typedef std::map<std::string, WidgetPtr> WidgetMap;
WidgetMap m_widgets;
```

När en widget läggs till i GuiSystem med funktionen AddWidget måste dess namn vara unikt. Om det inte är unikt kastas ett fel.

```
void GuiSystem::AddWidget(const std::string &name, WidgetPtr widget)
{
     if(m_widgets.find(name)!=m_widgets.end())
     {
           std::string msg=std::string("Widgetname
      collision:")+name;
           throw std::exception(msg.c_str());
     }
     m_widgets[name]=widget;
}
```

Funktionen GetWidget returnerar en WidgetPtr som är en shared_ptr till instansen. Om ingen widget finns under det givna namnet returneras en tom WidgetPtr, en shared_ptr utan instans lagrar helt enkelt NULL.

```
WidgetPtr GuiSystem::GetWidget(const std::string &name)
{
     if(m_widgets.find(name)!=m_widgets.end())
           return m_widgets[name];
     WidgetPtr w;
     return w;
}
```

Funktionen RemoveWidget gör naturligtvis bara en sak, den tar bort instansen som är associerad till det givna namnet. Detta gör att de weak_ptr’s som lagras på andra ställen blir tömda genom att deras pekare sätts till NULL, och därmed går det att se om en instans blivit borttagen.

```
void GuiSystem::RemoveWidget(const std::string &name)
{
     m_widgets.erase(name);
}
```

5.2.3.3 Meddelandehantering

GuiSystem hanterar också meddelanden till applikationen och till widgets. Klassen har två mappningar för funktioner som binder systemet till applikationen, en för att hämta data från applikationen och en för att skicka data till applikationen.

Funktioner som kan bindas till systemet måste se ut som specificeras nedan. En funktion som tar emot meddelanden tar två strängar, en för meddelande och en för data. En funktion som är datakälla tar en sträng som anger vilket värde som ska returneras och returnerar en sträng som innehåller värdet. Bindningarna görs med hjälp av boosts function.

```
typedef boost::function<void (const std::string &message, const
std::string &value)> AppReceiver;
typedef boost::function<std::string (const std::string &value)>
AppSource;

typedef std::map<std::string, AppReceiver> AppReceiverMap;
AppReceiverMap m_applicationReceivers;
typedef std::map<std::string, AppSource> AppSourceMap;
AppSourceMap m_applicationSources;

void GuiSystem::BindReceiver
(
     const std::string &name,
     AppReceiver binding
)
{
if(m_applicationReceivers.find(name) !=
m_applicationReceivers.end())
     {
          return;
     }
     m_applicationReceivers[name]=binding;
}

void GuiSystem::BindSource
(
     const std::string &name,
     AppSource binding
)
{
if(m_applicationSources.find(name)!=m_applicationSources.end())
     {
          return;
     }
     m_applicationSources[name]=binding;
}
```

Kärnan i denna hantering är funktionen message som tar parametrar precis som de specificerats i layoutdokumentet och bygger ett meddelande. Anledning till att den här funktionen finns i GuiSystem är att den behöver slumpmässig tillgång till alla widgets och de lagras ju här. Anledningen till att bindningarna till applikationen ligger i denna klass beror i sin tur på att funktionen Message ligger här. Det finns dock ingen stark anledning att ha dessa funktioner här, de har bara inte flyttats till MessageSystem än.

```
void GuiSystem::Message(const std::string &receiver,
const std::string &message,
const std::string &valueSource,
const std::string &value)
{
     std::string sendValue;
     if(valueSource!="")
     {
           WidgetPtr source=GetWidget(valueSource);
           if(source!=NULL)
           {
                 sendValue=source->GetAttribute(value);
           }

