

Proximity

En redogörelse för utvecklingsprocessen
av ett Tower Defence-spel

av Linda Evaldsson, Simon Gislén,
Hanna Römer och Johan Swanberg

Sammandrag

Syftet med denna rapport är att redogöra för utvecklingen av ett Tower Defence spel.

Teoridelen i rapporten redogör kort kring de fakta som krävs för att förstå resterande innehåll och växlar sedan in på de metoder som kan användas för att utveckla ett Tower Defence-spel.

Arbetsmetoden som beskrivs består av cykler som upprepas iterativt för att uppnå en produktiv arbetsmiljö. Metoder för att optimera hastigheten i spelet samt för att undvika designproblem beskrivs även.

I resultatet förklaras hur programmet blivit snabbare vid hastighetsoptimering samt att strukturen gjorts om och samtliga cirkelreferenser tagits bort för att undvika designproblem.

Det slutgiltiga resultatet är ett program med geometriskt tema som kan upplevas underhållande att spela men inte är särskild användarvänligt.

Slutligen diskuteras även några av de designval som genomförts under utvecklingsarbetet djupare och det konstateras att Singleton bör undvikas samt att applikationen enkelt hade kunnat fånga fler användare genom några få åtgärder för användarvänligheten. Vidare kommer diskussionen även fram till att det är högst väsentligt att börja använda program för att analysera strukturen i programmet i ett väldigt tidigt skede.

Innehållsförteckning

1. Inledning.....	1
1.1. Syfte.....	1
2. Teori: Bakgrundsinformation	2
2.1. Biblioteket libGDX	3
2.2. Partikelsystem och deras nytta	3
2.3. Designmetoden Model-View-Controller	3
3. Metod.....	5
3.1. En arbetscykel	5
3.2. Hastighetsoptimering.....	5
3.3. Vanliga designproblem och hur de kan lösas	6
3.3.1. Data som behöver vara åtkomlig överallt	6
3.3.2. Minneshantering	6
3.3.3. Upprätthålla en bra kodstruktur.....	7
4. Resultat	8
4.1. Hastighetsoptimering	8
4.1.1. Räkna ut vinkel	8
4.1.2. Visuella effekter.....	8
4.2. Strukturförändringar för att undvika cirkelreferenser	9
4.3. Det färdiga programmet.....	12
4.3.1. Strukturen i koden.....	12
4.3.2. Användarvänlighet	12
5. Diskussion.....	13
5.1. Användningen av Singleton	13
5.2. Omstrukturering av kod	13
5.3. Resultatet som helhet	13
Källförteckning	15
Bilagor	

1. Inledning

I dagens samhälle finns det en enorm tillgång till mindre spel som snabbt går att få igång och spela på datorn eller i telefonen. Tower Defence har länge varit en populär genre för att utveckla spel med framgångsspel som Plants vs. Zombies och Bloons Tower Defence i bagaget ("Tower Defense", 2015) ("Bloons Tower Defense", 2015).

Det går att utveckla ett Tower Defence-spel på flera olika sätt även om grundstrukturen är bestämd. På grund av detta finns det stora möjligheter att utveckla spelet från den givna ramen för Tower Defence men ändå vara unik.

Projektet Proximity som denna rapport byggs upp kring går ut på att utveckla ett Tower Defence-spel med en unik inriktning med mer fokus på användarinteraktion än traditionella Tower defence-spel som exempelvis Bloons Tower defense (Ninjakiwi, 2011).

1.1. Syfte

Syftet med rapporten är att redogöra för utvecklingen av Tower defence-spelet Proximity som en Java-applikation. Rapporten visar på hur en arbetsprocess för att utveckla ett strategispel kan fungera. Vidare ska rapporten även exemplifiera vilka val av modeller och struktur som bearbetas under processen. Syftet med rapporten är även att ta upp de svårigheter som kan uppstå under utvecklingen.

2. Teori: Bakgrundsinformation

På Wikipedia om Tower defense (2015) beskrivs Tower defence som en spelgenre med en särskild struktur. Genren beskrivs som ett strategispel som går ut på att hindra fiender från att nå en viss slutpunkt. Det utspelar sig på en karta och det kan exempelvis finnas en labyrinth som fienderna följer från en startpunkt på kartan till slutpunkten eller så kan spelaren ha möjlighet att bygga sin egen labyrinth med torn. Om fienderna når slutpunkten minskas livet.

