INDA11 - KTH

Projekt - Slutrapport

Sebastian Wigren, Philip Rendén

5/11/2012

Innehållsförteckning

Programbeskrivning

Avsnitt A
Samma som innan.

Avsnitt B
Programmet är fortfarande ett spel som fått animation gjord från ett perspektiv snett ovanifrån. Man kan röra sig pixelvis vilket innebär stor rörelsefrihet samt man kan även röra sig i höjdled nu, genom t.ex. trappor. Annars är det lika, man växlar mellan scener och jagas av fiender osv.

Avsnitt C
I stora drag har vi gjort det som vi planerade att göra. De skillnader som finns är att en höjdsimulering lades till de riktningar som man kan röra sig i samt att en kamera kan röra på det fönster genom vilken man ser spelvärlden.

Användarbeskrivning

Avsnitt A
Samma som innan.

Avsnitt B
De användare vi tänkt oss är personer som har viss datorvana, som normalt spelar spel.

Avsnitt C
Det har inte ändrats.

Användarscenarier

Avsnitt A
Samma som innan.

Avsnitt B
Ena scenariot är när man först startat spelet och man hamnar i en meny. I bakgrunden ser man en stad och till vänster finns text alternativ som leder en vidare till bland annat det riktiga spelet. Man förflyttar sig i menyn med hjälp av piltangenterna.

Det andra scenariot är en scen med ett par hus/block samt trappor som är byggda ovanpå varandra. Med hjälp av piltangenterna kan man styra en karaktär runt i den här scenen samt upp på block genom trappor. När man rör sig byter karaktären bilder vilken ger en illusion av att den går.

Avsnitt C
Det har inte ändrats så mycket, främst design.

Testplan

Avsnitt A

Samma som innan.

Avsnitt B

Testplanen gick i stort sett ut på att man som testare spelade igenom spelet/demot och noterade vad som gick bra och vad som gick mindre bra. Att utgå från de två användarscenariorna nämnda här ovanför var en bra idé eftersom det isolerade själva spelmiljön från menymiljön. Testerna visade på att menysystemet fungerar i stort sett fläckfritt medan själva spelet har ganska få funktioner tillgängliga.

En punkt som har stressats som ett problem av flera testare är att scener som är lite större (>15 entiteter) går ganska långsamt och segt.

Avsnitt C

Eftersom att spelet inte har så pass många spelfunktioner förutom att röra sig runt i världen så är det främst det som är den stora skillnaden från prognosen. Dessutom så har införandet av en ytterligare dimension gjort att själva förflyttandet av en karaktär/objekt i en scen blivit mer komplicerat och därför har det testats mer noggrant.

Programdesign

Avsnitt A

Input

Ett av de viktigaste systemen för ett spel är själva hanteringen av spelarinput, dvs. input från tangentbord och mus. Spelet måste t.ex. kunna särskilja mellan olika knapptryck och bestämma var musen senast befunnit sig. I stort sett allt i spelet utgår från spelarens input så det är viktigt att denna del är högst funktionell.

Kollision

För att man ska kunna röra sig i spelmiljön på ett intressant och naturligt sätt krävs det att man hanterar kollisioner mellan karaktärer och statiska objekt på ett effektivt sätt. Spelaren ska t.ex. inte kunna gå igenom väggar. Används projektiler (t.ex. kulor) som vapen måste även dessa kollisioner testas mot fienderna på ett funktionellt sätt, samt om man ska plocka upp nycklar eller dylikt.

Grafik

Det behövs implementeras ett grafiskt system som kan hantera ritandet av bilder och enkla animationer. Eftersom bilderna ligger på hårddisken behöver systemet kunna ladda upp dem i minnet och hålla koll på till vilken objekt instans som de tillhör. Ska karaktärer dessutom innehålla animationer (när de t.ex. springer) måste den tid som förflutit också tas i beaktande så att allt flyter på i en jämn takt. En annan aspekt som är viktig är att hålla koll på i vilket djup varje bild befinner sig på så att inte en bild längre ”ned” ritas ovanpå en bild längre ”upp”.

Meny

En meny ger spelet både en naturlig ingångspunkt samt ett snyggt inkapslande som ökar dess professionalitet. En meny ska tillhandahålla ett par olika meny alternativ som man som spelare ska kunna välja med hjälp av t.ex. pil-tangenterna. Det är också härifrån som man ska kunna starta det ”riktiga” spelet och avsluta programmet.