           GuiSystem::AppSource appSource =
                 m_applicationSources[valueSource];
           if(appSource!=NULL)
           {
                 sendValue=appSource(value);
           }
     }
}
```

5.2.4 Hierarkier

Vid specifikationen av filformatet framkommer det att widgets kan arrangeras i hierarkier. Vad detta innebär i implementationen är att vissa widgettyper kan innehålla barnwidgets. Hantering av widgets är ett mycket komplext område som skulle kunna utgöra ett antal exjobb helt på egen hand. Därför förklaras här bara en del av hur de hanterats i detta projekt med avseende på det faktum att widgets läggs till och tas bort på grund av dynamiken i arkitekturen. Övrig implementation är inte särskilt relevant för denna rapport.

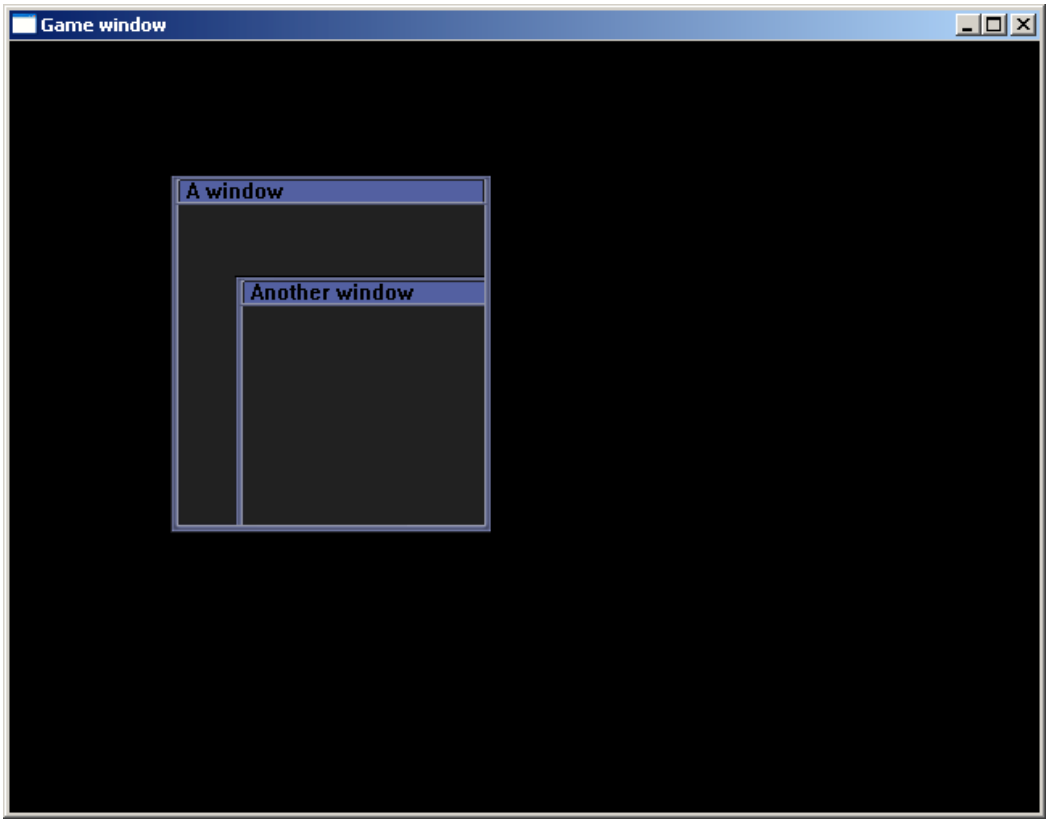
I följande exempel för filformatet syns att roten är en widget av typen Manager. Denna widget är en typ som är gjord för att hantera en mängd barn och kontrollera vilken som ska ha input och i vilken ordning alla widgets ska ritas ut. Manager själv har ingen grafisk representation, den arbetar i bakgrunden för att tillhandahålla viktig funktionalitet.