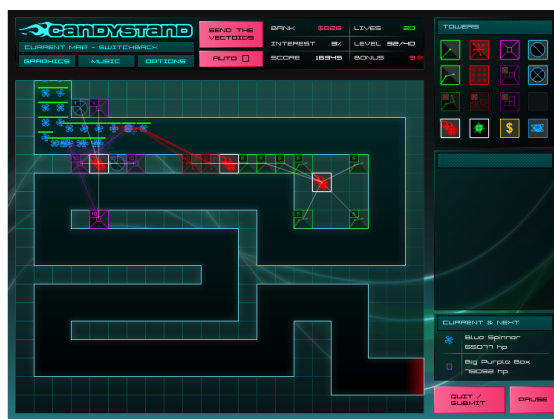
Vidare beskriver Wikipedia om Tower Defense (2015) även att spelaren för att hindra fienderna från att nå basen kan placera torn på kartan. Det förklaras att strategidelen av Tower defence är att spelare själv väljer vilka torn som ska köpas och var de ska placeras på kartan. Dessa kostar ofta någon typ av pengar eller resurser att köpa och resurserna fås normalt genom att döda fiender. De placerade tornen skjuter projektiler som träffar fienderna och skadar dem. I vissa Tower defence som exempelvis i Bloons Tower Defense (2015) skadas inte fienden utan den omvandlas i stället till en eller flera mindre fiender som i sin tur kan skjutas, till dess att den minsta fienden skapats. Denna minsta fiende skadas (och dör) när den blivit träffad.

Normalt förloras spelet om så pass många fiender nått basen att livet nått noll ("Tower Defense", 2015). Hur man vinner skiljer sig lite mellan olika spel, i Bloons Tower Defense (2015) vinner man en karta genom att klara av ett visst antal vågor av fiender. När spelet förlorats eller vunnits så har spelaren då möjligheten att starta en ny omgång.

Två populära Tower Defence-spel som finns är Bloons Tower Defense (figur 1) och Vector2d (figur 2).



Figur 1: Bloons Tower Defence



Figur 2: Vector2d

2.1. Biblioteket libGDX

LibGDX är ett plattformsoberoende spelutvecklingsramverk i Java. Det har support för både Windows/Mac/Linux samt mobila plattformar (LibGDX, 2015). Ramverket kommer med ett kraftfullt bibliotek som använder OpenGL ES 2.0 för all rendering av grafik (libGDX, 2015). Att ramverket är skrivet i OpenGL ES gör att man på ett bra sätt kan använda grafikkortet på de enheter som programmet körs på, och därmed har spel som utvecklas i denna miljö möjlighet att bli mycket kraftfulla (OpenGL.org, 2014). Att skriva koden i den gedigen OpenGL är väldigt tidskrävande och mycket mer komplicerat, så därför brukar ramverk som libGDX användas för att spara utvecklingstid (libGDX, 2015).

LibGDX innehåller även en implementation av partikelsystem som i stora drag förenklar användningen av ett sådant system i de spel som utvecklas (LibGDX, 2015).

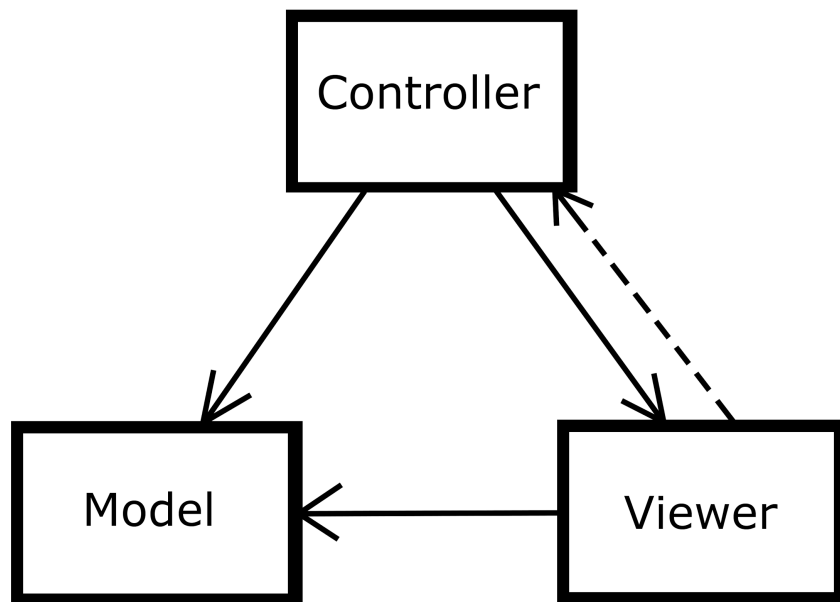
2.2. Partikelsystem och deras nytta

Ett partikelsystem inom datorsammanhang är ett system som skapar en visuell effekt (Shiffman, 2012). Det består av mängd bilder som visas, samt instruktioner som bilderna följer; vanliga regler som används är rörelsemönster, storlek, transparens, färg och rotation. Vanligtvis tilldelas bilderna i partikelsystemet reglerna med lite variation (Shiffman, 2012). Genom att låta ett system hantera en stor mängd bilder som alla följer ett visst antal regler kan man skapa en stor repertoar av olika visuella effekter. Partikelsystem kan användas för att skapa en massa olika effekter som exempelvis vatten, rök, eld eller gnistor.