Tidmätning

En viktig aspekt i spelkonstruktionen är hur man mäter tid. Tidmätning behövs både i och med animationer samt karaktärers rörelsescheman. Man vill också att ett spel har en jämn och fin uppdaterings- och ritfrekvens så att det inte uppstår plötsliga hopp med grafik och liknande vid en eventuell nedsegning av datorn. Har man en välfungerande timer får man dessutom möjligheten att använda sig av tid i en myriad av andra olika scenarier, t.ex. att planera rutter för fiender samt andra tidsbaserade events.

Avsnitt B

Det är svårt att koka ned ett spel till bara ett fixt antal klasser och peka ut dem som viktigast eftersom det är så många som samverkar till helheten. Alla klasser som skapats har dock ett syfte och det är framförallt genom detta som man kan dela in dem på ett någorlunda fungerande sätt.

Alla spel, likväl vårt, delar en gemensam bas i ett antal olika funktioner som måste implementeras. Funktioner som tillhör denna kategori är inläsning av input från användaren i form av t.ex. mus- och knapptryck på tangentbordet, någon sorts metod att rita upp bilder och grafik på skärmen samt en grundläggande uppdateringscykel som ger alla instanser i spelet en möjlighet att bland annat förflytta sig. Utöver dessa brukar även grafiska gränssnitt vara vitalt för ett spel, trots att dess roll inte är lika central och avgörande för användaren som i t.ex. ett kalenderprogram.

En central tanke i själva uppbyggnaden av de olika systemen och klasserna i spelet är att varje instansierbar klass bär på tre viktiga metoder; handleInput(), update() och draw(). Det är i regel endast genom dessa metoder som respektive funktion i spelet (dvs. hantera input från användaren, uppdatera instansen och rita upp instansen) sker och varje klass får snällt vänta på att någon längre upp i den hierarkiska kedjan kör metoden åt dem. På grund av denna grundläggande design är hanteringen av objekt i spelet, vare sig de är en meny eller en spelkaraktär, styrd av någon ovanifrån och de kan därför inte hitta på något oväntat som möjligen skulle kunna störa någon annan aktiv instans. Spelmotorn är pga. detta väldigt centraliserad, något som t.ex. gör det lättare att debugga problem i och med att man vet var man ska börja leta.

Ett exempel på denna typ av struktur är i hur varje nivå, eller scener, i spelet är utformad rent tekniskt. Längst upp till hierarkin är det en sk. SceneManager som håller koll på alla scener och bestämmer vilken utav dem som befinner sig i fokus. Den scen som för tillfället innehar fokus, dvs. den värld som spelaren befinner sig i och springer runt i, tillåts hantera input, uppdateras och ritas en gång vardera varje cykel i spelvarvet av dess SceneManager. Varje scen ger i sin tur varje underliggande entitet i scenen (som t.ex. en bokhylla) möjlighet att göra detsamma. Detta mönster fortsätter hela vägen ned i hierarkin tills man kommer till ett objekt som inte har några att delegera till.

Input

Ett mycket vitalt moment i varje spel är hur man hanterar input från användaren. I ett vanligt program får man i dessa lägen ganska mycket förspänt tack vare olika grafiska gränssnitt som redan finns klara i API:n och redo för att snappa upp både knapptryck och mustryck. Detta måste man implementera själv i en något större utsträckning om man har fått för sig att skapa ett spel. Grundtanken är dock detsamma som med de inbyggda gränssnitten i Java och det visade sig vara väldigt enkelt och smärtfritt att registrera användarens input, något vi gjorde genom sk. KeyListeners och MouseAdapters. Eftersom det fönster som ligger som grund till vårt spel är en JFrame så kunde vi enkelt haka på vår egenutvecklade inputlistener klass till den och på så sätt få tillgång till alla de mus- och knapptryck som användaren utför.

Allt som behövdes göras förutom att implementera de gränssnittsberoende metoder som krävs av en listener-klass var att spara ned knapptrycken som skedde i en vektor och ge alla andra klasser i spelet tillgång till denna. Ett par enkla hjälpmetoder senare och man kunde fråga input-klassen om en specifik knapp var nedtryckt eller om höger musknapp hade släppts än.

Kollision och förflyttning

Den i särklass största delen i uppdateringsmetoden för alla entiter i spelet, dvs. alla synliga objekt i spelvärlden, består i att förflytta dem och hantera de kollisioner som uppkommer i och med denna förflyttning. Båda dessa två högst väsentliga delar i spelet styrs av en central fysik motor där alla entiteter finns sparade i en lista, samt deras rektangulära kroppar genom vilka kollisioner undersöks.

