Sudden Impact

Processorienterad programmering (1DT049) våren 2012. Slutrapport för grupp 1

Gruppdeltagare

Olof Björklund	880304-7136
Marcus Utter	890830-1974
Mark Tibblin	870519-7898
Luis Armando Mauricio	840911-0791

Innehållsförteckning

1. Inledning

Att föra samman två olika programmeringsspråk och få dem att fungera i balans är något som kan ha många fördelar. På detta sätt kan den rent räknemässiga kraften hos ett språk förenas med den grafiska tillgängligheten hos ett annat. Det är precis dessa två faktorer som tagits i akt när projektet Sudden Impact skapats. Projektet använder Erlang för samtidighet och Java för grafiskt interface och användarvänlighet.

Syftet med projektet är att lära gruppmedlemmarna mer om samtidighet och kraften av att förena två olika språk som Erlang och Java. Hur detta kan göras, vad för problem som kan uppkomma och möjligheterna som finns med en sådan sammanslagning.

Målsättningen med projektet har varit att få en fungerande prototyp av ett action spel att fungera. Första målet är att som ensam spelare kunna spela mot datorstyrda spelare, andra målet att flera spelare skall kunna spela samtidigt på samma terminal, och slutmålet att fler spelare skall kunna spela tillsammans över ett öppet nätverk.

Det övergripande problemet med detta projekt är att få kommunikationen mellan Erlang och Java att fungera. Detta är den huvudsakliga nöten att knäcka för projektet, samt att få de två språken att fungera med samtidighet och smidighet. Utöver detta förväntas det bli en utmaning att med hjälp av *distrubuted Erlang* få programmet att kunna köras på flera terminaler samtidigt och på detta sätt göra det möjlighet att simulera scenarion för testning.

Avgränsningar som valts är att inte fokusera allt för mycket på nätverkskommunikation till en början. Programmets interna kommunikation måste fungera felfritt, inkl. tester, innan projektet går vidare och utökas.

2. Sudden Impact

Bild 1 – Översikt

Sudden Impact är ett klassiskt action spel i western-miljö. Du styr och kontrollerar din karaktär *Klintan* som måste besegra svurna fiender i ett nostalgiskt western-landskap som exemplifieras i *bild 1*.

2.1 Världen

När spelet först startas upp är världen det första som du kommer att se, som visat i *bild 1* så är det där som alla karaktärer och objekt finns. Du kontrollerar din karaktär i världen med tangentbordet. Kontroller och dess funktionerfinns angivna i *tabell 1*.

Tabell 1 – Kontroller i världen

Knapp	**Funktion**
W	Gå uppåt
A	Gå vänster
S	Gå nedåt
D	Gå höger

2.2 Strid

När *Klintan* stöter samman med en datorstyrd fiende startas en strid. Striden sker i ett nytt fönster där de två kombattanterna slåss i tvekamp, det som sker då är att *Klintan* och fienden sakta rör sig mot varandra för att efter ett tag vända sig om och då är det den som skjuter snabbast vinner. Den som blir besegrad dör och försvinner ur spelet, segraren ges en form av belöning och får fortsätta spela. Undantaget till denna regel är när *Klintan* blir besegrad och dör, då återuppstår han och så fortsätter man spela som inget har hänt. Kontrollerna vid en strid finns angivna i *tabell 2*.