En aspekt med partikelsystem jämfört med manuellt skapade visuella effekter är att de inte är repetitiva eftersom de baseras på regler. De behöver inte se likadana ut vid flera spelningar, eftersom variationen i reglerna tilldelas slumpmässigt.

2.3. Designmetoden Model-View-Controller

Model-View-Controller, eller MVC, är ett sätt att strukturera ett program. MVC har som regel en modell, vy och en kontrollerare (model, view och controller) ("Model-View-Controller", 2013). Det finns huvudsakligen i två varianter, passiv MVC och aktiv MVC (Basher, 2013). Endast passiv MVC kommer att behandlas här, då det är denna som använts i projektet. Se figur 3 för en visuell representation av passiv MVC.



Figur 3: En visuell representation av passiv MVC

Modellen är den del som hanterar informationen och innehåller data ("Model-View-Controller", 2013). Den kan inte förändra data i sig själv utan datan förändras utifrån.

Vyn har tillgång till modellen ("Model-View-Controller", 2013). Den visar upp information för användaren, och användaren kan i sin tur interagera med vyn på något sätt. Detta kan till exempel vara via mus- eller tangentklick.

Kontrolleraren har tillgång till både modellen och vyn ("Model-View-Controller", 2013). Den får information om någonting händer på vyn; om exempelvis användaren klickar någonstans på vyn får kontrolleraren veta detta. Denna indata hanteras och kontrolleraren avgör vad användaren har gjort; klickat på en knapp, skrivit något i ett fält eller liknande. Utifrån detta avgör den sedan om någon data borde ändras - knappen kan till exempel markera att information ska tas bort, att texten i fältet ska sparas eller liknande. Kontrolleraren informerar modellen om vilken data som bör ändras ("Model-View-Controller", 2013).

I passiv MVC notifierar kontrolleraren vyn när den ska uppdateras (Basher, 2013). Det kan exempelvis ske om någon förändring skett i modellen och vy behöver uppdatera vad som visas för användaren. Då hämtar vyn informationen från modellen och visar upp den istället för den information den hämtade tidigare. Detta sker även om den nya och den föregående informationen är densamma.

3. Metod

Nedan beskrivs de metoder som kan användas för att nå ett gott resultat under utvecklingen av applikationen. Framtagningen av applikationen genomförs iterativt; projektet genomförs i vissa steg som upprepas cykliskt. Varje cykel består av stegen planering, implementation och feedback.

3.1. En arbetscykel

Under planeringssteget identifieras möjliga mål för framtida utveckling av applikationen baserat på upptäckta behov samt tidigare feedback. Sedan diskuteras möjliga implementationsstrategier baserat på designmönster för att korrekt förverkliga målen. Det är viktigt att lösningarna som hittas under detta steg korrekt använder sig av MVC och har en övervägande god struktur. Planeringssteget genomförs i form av gruppmöten där relevanta mål diskuteras inom gruppen. Varje gruppmedlem tilldelas uppgifter i form av problem eller arbetsområden.

Under implementationssteget implementeras målen baserat på de strategier som togs fram under planeringen. Varje individ arbetar separat på sin uppgift och sammanfogar sin lösning i projektets huvudkodbas när arbetet med uppgiften är klart.

Efter implementationssteget kommer feedbacksteget. Här diskuteras problem som uppstått samt upptäckter och slutsatser som kan dras och hur arbetet bör fortsätta utifrån upptäckterna. Diskussionen sker under de regelbundna gruppmötena och möjliga nya mål sammanställs och används som grund för nästa planeringssteg.

3.2. Hastighetsoptimering

För att uppnå att applikationen kan köras utan att kräva onödigt mycket datorprestanda kan det vara bra att optimera de processer som körs. Tidskrävande operationer är processer eller metoder som antingen tar lång tid att genomföra eller kallas på fler gånger än vad som egentligen behövs. Det går att hitta tidskrävande metoder genom att använda den virtuella maskinen VisualVM som kommer som standard med JDK (Java development kit)(Project Kenai, 2015). Denna VM ger en grafisk översikt över vilka metoder som används mest och hur mycket tid de tar.

Testa tidskrävande operationer går att göra genom att låta systemet anteckna den nuvarande tiden och därefter köra metoden ett antal gånger. Antalet gånger måste avgöras på ett sätt anpassat till metoden som testas, så att resultatet blir statistiskt trovärdigt. När metoden körts det bestämda antalet gånger så antecknas den nya tiden av systemet. Tiden det tog mellan olika implementationer kan sedan jämföras.