```
<layout root=”Manager”>
     et type=”Manager” name=”Manager”>
           <widget type=”Window” name=”Window” top=”100” left=”100”
width=”200” height=”200” title=”A window”>
                 <widget type=”Window” name=”Window” top=”100”
left=”50” width=”100” height=”200” title=”Another window”/>
           </widget>
     </widget>
</layout>
```

I exemplet finns även widgets av typen Window. Den här typen ärver av Manager men den har en form och ritas ut som ett fönster liknande de som finns i ett modernt operativsystem. Window tillför också förflyttning, när ett fönster dras så flyttar det sig självt och sina barn. I exemplet finns ett fönster inuti ett annat fönster för att demonstrera att man kan lägga vilka widgets som helst inuti en widget som kan hantera barnwidgets. Resultatet visas i Figur 6.

Manager och Window är de två widgets som implementerats i det här systemet för att hantera hierarkier. Det finns fler sätt att hantera grupper av widgets men med dessa två täcks den grundläggande problematiken i ett hierarkiskt system.

I implementationen av manager går det att lägga till barn genom att skicka barnets namn med meddelandet add widget. Det går även att ta bort ett barn med meddelandet remove widget ihop med barnets namn. Det här sättet att hantera barn används för att det ska gå att göra genom meddelandesystemet. Därmed går det att kontrollera hur barn läggs till och tas bort i systemets filformat.



Figur 6 Resultatet av en hierarki av fönster.

I implementationen av Manager lagras dock inte namn på barnen. Om bara namn lagrades skulle den behöva hämta pekare till barnens instanser från GuiSystem varje gång de ska användas. Därför är det bra att lagra pekare, dels för att få enklare kod, dels för att göra koden mer effektiv. Pekare hämtas alltså bara när ett barn ska läggas till eller tas bort.

Att lagra pekare medför dock ett problem. När en widget tas bort från GuiSystem så förstörs instansen. Men i Managern ligger pekaren kvar och det finns en risk att Managern försöker använda en instans som inte längre finns. För att lösa detta introducerades boosts shared_ptr i systemet. Genom att lagra pekare till instanser i shared_ptr garanteras att instansen inte förstörs så länge någon shared_ptr för pekaren finns kvar.

Att bara använda shared_ptr löser dock inte hela problemet. Om en shared_ptr lagras i Manager så kommer ju inte instansen förstöras när den tas bort från GuiSystem, vilket ska göras. Lösningen finns i boosts weak_ptr, en weak_ptr är som en slav åt shared_ptr. När pekaren lagrats i en shared_ptr kan den sedan även lagras i en weak_ptr. Skillnaden mellan de två är att när alla shared_ptr försvinner så förstörs instansen, det spelar ingen roll om det finns weak_ptr till pekaren. De weak_ptr's som

finns kvar kommer dock automatiskt ändra sin pekare till NULL så att det går att få reda på att instansen inte längre finns.

5.2.5 MessageSystem

Meddelandesystemet är byggt på grunden att vid en given händelse skall en viss output genereras. En händelse har två nyckelattribut, namnet på en källa och namnet på en händelse som inträffade i denna källa.

För att lätt kunna hitta vilken output som skall genereras används en dubbel map just för de två nyckelattributen eftersom båda attributen krävs för att bilda en nyckel. Den output som skall genereras vid en händelse lagras i en lista eftersom systemet bara har stöd för ovillkorlig sekventiell output.

```
typedef std::list<MessageOutput> Output;
typedef std::map<std::string, Output> Eventmap;
typedef std::map<std::string,  Eventmap> Relaymap;
Relaymap m_relays;
```

Med funktionen AddOutput läggs output för en händelse. Det enda som krävs för att output ska läggas till är att källa och händelse har angetts. Det finns inget som hindrar att det finns identiska output eftersom detta kan tänkas vara önskvärt. När output ska tas bort anropas RemoveOutput som har exakt samma parametrar som AddOutput.

```
void MessageSystem::AddOutput(
     const std::string &eventSource,
     const std::string &eventname,
     const std::string &receiver,
     const std::string &message,
     const std::string &valueSource,
     const std::string &value )
{
     if(eventSource!="" && eventName!="")
     {
           Output &output=m_relays[eventSource][eventName];
           MessageOutput o(receiver, message, valueSource, value);
           output.push_back(o);
     }
}
```

När en händelse inträffar kommer systemet att anropa funktionen Handle i MessageSystem som slår upp listan med output och skickar meddelanden. Funktionen Dispatch som används på klassen Output anropar funktionen Message som finns i GuiSystem.