Tabell 2 – Kontroller i strid

Knapp	**Funktion**
Mellanslag	Skjuta

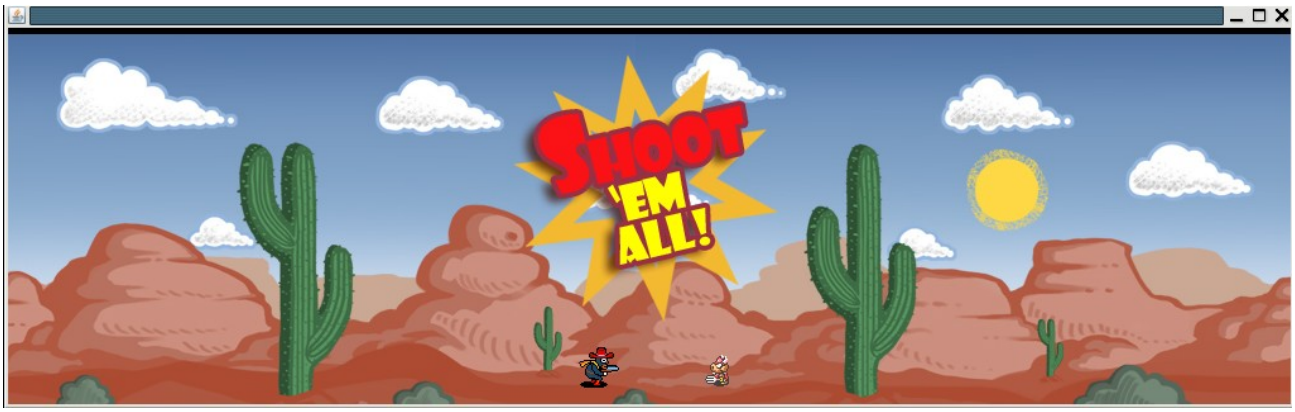

Bild 2 – Strid

När alla fiender har besegrats uppstår det ett antal fler fiender för en att besegra, och så fortsätter det in i all oändlighet, eller åtminstone tills spelet stängs ner. Vill du sluta spela stänger du helt enkelt ned fönstret med hjälp av krysset i högra hörnet.

3. Programmeringsspråk

För detta projekt har två språk valts: Erlang och Java.

Erlang har valts dels på grund av sin snabbhet gällande samtidighet och *message passing* dels i utbildningssyfte / av intresse för språket. Eftersom Erlang är ett funktionellt språk lämpade det sig väldigt bra för spelets logik och rent räknemässiga funktioner.

Java valdes på grund av dess kompatibilitet med grafisk programmering / GUI samt att man kan använda sig av ett bibliotek (som följer med Erlang) för att upprätta kommunikation mellan Java och Erlang, kallat JInterface.

Den största nackdelen med Erlang är att ingen av gruppmedlemmarna har någon direkt erfarenhet av språket, varför det kommer ta mycket tid att ta reda på exakt hur saker och ting kan göras. Framförallt för att kommunikationen med Java skall fungera bra.