Det är viktigt att testa metoderna flera gånger och även testa dem i realistiska scenarion, det vill säga i det sammanhang metoden normalt kallas. Metoder är inte konsekventa i hur lång tid de tar att genomföra, utan tiden beror på vilket sammanhang de kallas i (se bilaga 4). När metoden testats kan den skrivas om till en metod som tros vara mer effektiv och testas igen för att få fram om den effektivare metoden faktiskt är effektivare i praktiken.

3.3. Vanliga designproblem och hur de kan lösas

Under designprocessen finns möjligheten att ett flertal problem uppstår. För att undvika att denna typ av problem blir för stora är det viktigt att kontinuerligt analysera koden med program som exempelvis FindBugs och STAN. FindBugs är ett program för att hitta kod som kan vara dålig i programmet, medan STAN hjälper till med kodstrukturen genom att analysera koden och visa upp en grafisk bild av beroenden mellan klasser (www.findbugs.sourceforge.net) (www.stan4j.com).

Nedan beskrivs en del vanliga designproblem och hur de går att lösa.

3.3.1. Data som behöver vara åtkomlig överallt

När man skriver kod är det viktigt att avgöra vad som behöver vara tillgänglig överallt (statiskt och publikt). Generellt gäller att klassers metoder och variabler i största möjliga mån inte bör vara statiska och publika då detta minskar inkapslingen, som är bra för att minska klassers beroenden av varandra.

Om det finns en klass som kommer användas av i princip alla övriga klasser så kan det ändå vara ett alternativ att låta den vara tillgänglig överallt. En möjlighet är då att skapa klassen som en Singleton, vilket är en typ av designmönster som försäkrar att det endast finns en enda instans av klassen vid varje tillfälle (Geary, 2003).

Trots vissa fördelar med singleton bör man alltid tänka två gånger innan man skapar en sådan klass. Enligt Dam (2010) bör Singleton-designmönstret undvikas så långt det bara går, då det är ett mönster som kan verka väldigt trevligt till en början, men ställa till problem när förändringar behöver genomföras i koden.

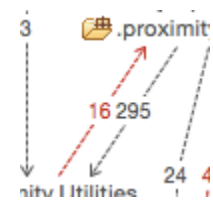
3.3.2. Minneshantering

Ett problem som ofta uppstår under utveckling av projekt är problem med minnesläckor. I datavetenskap är en minnesläcka en typ av resursläcka som uppstår när ett datorprogram allokerar minne utan att fria upp det när det har använts klart (Tech Terms Computer Dictionary, 2008). Att minnet inte frigörs kan innebära att programmet till slut tar upp mer minne än vad datorn klarar av, vilket då innebär att programmet kraschar. Om det uppstår ett problem med minnesläckor kan man lösa det genom att utveckla ett så kallat cachesystem. Rör

det exempelvis bilder så kan programmet i stället för att hela tiden skapa en ny bild leta efter bilden i minnet och skicka vidare en referens till bilden. Om bilden inte existerar, skapas bilden och läggs till i minnet. Under programmets avstängning frigörs allt cacheminne.

3.3.3. Upprätthålla en bra kodstruktur

För att upprätthålla en bra kodstruktur bör programkoden i projektet regelbundet analyseras med STAN. En sak som går att se i programmet är cirkelreferenser som markeras med en röd pil i programmet (se figur 4). Cirkelreferenser är när klasser refererar till varandra i en cirkel och bör i största möjliga mån undvikas på grund av att det försvårar för förändringar, testning och vidareutveckling av ett program (Aaronaught, 2010). Som exempel är en cirkelreferens när klass A har en referens till klass B som i sin tur har en referens tillbaka till A (Koirala, 2013).



Figur 4: Röd pil i STAN som indikerar cirkelreferens

Upptäcks en cirkelreferens i STAN kan den analyseras genom att man klickar på den röda pilen som markerar cirkelreferensen. Den bör sedan tas bort genom ändringar i koden.

4. Resultat

Här följer resultatet av ett urval av de förändringar som genomförts under applikationens utveckling för att åtgärda flera problematiska scenarion. Därefter följer en redovisning av den slutgiltiga applikationen.

4.1. Hastighetsoptimering

Applikationen har som mål att under normala omständigheter generera 60 bilduppdateringar per sekund. Detta betyder att varje bilduppdatering maximalt får ta cirka 16 millisekunder att köra. På grund av detta så måste varje metod i applikationen vara optimerad för hastighet.

4.1.1. Räkna ut vinkel

Under projektet uppstod problemet att metoden för att räkna ut vinkeln mellan en punkt och en annan både var långsam och användes ofta. Därför optimerades denna metod.