Vid varje uppdateringscykel går fysikmotorn igenom listan med entiteter och letar efter kollisioner. Om en kollision har upptäckts räknas riktning och magnitud på den överlappande vektorn ut och de två objekten kan säras på igen. Slutligen adderas en sammanslagning av tillagda krafter till vart och ett av objekten i fysikmotorn. Dessa krafter har uppstått genom bland annat gravitation eller krockar mellan objekt. Även huvudkaraktären förflyttas genom att knapptryck kopplas till olika krafter.

Grafik

Det grafiska är i många fall den viktigaste aspekten för ett spel, något som således gör det till en mycket viktig del i utformandet av en spelmotor. Till vårt spel nöjde vi oss med de inbyggda grafiska metoderna som finns i Javas 2D-paket, även om externa bibliotek så som Slick2D intresserade oss.

Precis som med hanteringen av input så fanns redan grunden med 2D-grafik implementerad och klar i och med att vi använde oss av en JFrame som fönster. Allt man behövde göra var att få tillgång till dess egen Graphics2D-klass och använda den för att rita upp vår egen grafik, något vi gjorde genom att skicka den vidare ned genom vår spelhierarki.

Att ladda in bilder som befann sig på hårddisken visade sig inte heller vara något problem så det gick väldigt snabbt att få fram bilder på skärmen. Nästa steg i utformandet av spelets grafikklasser blev lite större och bestod av att spara ned de laddade bilderna så att de kunde kommas åt på ett enkelt sätt samt att man skulle kunna skapa animationer med dem genom att byta mellan dem väldigt fort. Vårt huvudsakliga fokus när vi designade dessa klasser var att det skulle vara enkelt att lägga till bilder till ett objekt och rita upp dem; så mycket som möjligt skulle ske automatiskt.

På grund av att spelet utnyttjar en värld med 3-dimensioner men bara kan använda sig av 2D-bilder har vi varit tvungna att tänka på ett per ytterligare saker än vad vi hade behövt göra annars. En sådan aspekt är i vilken ordning bilderna ska ritas upp på skärmen. För att lösa detta problem var vi tvungna

att emulera en z-buffer, dvs. vi använde oss av samma teknik som seriösa 3D-spel. Men på grund av att vi skapade denna funktion själva så drar den väldigt mycket kraft och segar därför ned spelet ordentligt om tillräckligt många bilder ska ritas upp på skärmen på samma gång.

Meny

Menysystemet som vi har implementerat är väldigt likt det som styr spelets olika scener. Varje ”tillstånd” som spelet befinner sig i (pausemeny, huvudmeny, spelvärld etc.) kallar vi en ”screen” och det finns en ScreenManager som håller koll på alla dessa och bestämmer vilken som ska ritas överst och så vidare. Spelet börjar med sin huvudmeny-screen aktiv och beroende på vilket alternativ man väljer så skapas en ny screen som positioneras främst. Eftersom bara en ”screen” kan vara aktiv åt gången (med undantag för t.ex. pausemenyn) så ritas och uppdateras bara en ”screen” åt gången.

Tidmätning

För att försäkra oss om att spelet genomgår sina cykler relativt regelbundet så bestämde vi oss för att det skulle krävas en klass som mätte tid. Det fanns två tydliga alternativ att välja mellan när det kom till att hantera tid; antingen använde man sig av en utav Javas inbyggda timers och lyssnade på dess events eller så byggda man en egen klass som emulerade en timer genom att kolla upp tiden väldigt ofta och jämföra den med den tid som redan fanns sparad. Vi valda att använda oss av det sistnämnda alternativet huvudsakligen för att detta alternativ var mer flexibelt och gav oss fler alternativ. T.ex. kunde vi använda samma timer för alla instanser i spelet samt för att reglera spelets huvudcykel eftersom allt den gör är att spara ned tiden som gått på ett åtkomligt samt enkelberäknat sätt. En eventbaserad timer stödjer i och med sitt bestämda intervall bara en typ av tidscykel och vill man använda sig av en annan så behöver man istället koda sig runt problemet vilket är onödigt omständligt.