4. Systemarkitektur

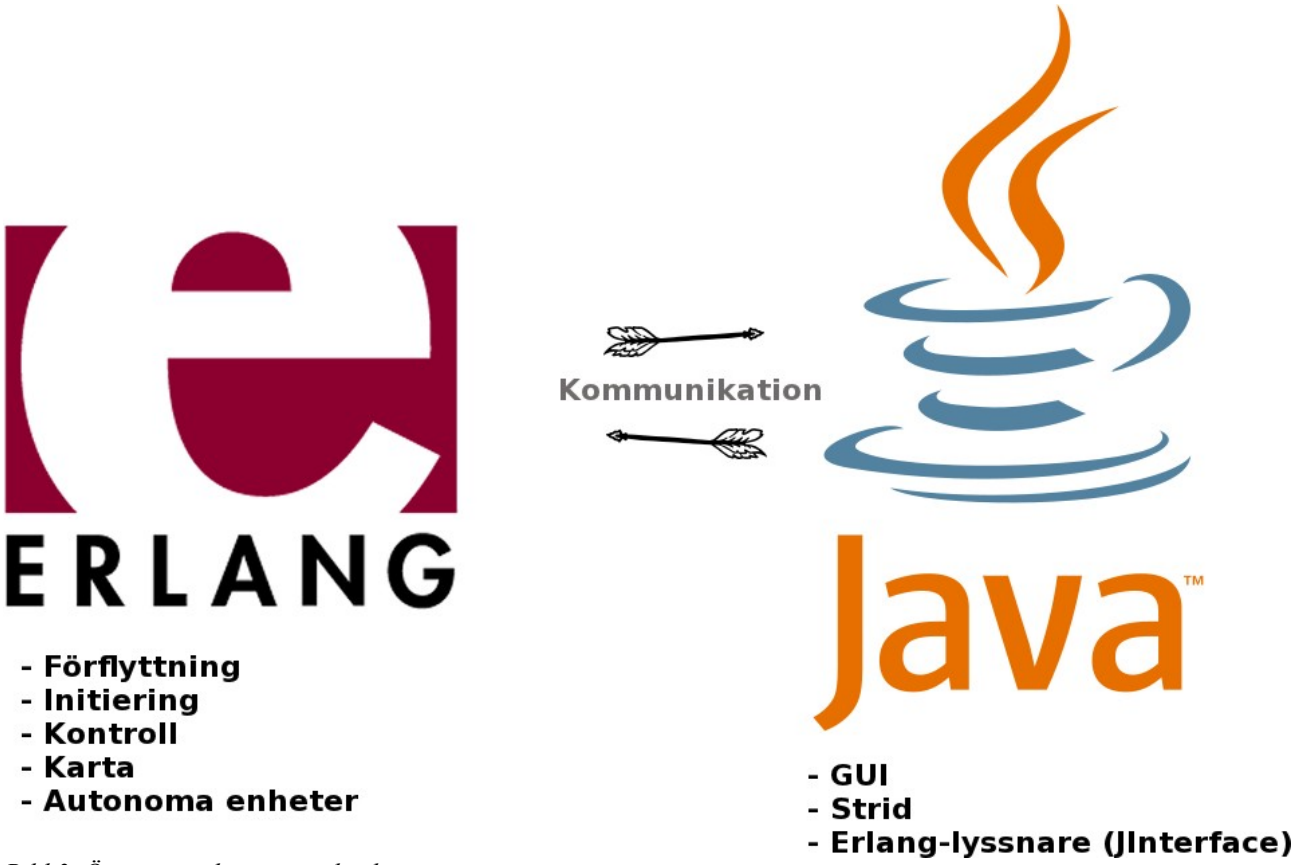

Bild 3. Övergripande systemarkitektur

I projektet används en arkitektur som bygger på kommunikation mellan Java och Erlang med hjälp av ett bibliotek från Erlang kallat JInterface[1]. Utvecklat av Ericsson tillåter detta bibliotek att kommunikation sker direkt mellan Erlang och Java. Den huvudsakliga kommunikationen sker genom att Erlang skickar tupler som meddelanden till Java och vice versa. Java hanterar GUIt samt själva tvekampen.

[1]