Metoden optimerades genom att förändras från att genomföra två långsamma matematiska uträkningar till att i stället hämta resultat från en färdiguträknad tabell (eng: lookup table), då detta är snabbare (se bilaga 1). Eftersom denna tabell inte innehåller alla värden kan den endast ge ett approximativt svar på uträkningen som ska göras (se bilaga 2). På grund av detta så lämpar sig metoden inte för situationer som kräver precision, men eftersom hastighet prioriteras över precision i detta projekt är det lämpligt att den används. Felmarginalen för uträkningen är 0,46 grader (se bilaga 3) vilket är så pass lite att det inte har betydelse för applikationen.

Resultatet av optimeringen blev att metoden blev cirka 60 gånger snabbare (se bilaga 1).

4.1.2. Visuella effekter

Den visuella representationen av projektet, det vill säga vad användaren observerar och interagerar med, påverkar mycket hur användaren upplever applikationen. För att göra applikationen mer intressant implementerades ett flertal partikeleffekter som spelas upp till följd av vissa händelser, som exempelvis explosioner, magiska attacker och då spelaren tar skada (se figur 5).



Figur 5: Exempel på en partikeleffekt i spelet Proximity

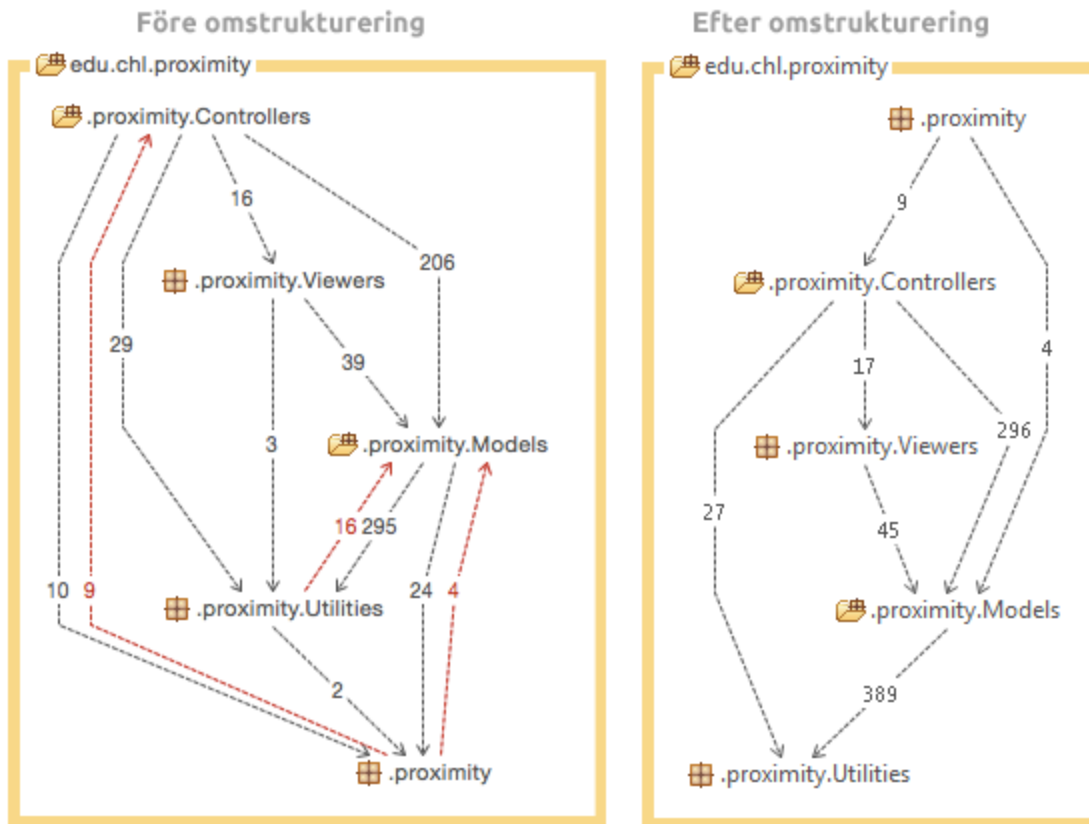
Dessa visuella effekter skapas med hjälp av libGDX bibliotek för partikeleffekter. Att skapa effekter via biblioteket anses bra utifrån ett prestandaperspektiv då biblioteket implementerar flera partikel-regler som transparens, skalning och rotation med hjälp av OpenGL, som kan köra effekterna effektivt via grafikkortet. (libGDX, 2015)

Vidare hanteras partikeleffekterna i en pool från biblioteket libGDX då detta är en effektivare lösning än att skapa nya effekter hela tiden. Poolen implementerar designmönstret Pool, som är ett designmönster som förhindrar att objekt tas bort av Javas skräpsamlare. I stället för att tas bort så återanvänds objekten (Nystrom, 2009-2014).