```
void MessageSystem::Handle(
     const std::string &widgetname,
     const std::string &event)
{
     for(Output::iterator i=m_relays[widgetname][event].begin();
     i!=m_relays[widgetname][event].end(); ++i)
           (*i).Dispatch();
}
```

5.2.6 LayoutSystem

Hanteringen av layoutfiler sköts med hjälp TinyXML som är ett minimalt bibliotek för XML. Med TinyXML går det lätt att läsa in XML-dokument och tolka dess data efter behov. Det går även att ändra i dokument samt skapa nya dokument.

När en layout ska laddas in till eller ut från systemet görs detta genom klassen LayoutSystem som håller reda på vilka filer som är inladdade. Vid inladdning skapas en ny instans av klassen LayoutHandler som har hand om hanteringen av det specifika XML-dokumentet. Vid urladdning tas instansen bort.

LayoutHandler använder sig av basklassen TiXmlVisitor, som finns i TinyXML, för att gå igenom XML-filens element i djupet först ordning. När en klass ärver TiXmlVisitor kan den implementera funktioner för varje sorts nod i ett XML-dokument, i det här systemet används funktionerna som anropas då visitor-klassen besöker elementnoder. Eftersom olika saker ska göras beroende på om ett dokument laddas in eller ur finns flera klasser som ärver av TiXmlVisitor.

Vid inladdning körs först en preload då går dokumentet gås igenom för att hitta allvarliga fel som t.ex. eventuella kollisioner med tidigare inladdade widgets. De fel som hittas skrivs till en loggfil så att användaren kan kolla upp varför en layout inte fungerar. Om dokumentet inte innehåller några fel som hindrar säker inladdning så körs själva inladdningen.

Under inladdningen skapas widgets och meddelanderelän. Här kan också fel upptäckas men då handlar det om fel som systemet tillåter. Dessa fel skrivs naturligtvis också till loggfilen. Namnkollisioner mellan widgets i samma layout är förvisso inte tillåtet, men det hindrar inte inladdningen.

Anledningen till att inte tillåta namnkollision mellan olika layoutfiler är problem vid urladdning. När ett dokument laddas ur ska det inte kunna orsaka att en widget som tillhör en annan layout försvinner från systemet. Detta är förvisso en smaksak och inte helt nödvändig för att systemet ska fungera.

Vid urladdning av en layout körs förstås en visitor som gör igenom dokumentet och tar bort alla widgets och meddelanderelän som är specificerade däri.

5.2.7 Exempelprogram

För att visa hur implementationen används följer här kod från ett exempelprogram som laddar in och byter ut gränssnitt dynamiskt. Först måste systemet initialiseras. Detta är den krångligaste biten, men i applikationskoden anropas bara funktionen Initialize. Denna funktion implementeras specifikt för de grafik- och inputsystem som används. Detta är dock inte relevant för denna rapport och detaljer om hur systemet initialiseras utelämnas därför här. Det är applikationskoden som är viktig, portabilitet lämnas åt andra projekt att utreda.

```
gui::Initialize();
```

När systemet initialiserats går det att ladda in dokument som innehåller specifikationer för gränssnitt genom layoutsystemet.

```
gui::LayoutSystem::GetSingleton().Load("Data/guilayout/root.xml");
```

I gränssnittet finns meddelanderelän som använder en virtuell widget vid namn ”App”. Denna widget finns inte med i gränssnittet utan ska hänvisa till funktioner i applikationen. I applikationen finns därför en klass som har funktioner som ska användas till detta. Funktionerna binds enligt följande kod så de blir registrerade under namnet ”App” i GuiSystem.

```
gui::GuiSystem &gs=gui::GuiSystem::GetSingleton();
gs.BindReceiver(
           "App",
           boost::bind(&Application::Message, this, _1, _2)
     );
gs.BindSource(
           "App",
           boost::bind(&Application::Source, this, _1)
     );
```

Funktionen Message är bunden som mottagare. Den här funktionen kan ta emot vilka meddelanden som helst, men i det här fallet används bara meddelandet switch child. När det meddelandet kommer byts en layout ut i gränssnittet.

```
void Application::Message(
           const std::string &message,
           const std::string &value)
{
     if(message=="switch child")
     {
      rf::gui::LayoutSystem::GetSingleton().Unload(m_childName);
      m_childName=value;
      rf::gui::LayoutSystem::GetSingleton().Load(value);
     }
}
```

Funktionen Source är bunden som källa och returnerar värden som berättar något för gränssnittet. I det här fallet returneras bara filnamnet på nuvarande layout.

```
std::string Application::Source(const std::string &value)
{
     if(value=="child")
           return m_childName;
}
```

Varje grafisk frame ska gränssnittet ritas ut, annars kommer det inte synas. Allt som behöver göras är ett anrop till Render i GuiSystem.

```
gui::GuiSystem::GetSingleton().Render();
```

Till sist ska systemet stängas ner innan applikationen avslutas. Funktionen Shutdown ser till att allt minne som används i systemet frigörs korrekt.