4.1. Systemarkitektur – Erlang -->GUI

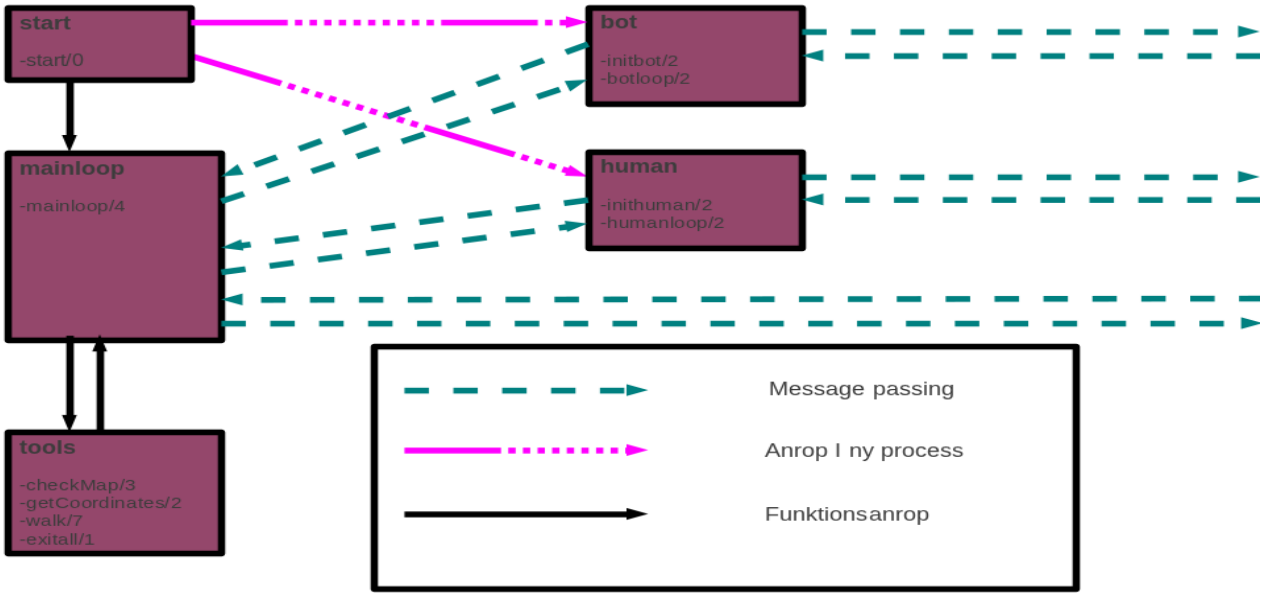

Bild 4. Systemarkitektur – Erlang

Spelmotorn (Erlang-delen) startas genom funktionen start/0 i modulen med samma namn som upprättar kommunikation mellan GUIt och motorn. Därefter initieras tre datorstyrda spelare samt en mänsklig, alla i olika processer. Efter detta anropas funktionen mainloop/4 från modulen med samma namn. Denna funktion anropar en rad olika hjälpfunktioner som vi har valt att placera i modulen tools.

En sak som är värd att notera är att vi inte har särskilt många funktioner utan vi har ett gäng processer som loopar (mainloop/4,botloop/2,humanloop/2) och talar med varandra genom att skicka meddelanden. Beroende på vilket meddelande som tas emot så utförs olika handlingar, man kan likna de olika meddelanden vid vanliga funktioner.