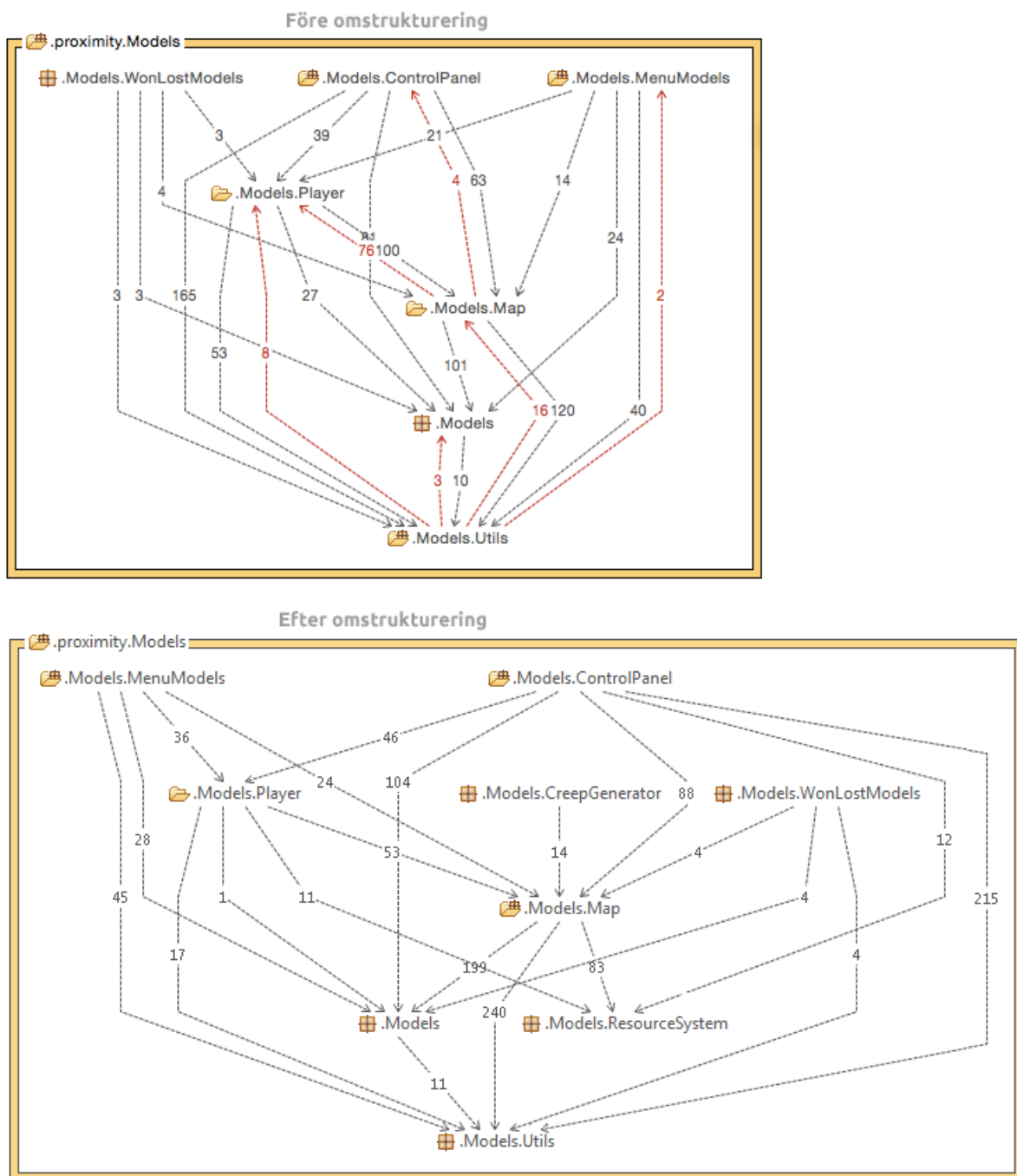
Generellt är det inte bra att använda en pool eftersom Javas skräphanterare är väldigt effektiv. Att kontrollera om objekt redan finns är en relativt dyr operation tidsmässigt och denna operation utförs ofta i en pool (Goetz, 2004). Orsaken till att en pool ändå används är för att skapandet av partikeleffekter är en prestandakrävande operation, då den läser bildinformation från hårddisken. Att hela tiden hämta information från hårddisken är mycket mindre effektivt än att läsa från RAM-minnet (Goetz, 2004). En pool av partikeleffekter spenderar relativt mycket RAM minne, men kräver relativt lite prestanda från processorn.

4.2. Strukturförändringar för att undvika cirkelreferenser

Under projektets gång har applikationens kod analyserats i programmet STAN för att upptäcka cirkelreferenser och andra strukturproblem. Cirkelreferenser i STAN markeras med röda pilar. Koden har därefter omstrukturerats eller ändrats på andra sätt för att minska problemen. Strukturförändringarna har resulterat i att strukturen förbättrats avsevärt (Se figur 6-7).