```
gui::Shutdown();
```

Figur 7 och Figur 8 visar två olika gränssnitt laddade I exempelprogrammet. Båda gränssnitten laddas efter varandra och fyller samma funktion. Med programmet kan man testa oändligt antal prototyper av sitt gränssnitt utan att starta om applikationen.

Figur 7 Ett gränssnitt laddat i exempelprogrammet.

Figur 8 Ett annat gränssnitt som ersätter det förra.

5.3 Delmål 3: Utvärdering

Först anges vad som fungerar i implementationen samt vad som blivit dåligt. Sedan resoneras det om styrkor och brister i själva arkitekturen för delsystemet. Sist spekuleras om arkitekturens möjligheter och gränser.

5.3.1 Widgethantering

Widgethanteringen i implementationen fungerar genom att widgets skapas och lagras i en widgethanterare där den mappas till det namn som är angett i gränssnittsdokumentet. Den skapade widgeten finns sedan tillgänglig för användning i hela systemet och hanteringen av den är säker tack vare användningen av boosts shared_ptr. Funktionaliteten är fullt acceptabel i nuvarande skick. Den största bristen i widgethanteringen är att den klass som lagrar widgets inte är dedikerad till enbart detta. Men eftersom det är en ren implementationsdetalj har det ingen större vikt i den stora bilden.

Så som arkitekturen ser ut nu verkar den här delen vara så gott som komplett. Möjligheterna för widgettyper i princip oändliga, det går att skapa egna widgettyper och registrera dessa i widgetfabriken utan problem. En begränsning för widgets är dock att meddelanden bara använder strängar. Det är något som borde undersökas för att upptäcka ett bättre sätt att hantera olika sorters data, detta gäller framförallt komplexa datastrukturer.

5.3.2 Meddelandehantering

Systemet klarar att förmedla meddelanden bland widgets inom en layout, widgets i flera layouts samt använda applikationskod som både källa och mål. Det går att binda funktioner i applikationen till gränssnittet för användning som callbacks och det fungerar smidigt och enkelt. Den meddelandehantering som specificeras i gränssnittet skapas vid inladdning och tas bort vid urladdning utan att fel uppstår. Det kan tänkas att det finns brister i säkerheten som inte kan demonstreras utan grundlig testning och analys som inte hunnit med i detta projekt. Men dessa säkerhetshål kan troligtvis lösas genom lite extra arbete i implementationen.

Arkitekturen beskriver ett system som är lätt att arbeta med. Dokumentformatet är lättarbetat och interfacet för att binda funktioner i applikationen är tydligt. Systemet är ändå mycket begränsat. Det finns mycket som skulle kunna göras för att göra det mer användbart. Meddelandehanteringen är den del av systemet som är i störst behov av vidare utveckling. De två exemplen som följer är problem som inte hann lösas under detta projekt.

En begränsning är att det inte finns stöd för villkor för output. När en händelse inträffar kommer alltid all output för händelsen att skickas vilket inte alltid är önskvärt. Som exempel kanske det finns behov av att skicka ett värde från en inputbox till en annan widget, men värdet ska bara skickas om det är ett numeriskt värde.

En annan begränsning är att det bara går att skicka en textsträng som värde. För att få större spelrum vore det önskvärt att kunna förmedla mer avancerade meddelanden. En fillista är ett exempel på en widget som har behov av flervärdesmeddelanden. Den kan ha flera kolumner för att visa t.ex. filnamn och filstorlek. När en fil ska läggas till i listan behövs kanske flera värden i meddelandet.

5.3.3 Layouthantering

Implementationen av layouthanteringen fungerar bra. Systemet klarar att ladda in och ur en layout samt ersätta en layout med en annan som har överlappande widgetnamn utan problem. Det är dock sannolikt att inte alla möjliga fel hanteras i det nuvarande systemet. Vid komplexa scenarion kan det troligtvis uppstå fel som inte förutsetts, tyvärr har ingen fördjupad testningsfas hunnits med.

Hanteringen av layoutfiler är mycket enkel att använda, men systemet kan lätt drabbas av konflikter när flera layoutfiler hanteras. Den här delen av systemet lägger till och tar bort widgets och meddelanderelän vilket kan leda till stora problem om flera layouts innehåller överlappande delar. Genom att söka igenom varje layout innan inladdning för att upptäcka konflikter undviks de vanligaste felen som skulle uppstå om layoutfiler laddades in helt utan kontroll.