4.2. Systemarkitektur - JAVA

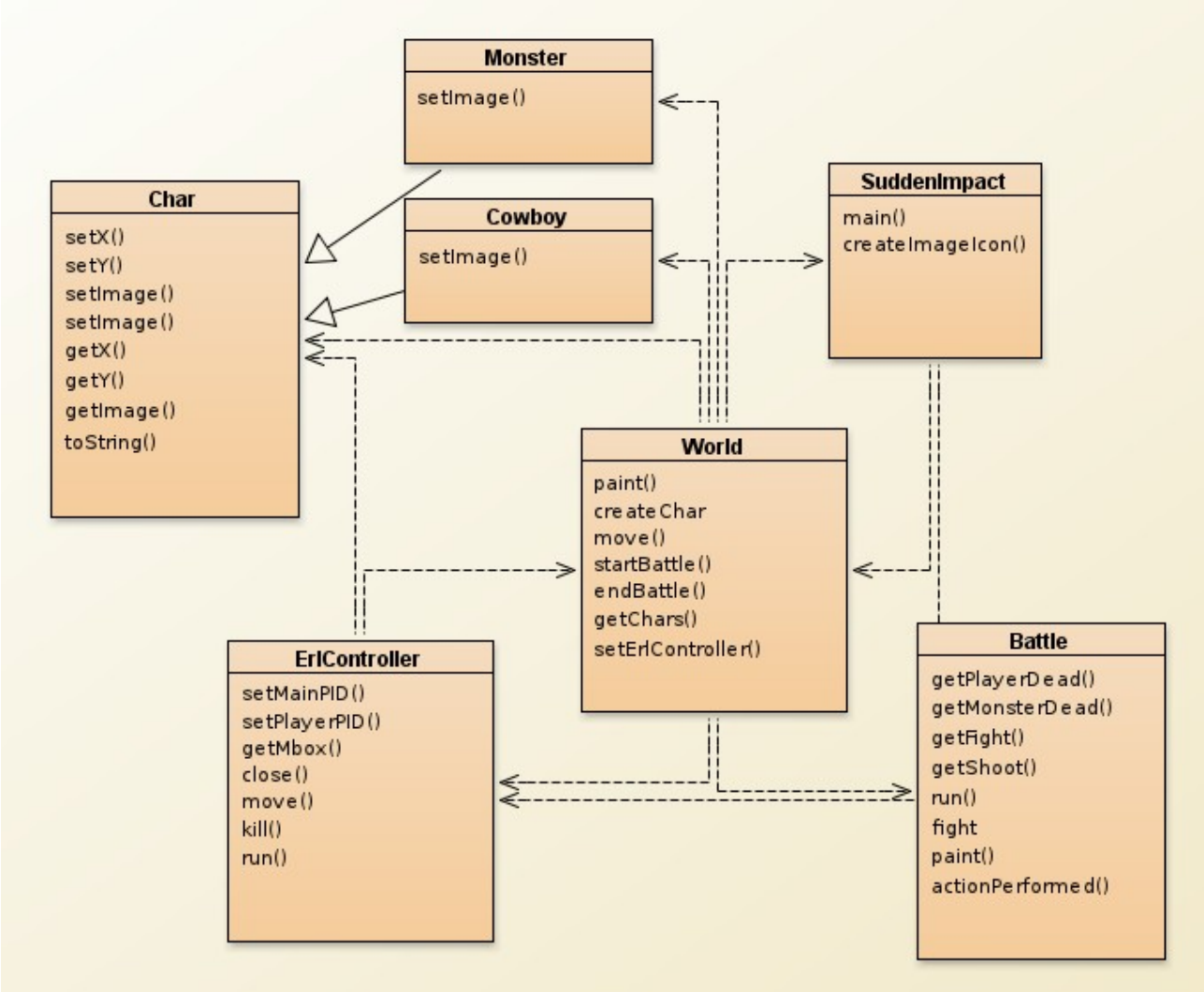

Bild 5. Systemarkitektur – JAVA

I GUIt (JAVA-delen) som drivs av huvudklassen SuddenImpact, representeras varje mänsklig spelare som objekt från Cowboy klassen och varje datorstyrd spelare som ett objekt från Monster klassen. Dessa klasser är subklasser från Char klassen. Den grafiska kartan innehållandes spelare definieras av World klassen och tvekampen sker i Battle klassen. Utöver de nämnda klasserna har vi ErlController klassen, där kommunikationen mellan GUIt och spelmotorn sker.

5. Samtidighet (concurrency)

I projektet används *message passing* i samband med *processer* för att skapa samtidighet / concurrency. T.ex. är varje individuell spelare en unik process som har sitt eget PID (process identification number) vilket kan användas för att kommunicera med just den processen. På samma sätt har Main-processen sitt eget PID vilket bland annat används för att kommunicera med GUI.

Message passing i Erlang underlättar kommunikationen med GUI avsevärt då meddelanden kan ”bangas” mellan Java och Erlang oberoende. Om något händer i Java som direkt skall meddelas till Erlang kan vi enkelt skicka ett meddelande som snappas upp av en väntande process i Erlang.

De delar av systemet som använder sig av message passing är framförallt Erlang-delen. Implementationen är gjord så att kommunikationen mellan Java och Erlang huvudsakligen sker från Erlang till Java, alltså att Erlang skickar meddelande till Java som reagerar på dessa och målar upp förändringar på GUI. Till viss del sker kommunikationen även åt andra hållet, t.ex. då spelets stängs ned eller när en spelare flyttar på sig.