Figur 6: Före- och efterbild från STAN över strukturen av projektet



Figur 7: Före- och efterbild från STAN över strukturen av projektets paket Models

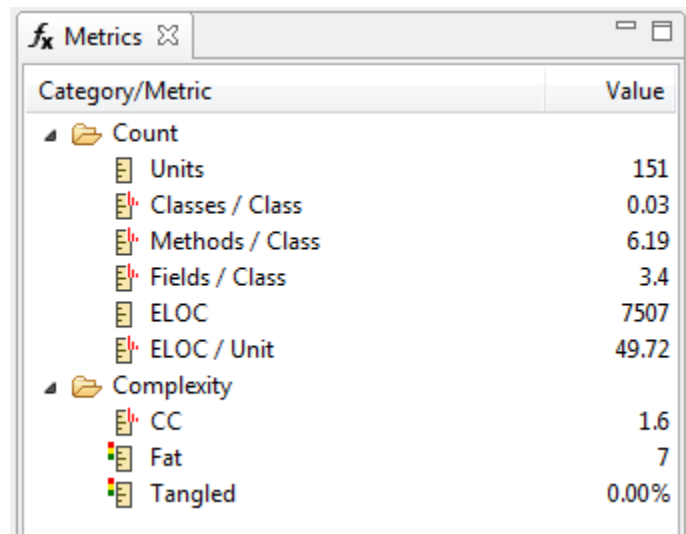
4.3. Det färdiga programmet

Utvecklingen av applikationen har resulterat i ett fungerande och balanserat spel som kan upplevas underhållande att spela (se bilaga 5). I spelet finns möjlighet att placera torn för att försvara sin bas och använda magiska attacker på de fiender som försöker nå basen.

4.3.1. Strukturen i koden

Strukturen i programmet är i mångt och mycket uppbyggd för att undvika de problem och exempel på dålig kodstruktur som beskrivits under metodavsnittet. Generellt har publika metoder undvikits för att upprätthålla inkapslingen, i programmet finns endast några få klasser som är tillgängliga överallt. Klassen för kartan (Map) skrevs först enligt designmönstret Singleton men har senare skrivits om till en vanlig klass i stället. Även klassen för spelaren (Player) skrevs som en Singleton från början, men skrevs senare om och nås i stället via en annan Singleton som hanterar speldata (GameData).

En analys via STAN anger att strukturen i programmet ser bra ut, på grund av minskandet av cirkelreferenser ligger mängden cirkelreferenser (Tangled) på 0.00% (se figur 8).



The screenshot shows a window titled 'Metrics' with a table of code quality metrics. The table has two columns: 'Category/Metric' and 'Value'. The metrics are grouped into two main categories: 'Count' and 'Complexity'.

Category/Metric	Value
Count	
Units	151
Classes / Class	0.03
Methods / Class	6.19
Fields / Class	3.4
ELOC	7507
ELOC / Unit	49.72
Complexity	
CC	1.6
Fat	7
Tangled	0.00%

Figur 8: Statistik från STAN kring strukturen i Proximity

4.3.2. Användarvänlighet

Under utvecklingen av applikationen har användarvänligheten i programmet funnits med under diskussioner men prioriterats bort till fördel för andra funktioner. Av den anledningen är dessa funktioner inte helt implementerade vilket innebär att applikationen kan upplevas som förvirrande vid första körningen.

5. Diskussion

Utifrån resultatet har några punkter kommit fram som bör belysas ytterligare. Här redovisas en djupare diskussion runt dessa punkter.

5.1. Användningen av Singleton

En problematik som uppstod i samband med utvecklingen var att det fanns mycket information som behövde nås av många klasser, vilket var orsaken till att både klassen för kartan (Map) och klassen för spelaren (Player) från början skrevs enligt designmönstret Singleton. Mönstret ansågs lämpligt till en början då det inte behöver finnas mer än en aktiv karta och spelare i programmet på en och samma gång.

Orsaken till att kartan skrevs om så att den inte längre är en Singleton är att detta förenklar för testningen då alla tester blir mer självständiga och oberoende av varandra eftersom de inte behöver nå samma instans av en Singleton. Vidare behöver man inte testa många objekt för att exempelvis bara testa ett torn (Tower).

Att spelaren inte skrevs om utan fortfarande nås via en Singleton innebär att det bara kan finnas en aktiv spelare åt gången. Detta är olämpligt eftersom det gör det svårare att testa programmet samt försvårar om man vill utveckla spelet vidare och exempelvis introducera att det kan vara mer än en spelare åt gången. En bättre lösning hade varit att skriva om även klassen för Spelare så att den inte längre nås via en Singleton-lösning.