En svaghet i att använda ett dokumentbaserat system är att det tar mycket längre tid att ladda gränssnitt från filer än att använda hårdkodade gränssnitt. Filinläsning i sig är alltid tidskrävande och eftersom dessa gränssnitt även måste kontrolleras innan det går att skapa gränssnittet tar det ännu längre tid. Det är förstås möjligt att göra detta snabbare om förinläsningen bara behövs under utvecklingen och optimerar filformat och inläsningsprocess när spelet släpps. En annan svaghet är att i nuvarande arkitektur måste gränssnitt skrivas direkt i dokument. I avancerade utvecklingsmiljöer finns editorer där det går att dra omkring kontroller och ändra deras egenskaper visuellt. Det är högst troligt att arkitekturen som beskrivs i denna rapport kan expanderas och vidareutvecklas för att göra detta möjligt på ett bra sätt.

6 Slutsats

6.1 Diskussion

I arbetet har en stabil och kraftfull arkitektur utvecklas. Men framförallt är den ganska enkel att arbeta med. Arkitekturen har utformats så att varje del blir så smal som möjligt. Arkitekturen beskriver ett dokumentbaserat system, det är grunden till hela projektet. Det går att beskriva sina gränssnitt i ett format som är lätt att förstå istället för att använda ett programmeringsspråk vilket gör att systemet blir lätt att ta till sig. Dokumentformatet använder XML som är ett kraftfullt och enkelt format. Det är enkelt att arbeta med både när arkitekturen realiseras i ett riktigt system och för den som skapar gränssnitt.

Arkitekturen har behövt sträcka sig längre än att bara specificera vad som ska finnas i gränssnittet och hur det ska se ut. För att kunna göra ett bra dynamiskt system kan kommunikation också beskrivas i dokumenten. Möjligheten att göra detta leder till att arkitekturen har stora möjligheter. Dock har detta bara nått en grundläggande nivå. I framtida arbete behöver meddelandesystemet utvecklas, som det ser ut nu finns inte all funktionalitet som behövs för gränssnitt som behöver avancerad kontroll av kommunikation.

Implementationen genomsyras av principen att tydligt separera gränssnittssystemet från övrig applikationskod. I implementationen har systemet isolerats så mycket som möjligt. Genom att göra ett tydligt och minimalt interface skapas ett system som är enkelt för applikationsprogrammeraren att använda. Det har visat sig att arkitekturen varit ganska lätt att implementera. Inga allvarliga hinder har stått i vägen för realiseringen.

6.2 Framtida arbete

Om det funnits någon vecka över på projekttiden skulle systemet för kommunikation ha utvecklats vidare. Speciellt möjligheten att sätta villkor för output så det går att styra kommunikationen bättre. Med en större tidsram skulle det vara en bra idé att göra en djupare undersökning av hur data kan skickas på bättre sätt. Dels genom ett format som är mer optimerat än strängar, dels på ett sätt som är mer flexibelt.

Vad gäller inladdning kan det vara intressant att undersöka hur inläsningsprocessen kan optimeras. Dagens spel tar redan ganska lång tid att ladda så för vissa spel kan det vara av stort intresse att spara in så mycket tid som möjligt för att minska frustrationen hos spelaren. En optimering som vore relativt snabb att göra vore att ändra systemet så att säkerhetskontroller bara används under utvecklingen av spelet. Med en större tidsram kan det vara en idé att utveckla ett binärt filformat som går snabbare att läsa in. Dessa filer kan skapas genom ett program som konverterar från XML-formatet alternativt utvecklar en möjlighet att spara binära dokument från en editor.

Om det fanns riktigt gott om tid och resurser borde det byggas en grafisk gränssnittseditor för arkitekturen. Det skulle göra mycket för arkitekturens användbarhet. Att sitta med textdokument och ändra på positionsvärden numeriskt är inte det mest effektiva sättet att arbeta med en layout. Möjligheten att arbeta visuellt och använda en mus för att flytta och ändra på kontroller är av stort värde för användbarheten av en dokumentbaserad arkitektur.

Referenser

Benyon, D., Turner, P. & Turner, S. (2005) *Designing interactive Systems.* Pearson Education Limited

Bishop J. & Horspool N. (2004) *Developing Principles of GUI Programming Using Views* ACM Press

Boshart M. A. & Kosa M. J. (2003) *Growing a GUI from an XML tree* ACM Press

Draheim D., Lutteroth C. & Weber G. (2006) *Graphical user interfaces as documents.* ACM Press

Maya (2004) [Datorprogram] Alias Systems Corp

W3C (2004) *Extensible Markup Language (XML)* (W3C) Tillgängligt på internet: http://www.w3.org/xml/ [Hämtad 07.02.28]

Wikipedia (2004) *Widget (computing)* Tillgängligt på internet: http://en.wikipedia.org/wiki/Widget_(computing) [Hämtad 07.02.28]

World of Warcraft (2004) [Datorprogram] Blizzard Entertainment