Delar av systemet som inte utnyttjar sig av samtidighet är t.ex. poäng-registrering vilket sker oberoende av annan kommunikation. Även när en strid startas sker de som en separat process vilken inte kommunicerar med de övriga systemet via concurrency.

Ett behov av synkronisering uppkommer då en strid skall startas. Efter att två spelare stött ihop och skall börja en strid kan inte en tredje spelare tillåtas stöta ihop med någon av dessa. Detta löser vi med hjälp av ett tillstånd *freeze* som avgör om en spelare är i strid (freeze) eller inte i strid (unfreeze).

Vidare används synkronisering för spelarnas positioner i världen vilket representeras av ett ständigt flöde mellan Java och Erlang.

Fördelar med den valda implementationen är att kommunikationen mellan processer sker väldigt snabbt tack vare message passing. För övrigt underlättar den avsevärt för testning då scenarion kan specificeras och testas för trovärdighet. Nackdelen med att använda två olika språk är att biblioteket *JInterface* krävs för att kommunikationen skall fungera. Om ett fel uppstår i Java kommer detta att påverka vad som händer i Erlang och vice versa.

Ett alternativ till den valda implementationen hade kunnat vara att skriva hela programmet i Java eller Erlang. Detta skulle minska risken för fel men projektet skulle förlora lite av sitt syfte, att lära mer om kommunikation mellan två olika språk. Skulle implementation gjorts i t.ex. enbart Java skulle trådar komma att användas istället för processer och message passing, men eftersom endast en tråd används i Java finns ingen risk för deadlock, vilket är väldigt positivt!

6. Algoritmer och datastrukturer

Hela spelvärlden representeras som ett tvådimensionellt koordinatsystem. När en karaktär förflyttas skickas nya X- och Y-koordinater till Erlang vilka registreras i en hash-tabell. Denna tabell används senare för att identifiera om en viss koordinat är upptagen eller inte och på så vis avgörs om en förflyttning kan genomföras eller ej.

I Erlang har en funktion övergripande betydelse, *mainloop/4*. Denna loop registrerar knapp-tryckningar vilka identifierats i Java och skickats till Erlang. Loopen förflyttar spelare till nya koordinater, registrerar nya spelare samt avregistrerar de som förlorat en tvekamp.

Funktionen *start/0* i Erlang-modulen start skapar nya spelare (datorstyrda som mänskliga) samt bestämmer ett korrekt *HostName,* i prinip ett PID, som används för kommunikationen mellan Erlang och Java.

```
{ok,Host} = inet:gethostname(),
  HostFull = string:concat("sigui@",Host),
```

Kod-exempel 1 - *HostName*

Denna funktion används sedan i make-filen för att starta spelmotorn

```
start: all
        (cd ebin && erl -sname "player" -eval 'start:start()')
```

Kod-exempel 2 – m*ake start*

Modulen för en mänsklig respektive datorstyrd spelare är i princip densamma, med undantaget att en datorstyrd spelare får sin rörelserikting slumpvis. En knapptryckning registreras i Java med hjälp av en lyssnare som skickar tillbaka en tupel innehållandes {move, ”tangent som tryckts”} till Erlang. Denna registreras och spelarens position uppdateras. T.ex. skulle en registrering av att knappen ”w” tryckts tolkas på följande sätt:

```
{move,w} ->
        MainPID ! {walk, self(), ["w"]},
           humanloop(MainPID, GUIPID);
```

Kod-exempel 3 – move

Där MainPID är PID till main-processen och GUIPID är PID till GUIt (JAVA). När spelaren erhållit sina nya

koordinater skickas dessa tillbaka till GUIt:

```
{newposition, {CoordinateX,CoordinateY}} ->
        GUIPID ! {move,human,self(),self(),CoordinateX,CoordinateY},
```

Kod-exempel 4 – newposition

Erlang och JAVA kommunikationen sköts i klassen "erlController" som använder JInterface. Klassen har en *listener* som under hela sin körning lyssnar på allt Erlang skickar till JAVA, bland annat koordinater och när tvekamper ska ske. Dataflödet mellan språken sker via objektet kallade "mbox" asynkront. Meddelanden plockas genom noder från Erlang (processer) som identifieras genom deras PID.

