Erfarenhetsrapport

Det här dokumentet innehåller grupp c4's erfarenheter från projektet Ray Fighters i kursen TDDI02. Erfarenheterna är indelade efter fyra olika faser som uppstod under projekttiden. Dessa är krav, design, implementation och resultat. Totalt innehåller dokumentet åtta erfarenheter, fyra gemensamma och fyra personliga erfarenheter.

Krav

Viktigt att täcka alla designval till kraven

I början av projektet när vi definierade kraven i kravspecifikationen tyckte vi det var svårt att definiera alla krav, främst på grund utav att vi inte hade en helt klar design. Nu när vi implementerar uppstår många frågetecken angående spelets mekanik och inser nu att det skulle vara smidigt med krav till ALLT.

Design

Att planera ett projekt ordentligt ifrån grunden har varigt en positiv erfarenhet. Den noggranna och väl genomförda designplanen har gjort projektet mycket lättare att genomföra än tidigare projekt där fokus mer har legat på att bara utföra en uppgift utan en längre planeringsfas. Detta har lett till att det har blivit uppenbart att ett väldesignat program går att implementera på kortare tid även om det är ett mer avancerat system.

Optimera mängden klasser

Att optimera mängden klasser är en svår vägning eftersom man vill eftersträva en tydlig struktur och att varje klass ska ha en specifik uppgift och inte vara en onödig klass som inte är essentiell för projektet.
I vårt förra projekt hade vi onödiga klasser och det tog vi lärdom av och bestämde oss för att i detta projekt endast ha klasser som vi tyckte har en specifik uppgift. Detta gav konsekvensen att vi i detta projekt fick för få klasser och blev då tvungna lägga till ett par klasser som saknades i ULM-diagrammet, detta var i vår erfarenhet en mycket lättare process än att ta bort överflödiga.

Implementation

Statiska klassmedlemmar I C++

I vårat förra projekt använde vi oss av en klass med en statisk map i för att kunna återanvända en textur flera gånger och alla filer i dokumentet hade då tillgång till denna klass. Till detta projekt tänkte vi förbättra denna idé och göra att endast de klasser som behöver använda texturer får tillgång till textur-library-klassen.
För att lösa detta gjorde vi textur-library-klassens medlemmar protected och static så att man endast fick tillgång till dem om klassen man är i ärvde ifrån textur-library-klassen. Eftersom att textur-library-klassen var såpass liten så gjorde vi endast en .h till den. Då uppstod ett problem då vi i samma fil deklarerade den statiska filens existens vilket gjorde att vi deklarerade den varje gång en fil inkluderade h-filen. Vi upptäckte senare att man är tvungen att då ha en cc-fil till en klass som behöver deklarera i det statiska fältet i c++.

Mer kommunikation och hänvisningar till designdokument /Tobias

När designfasen var avklarad och gruppen påbörjade implementationsfasen, arbetade alla individuellt på olika klasser i projektet med få möten eller läsande av designspecifikation. Detta ledde till att många

av delarna inom projektet inte höll måttet vid första visning för resten av gruppen. Koden kan ha varit feloptimerad för andra klasser som ska använda den eller så stämde inte det grafiska gränssnittet. En bättre arbetsmodell med mer prat innan kodande kan hjälpa för att lösa dessa potentiella improvisationer bland gruppmedlemmarna (är väldigt nöjd med gruppens ambitioner men mer kommunikation kan ha hjälpt).

Inkluderingar i header fil /Anders

Under implementationen så råkade jag ut för problemet cirkulär inkludering. Cirkulär inkludering innebär att en klass A har en instans av en annan klass B och därför inkluderar dess header-fil, men klassen B har en instans av klass A och behöver därför inkludera A:s header-fil. Nu har ett oändlig rekursions problem uppstått trotts att båda header-filerna har *headerguard*s. Det var svårt att tolka kompileringsfelen men det visade sig senare att det var problem med just inkluderingen. Lösningen var att använda pekare till klass istället för att skapa en lokal instans av objektet och sedan fördeklarera(forward declaration) klassen i header filen och inkludera klassens header-fil i cc-filen istället. Ett bra förhållningssätt till hur man bör inkludera hittades på hemsidan[1].

[1] http://www.cplusplus.com/forum/articles/10627/

Joystick i SFML / Albin

Eftersom att vi hann bli klara med grundkraven inom bra tid så kunde vi utveckla några av extrakraven. En av dessa krav var att spelet skulle stödja handkontroller. Så därför fick vi läsa på hur SFML hanterar inkopplade handkontroller. SFML har inga färdiga klasser du kan skapa för att hålla koll på kontroller ifrån en joystick som du kan med ett tangentbord utan vi blev istället tvungna att skapa våra egna då SFML endast har stöd för att kolla handkontrollers status med hjälp av statiska funktioner.

Resultat

Implementation av klasser /Chris

Att implementera klasserna enligt designplanen har gått relativt enkelt för de klasser som haft en tydlig färdigställd struktur i designplanen. Dokumentet har för vissa klasser inte innehållit en tydlig vision för hur dessa skulle implementeras medans det för andra delar av systemet så har klasserna inte gått att implementera på det sätt som det var tänkt från början. Detta har lett till att gruppmedlemmarna har fått implementera vissa av klasserna på ett sådant sätt att de fungerar med resten av programmet tillexempel så var klasserna tänkta att skicka referenser av *Player* och *Character* till varandra. Detta hade fungerat men det kändes oklart vad som skedde när tilldelningsoperatorn = användes, därför valde gruppen att ändra alla referenser till unique-pekare istället. Eftersom unique-pekare inte får kopieras som vanliga pekare ger kompilatorn fel om en unique-pekare kopieras. Dessutom så har en överliggande klass, tillgång till unique-pekaren sedan får underliggande klasser en kopia av unique-pekare adressen. Nu behöver inte den underliggande klassen bry sig om destruering eftersom det kommer skötas när unique-pekaren går ur scope.