Avsnitt C

Den enda skillnaden i vilka delar och klasser som vi trodde skulle bli viktiga och de som verkligen blev viktiga för spelet bestod mest av vissa små detaljer, detaljer som uppenbarade sig först när vi brottades med problemen på riktigt. Ett exempel på detta var valet mellan vilken typ av timer vi skulle använda oss av.

Tekniska frågor

Avsnitt A

Samma som innan.

Avsnitt B

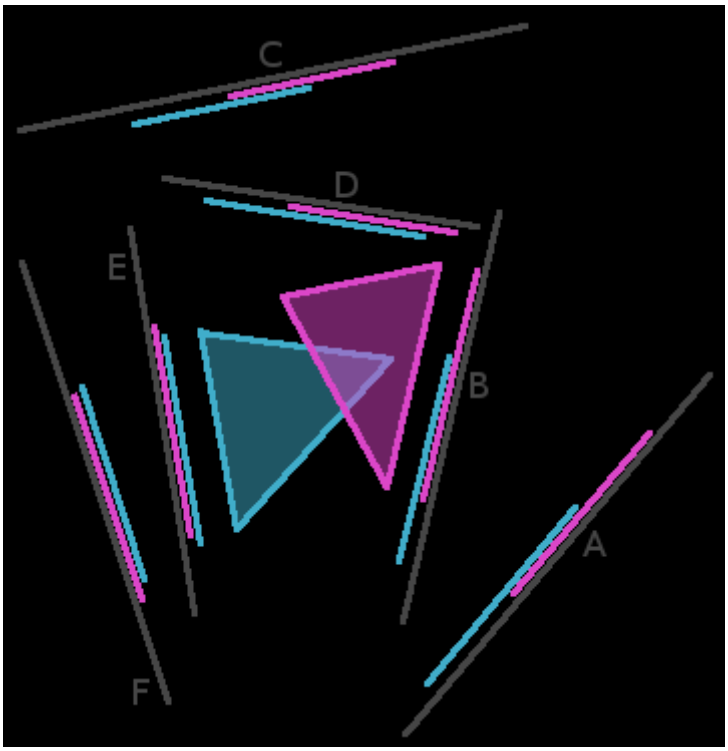
De tekniska problem som vi stötte på som var extra kluriga var följande:

Kollision

Hur kollisioner skulle hantering var en teknisk fråga som vi ställde oss väldigt tidigt i projektets idémässiga utvecklingsfas. Det optimala var i fall det hela skedde så mycket som möjligt ”behind-the-scenes” och därmed inte var något man riktigt behövde bry sig om när man designade och kodade mer högnivå klasser. Eftersom vårt spel inte bara nöjer sig med att lura spelaren att det existerar en tredje dimension som de flesta andra ”topdown/isometric” spel gör utan verkligen implementerar

den så blir självklart kollisionshanteringen snäppet svårare att genomföra än vad den annars skulle ha varit.

Kollisionssystemet utgår ifrån att varje entitet som vill kunna kollidera med omvärlden skapar sig en kropp och notifierar fysikmotorn om dess existens. Denna kropp har en tredimensionell rektangulär form, över vilken man sedan målar en bild för att lura en att det i själva verket är bilderna som krockar. Kollisionstestningen utgår från en algoritm som kallas SAT[1] (Separated Axis Theorem) som räknar ut den minsta vektorn som två 2D-kroppar överlappar varandra och drar isär dem genom denna. Detta sker genom att man projekterar varje sida av en konvex form på en endimensionell axel och mäter själva överlappandet mellan formerna. Om inget överlappande upptäcks så har ingen kollision ägt rum. Denna algoritm är utformad så att den ska vara snabb trots många kroppar som testas. Denna snabbhet bibehåller den genom att man i de allra flesta fallen inte behöver iterera igenom en forms alla sidor för att försäkra sig om att två former har kolliderat utan det räcker med att de överlappar på en axel så överlappar de på alla.



Trots att man även kan använda sig av denna algoritm vid 3D så valde vi att inte bygga ut vår SAT-metod så att den skulle stödja detta. Vi tänkte istället att eftersom alla våra former skulle vara rektangulära med varierande höjd så skulle det vara lättare om vi bara hittade höjdpunkten för en kollision och sedan utförde en SAT för de 2D-lagren som då skulle uppstå av deras 3D-kroppar. Detta visade sig fungera väldigt bra, men det utgjorde ändå ett problem genom att man inte kunde röra sig mellan olika höjdnivåer i spelet på ett naturligt sätt. På grund av att vi bestämt oss för att bara använda rektangulära kroppar kunde vi inte skapa trappor eller sluttningar och således skulle man inte kunna röra sig uppåt i spelet utan bara nedåt med hjälp av gravitation.