Strukturen på meddelandet som skickas till JAVA-delen är nästan alltid enligt följande (atom,atom,PID,PID,long,long), undantagsfallet är det absolut första meddelandet som skickas från Erlang då endast mainprocessens PID skickas. Anledningen till denna struktur är att det enkelt ska gå att behandla alla tupler som tas emot på samma sätt i JAVA-delen se Kod-exempel 5.

```
msg = (OtpErlangTuple)object;
command = (OtpErlangAtom)msg.elementAt(0);
player = (OtpErlangAtom)msg.elementAt(1);
player1PID = (OtpErlangPid)msg.elementAt(2);
player2PID = (OtpErlangPid)msg.elementAt(3);
x = (OtpErlangLong)msg.elementAt(4);
y = (OtpErlangLong)msg.elementAt(5);
```

Kod-exempel 5 – ErlController

En följd av vår implementation är att oftast är en eller flera element i tupeln som skickas oväsentliga. Exempelvis behövs inga element av typen long vid en tvekamp, ändå skickar vi med två stycken som inte används, för att kunna koda av meddelandet som i exemplet ovan.

7. Förbättringar

7.1. Grafik

Grafikmässigt skall det se smidigare ut med annan typ av implementation men det som är aktuellt är ett gränssnitt som funkar för att demonstrera hur det eventuellt kan komma att se ut framöver. Allt skall ske i en och samma "frame" dessutom skall ett GUI finnas för val av karaktärsutseende, spelnivå, online-gaming och inställningar.

7.2. Multiplayer

Multiplayer spelandet skall ske med en client/server uppkoppling där val av DM (Deathmatch) eller co-op (Co-operative) kan ske. Servern skall göras i Erlang där vi har en uppsättning av att all logik ska skötas i Erlang. Detta skulle göra att nuvarande systemarkitektur uppdateras det vill säga i princip skall all logik förbli i Erlang för att underlätta synkroniseringen via nätverket. Det nuvarande "battle" logiken flyttas till Erlang då nätverkskommunikationen ska skötas i Erlang. Mer ingående till multiplayer upplevelsen skulle det även bli lämpligt att implementera en chatfunktion.

7.3. World

För en bättre upplevelse skall en större värld skapas. Detta görs genom att vid dåvarande spelyta försöka att lämna det längs utkanterna av "rutan" och då skall en ny yta dyka upp och om så önskas ny typ av miljö med aktuella spelare, botar och hinder som förekommer på den nya spelytan. Detta skulle medföra att nya mekanismer och parametrar implementeras på ett smart sätt för att koordinera objekten på spelfältet för att inte skapa konflikter.

7.4. In-game

7.4.1. Duel (battle)

Först och främst finns ett flertal idéer om självaste “battle”. Den nuvarande utgår från “one-shot, one-kill” med en hundraprocentig chans för träff. Denna implementation som nämndes i tidigare punkt skall all statistik/spellogik skötas concurrently i Erlang-delen för att underlätta synkroniseringen av spelhändelser. Följande punkter ska förbättras:

- I duellen skall en “healthbar” införas som representerar hur stor livslängd spelaren har eller i liknande former där antalet träffar mot sig avgör livslängden.
- Självaste skjutandet skall utgå från att man siktar in sig mot ett mål så bra som möjligt för att träffa fienden.
- Efter stridens slut så ska en statistik sparas och poppa upp med värden, som “Accuracy”, “Hits”, “Kill-style” med mera, för fortsatt utvecklande färdigheter under spelets gång.

7.4.2. Wild west

Ett “Wild West booze fight” som innefattar en all-out strid för alla som är på plats för skjuta ner varann. Man ska kunna röra sig runt I antingen öken miljö eller stad med liknande förutsättningar på livslängden med en så kallad ”healthbar”.

7.4.3. Minigaming

Beroende på område och spelstadiet ska man ha olika typer av “minispel” förekomma med olika typer av uppdrag relaterat till vilda västern som, till exempel, lassokastning.

7.4.4. Bot

En mer utvecklad AI upplevelse på botarna för att få mer interaktion med spelaren. Ett exempel är att en bot hamnar i “frenzy mode” och börjar jaga en.

8. Reflektion

Det mest intressanta var att jobba med två språk i ett och samma projekt. Först och främst att se hur pass enkelt det var med messagepassing mellan språken och utnyttja båda språkens fördelar. I och med Erlang var ett nytt språk för alla i gruppen och användningen av två språk var nytt så angrep vi detta genom att dela upp oss två och två. Detta för att snabbare få en bättre helhets bild på hur arkitekturen eventuellt skulle se ut. Arbetet i sig blev som mest effektivast vid bestämda möten för att hacka kod och diskutera på vad som behövde göras vilket i efterhand sågs som ett bra val av arbetssätt för snabb feedback på funderingar.

Troligtvis skulle spelet hade kunnat göras snyggare, mer avancerat och snabbare att utveckla om enbart Java använts. Det var ett klart val ur utbildningssynpunkt att inkludera Erlang. Att det skulle vara mer avancerat menas att en mer komplexitet skulle förekomma, det vill säga en mer utvecklad nivå på utmaningar för en förbättrad spelupplevelse. Då det upplevdes inledningsvis oklarheter om arbetsgången och vad som förväntades levereras så var de svårt att lägga upp en rimlig tidsplan på slutprodukten. I efterhand anses detta att de borde ha varit tydligare om hur den faktiska tidsplanen för deadlines och förväntningar skulle vara.

Ett problem moment som uppkom under början av projektet var användning av ”git” repository där gruppen inte hade en tillräcklig kunskaps nivå på dess funktionaliteter. Det skulle tidsmässigt sparats mycket tid om det inledningsvis fanns en ”crash-course” för de grupper som kände brister på en viss typ av repository eller diskussions forum på detta område, vilket i slutändan dök upp men alldeles för sent. ”The hard way is the best way” blev drivkraften i slutändan.

9. Installation och fortsatt utveckling

Av Java så används den senaste versionen och av Erlang används version R15B. JavaDoc och Edoc används för att generera dokumentationen av koden. För testningen använder vi oss av JUnit och EUnit för Java respektive Erlang.

Källkoden finns tillgänglig på Github. För att ladda ner den så kör Ni följande kommando: “git clone https://[KONTONAMN]@github.com/isqb/POP.git”, där [KONTONAMN] ska bytas ut till det användarnamn som Ni har på Github.

Som det ser ut så finns all kod under src/Game, där inne finns det en till katalog, Graphics, som innehåller alla bilder som används. Katalogen doc innehåller rapporten för denna projekt. Katalogen ebin finns för att innehålla alla beam filer som skapas när Erlang koden kompileras

För att starta upp programmet ställer Ni er i projektekts katalog och skriver ”make start” i kommandotolken. Då körs först en jar-fil innehållandes hela JAVA-delen av projektet, sedan startas spelmotorn i en Erlang-prompt.

Vi genererade en jar-fil genom att använda NetBeans inbyggda funktion, Eclipse har också detta stöd. Vid vidareutveckling krävs att man genererar en ny jar-fil för att JAVA-delen om man inte vill ändra i make-filen. Det går annars bra att manuellt starta igång delarna var för sig, då kör ni först JAVA-delen i exempelvis NetBeans och anropar efter det funktionen start/0 i modulen start i en Erlang-prompt.