[1] http://www.codezealot.org/archives/55

Lösningen till detta problem blev att vi skapade specialfall med stående trianglar, dvs. vi utökade alltså en kropps olika formalternativ från bara rektanglar till även trianglar. Fyra typer av trianglar finns nu definierade; topp till höger eller till vänster samt topp uppåt eller nedåt. Genom att räkna ut höjden annorlunda för kroppar som inte var rektangulära löste vi alltså även detta problem och man kunde nu även röra sig uppåt i spelet med hjälp av bland annat trappor.

Djupsortering av grafik

I ren 2D är sorteringen av element på skärmen i förhållande till deras djup inte någon särskild svår övning. Lägger man däremot till en extra dimension märker man snart att sorteringen blivit aning mer komplicerad, till och med så komplicerad så att vi i vårt fall inte hunnit lösa problemet fullt ut än. En snabb sökning på internet om ämnet ger en dock åtminstone två populära alternativ; målarens algoritm och z-buffer.

Målarens algoritm går ut på att man för varje objekt som man ska rita upp på skärmen beräknar avståndet till från en given kamera position, dvs. från den position man som användare befinner sig i relation till det som ritas upp i spelet, och sedan sorterar dem efter detta värde. Resultatet blir en lista med objekt som man enkelt kan iterera över och rita i tur och ordning.

En z-buffer är vad dagens 3D-spel i regel använder sig av. Istället för att beräkna vilket objekt som ska ritas framför ett annat, vilket kan leda till komplikationer vid vissa överlappningar, räknar en z-buffer ut ett djup för varje pixel i bilden som man vill rita och jämför det med en redan tidigare sparad pixel. Visar det sig att den nya pixeln befinner sig närmare kameran och därmed användaren än den tidigare pixeln skrivs denna över med den nya informationen. I jämförelse med målarens algoritm leder detta till att bilder/former kan delvis överlappa varandra utan att det vållar några problem, något som det mycket väl skulle kunna göra med målarens algoritm.

Använder man sig av Javas 3D-grafik har man delar utav detta redan inbyggt från början. Tyvärr så använder vi oss uteslutande av Javas 2D-grafik och det finns alltså inte någon enkel implementering utav detta. Vår implementation använder sig av ett Composite-objekt som kopplas till Graphics2D-klassen för att bygga ihop en z-buffer och på så vis simulera djup. Metoden fungerar med undantag för ett par smärre grafiska artefakter då och då, men den har tyvärr visat sig vara ordentligt långsam för flera objekt. Eftersom att en z-buffer gör beräkningar på varje pixel hos varje bild som vill ritas upp så är det väldigt lätt att sega ned metoden och ganska svårt att få den snabb och effektiv. Vill man kunna krama ur maximalt ur Java2D måste man ha stenkoll på vilka typer av bilder man använder sig av och helst vill man dessutom få dem hårdvaruaccelererade, dvs. spara ned så mycket som möjligt direkt i grafikkortets minne. Vi misstänker att vår metod kontinuerligt får skicka data fram och tillbaka mellan datorns RAM-minne och grafikkortets minne och att detta är något som bidrar till att spelet går långsamt.

Bildflimmer

Ett tidigt problem som vi stötte på var irriterande bildflimmer varje gång vi försökte rita upp något på skärmen. Problemet låg i att vi ritade upp allting direkt till skärmen när vi kände för det, dvs. med ojämna mellanrum, och detta i maskopi med att en JFrames egna ritfrekvens inte är regelbunden gjorde så att de olika ritfrekvenserna skar sig mot varandra med resultatet att bilden flimrade. Lösningen till detta låg i en teknik som kallas ”double-buffering”. Denna teknik går ut på att man använder sig av två bilder för att rita upp sitt spel; en skärmbild (front-buffer) och en bakbild (back-buffer). Man vill rita allt under en cykel till den bild som ligger utanför skärmen, dvs. back-buffer, och

sedan, när man är helt klar, byta man plats på dem så att back-buffer blir front och vice versa. Detta försäkrar en att man inte råkar ut för att det ”skär” sig mellan de olika uppritningsfrekvenserna.

I Java finns det en klass som heter BufferStrategy som hanterar mycket utav själva bytet mellan de olika bilderna på egen hand. Allt man behöver göra är att få tag på det Graphic2D-objekt som är kopplat till den nuvarande cykelns back-buffer och rita till den. När man sedan är klar säger man till BufferStrategy och den byter plats på dem åt dig. Enkelt och behändigt.

Arbetsplan

Avsnitt A

Samma som innan.

Avsnitt B

Vår arbetsplan har i stort sett varit oförändrad genom projektets lopp, i alla fall gällande hur vi har delat upp arbete emellan oss. Några utav de moment vi planerat fortlöpte i en annan ordning än den vi ursprungligen tänkt oss men i det stora hela så har arbetsplanen följts. Det har dock lagts ned mer tid på grundläggande infrastruktur kodning (t.ex. fysikmotor) än t.ex. utformning av ett melee-system eller dyl..

Avsnitt C

Arbetsplanen förändrades som sagt ganska lite men de förändringar som skedde berodde oftast på att vissa delar tog längre tid och krånglade mer än vad vi ursprungliga hade föreställt oss. Vi underskattade helt enkelt det jobb som krävdes för att bygga upp den grundläggande infrastrukturen. Hade vi kört med ett extern bibliotek så som Slick2D istället för att bygga ett eget så kanske vi skulle ha följt arbetsramen bättre men samtidigt har vi haft större möjlighet att utforma vårt spel och system som vi har velat ha det. Slutligen har denna fokusering på lågnivåfunktioner lärt oss en massa nytt, saker som vi antagligen inte hade stött på om vi haft ett externt bibliotek som gjort saker och ting åt oss. Vårt spel är alltså lite mer av ett spelsystem/spelmotor än riktigt spel.

Sammanfattning

Projektet har varit både roligt och lärorikt, trots att vi kanske inte riktigt nådde upp till de spelmässiga mål som vi inledningsvis satt upp. Vi underskattade det arbete som krävdes för att bygga grunden för ett välfungerande spel och lyckades således inte implementera lika många spelfunktioner (så som ett melee-system) som vi hade hoppats på. Att vi dessutom valde att lägga ned tid på att utveckla spelet från 2D till 3D med platt grafik underlättade självklart inte heller saken.

Resultatet av projektet och det arbeta vi lagt ned har inneburit att vi lärt oss mer om hur man bygger upp en välfungerande infrastruktur för ett spel än hur man designar det i slutändan. Hade vi valt att använda oss av ett färdigbyggt externt bibliotek som tog hand om allt det grundläggande åt oss (t.ex. kollision osv.) hade det antagligen resulterat i ett mer komplett spel men samtidigt hade vi inte exponerats för alla dessa elementära spelmakarproblem som vi nu har stött på.

Allt som allt har knappt hälften av allt arbete som gjorts lagts ned på kollisionsdetektionen. I efterhand insåg vi hur mycket mer komplicerat allt blir av att bara lägga till en ytterligare dimension,

något vi inte riktigt hade en uppfattning om innan. Att vi dessutom inte har ett regelrätt 3D-spel utan närmare ett 2.5D-spel tror vi har ytterligare komplicerat processen eftersom denna mellanväg mellan 2D- och 3Dgrafik blir lite av ett special fall. Förutom att Java har ett fungerande bibliotek för antingen 2D eller 3D och inte för något mittemellan så finns det dessutom väldigt få resurser på internet som kan hjälpa en med problem gällande denna typ av grafik om man kört fast.

Det näststörsta problemet som vi stötte på var skapandet av en z-buffer för att kunna rita upp objekt i rätt ordning på skärmen. Eftersom detta är en funktion som begränsas till 3D-grafik fanns det väldigt få resurser som diskuterade hur man skulle kunna implementera detta i Java2D. Vi fick alltså leta oss fram till information som vagt beskrev vår situation och sedan koda delvis i blindo och hoppas på resultat. Förutom att det går lite långsamt så lyckades vi i alla fall till slut.

I slutändan har vi dock lärt oss en massa av att gå den väg vi gick med detta spelprojekt. Trots allt extra arbete och den mindre mängd spelfunktioner tillgängliga i spelet tycker vi ändå att syftet med ett projekt så som detta har uppfyllts. I tusentals rader kod har vi brottats med grundläggande spelproblem ur ett kodningsperspektiv och kommit ut visare på andra sidan med massor utav värdefull erfarenhet, erfarenhet som vi inte bara kan applicera till våra framtida spel utan även till alla andra programmeringsprojekt vi skulle sätta tänderna i.