5.2. Omstrukturering av kod

Under projektets gång användes inte STAN förrän en hel del av kodimplementeringen redan var färdig. Av den anledningen var vissa problem väldigt fast förankrade när STAN väl användes för att analysera koden. Det gjorde det till ett stort och tidskrävande arbete att omstrukturera denna kod vilket tog upp mycket onödig tid som bättre hade kunnat användas till annat.

Ett bättre förhållningssätt vid utveckling är att redan från början använda STAN för att analysera koden. Ju tidigare ett problem upptäcks, desto enklare är det att åtgärda då det inte är lika fast förankrat tidigt i processen.

5.3. Resultatet som helhet

Det är tydligt att användarvänligheten i programmet inte är på en hög nivå. Någonting som saknas är en tydlig användarguide när programmet skapas som varsamt lotsar in användaren i hur programmet används. Det hade kunnat användas informationsrutor som dyker upp vid vissa händelser, som exempelvis en informationsruta som informerar om att man kan placera torn första gången man startar spelet och liknande.

Om man bortser från den dåliga användarvänligheten så är det färdiga programmet ett bra program. Det har en bra struktur, är snabbt att köra och hanterar minnet på ett effektivt sätt. Projektet har därför uppnått sitt mål att skapa ett Tower Defence-spel, och det med ett lyckat resultat. Spelet kan anses vara unikt och annorlunda om det jämförs med traditionella Tower Defence-spel, eftersom magiska attacker som finns i spelet inte är något som generellt erbjuds inom Tower Defence.

Källförteckning

Aaronaut. (2010). *What's wrong with circular references?*

Hämtad från <http://programmers.stackexchange.com/questions/11856/whats-wrong-with-circular-references>

Basher, Khademul. (2013). *MVC Patterns (Active and Passive Model) and its implementation using ASP.NET Web forms.*

Hämtad från <http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its>

Bloons Tower Defense. (2015). *I Wikipedia.*

Hämtad 2015-05-06 från http://en.wikipedia.org/wiki/Bloons_Tower_defense

Dam, J. (2010). *Singletons: Solving problems you didn't know you never had since 1995.*

Hämtad från <http://jalf.dk/blog/2010/03/singletons-solving-problems-you-didnt-know-you-never-had-since-1995/>

Geary, D. (2003). *Simply Singleton – Navigate the deceptively simple Singleton pattern.*

Hämtad från <http://www.javaworld.com/article/2073352/core-java/simply-singleton.html>

Goetz, B. (2004). *Java theory and practice: Garbage collection and performance.*

Hämtad från <http://www.ibm.com/developerworks/library/j-jtp01274/index.html>

Koirala, S. (2013). *What is Circular dependency and how do we resolve it?*

<http://www.codeproject.com/Articles/616344/What-is-Circular-dependency-and-how-do-we-resolve>

libGDX. (2015). *Goals and Features.*

Hämtad från <http://libGDX.badlogicgames.com/features.html>

Model-View-Controller. (2013). *I iOS Developer Library.*

Hämtad 2015-05-06 från

<https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>

Ninjakiwi. (2011). *Bloons Tower Defense 5*.

Hämtad från <http://ninjakiwi.com/Games/Tower-Defense/Play/Bloons-Tower-Defense-5.html#.VUnCnUKZang>

Nystrom R. (2009-2014). Object Pool – Game Programming Patterns / Optimization Patterns.

Hämtad från <http://gameprogrammingpatterns.com/object-pool.html>

Obviam.net. (2012). *Building Games Using the MVC Pattern - Tutorial and Introduction*.

Hämtad från <http://obviam.net/index.php/the-mvc-pattern-tutorial-building-games/>

OpenGL. (2014). *What is OpenGL?*

Hämtad från https://www.opengl.org/wiki/FAQ#What_is_OpenGL.3F

Project Kenai. (2015). *Download VisualVM*.

Hämtad från <http://visualvm.java.net/download.html>

Shiffman, D. (2012). *The Nature of Code*.

Hämtad från <http://natureofcode.com/>

Tech Terms Computer Dictionary (2008). *Memory Leak*.

Hämtad från <http://techterms.com/definition/memoryleak>

Tower Defense. (2015). *I Wikipedia*.

Hämtad 2015-05-06 från http://en.wikipedia.org/wiki/Tower_defense

Bilagor

Nedan följer bilagorna som hör till rapporten.

Bilaga 1 - Test av skillnad i hastighet mellan två olika implementationer av `getAngle`:

av Johan Swanberg, 2015-05-18

Kod som utför testet:

```
Vector2 firstTestVector = new Vector2(123,123);
Vector2 secondTestVector = new Vector2(623,153);
int testamounts = 10000000;
double currentTime = System.currentTimeMillis();
for (int i=0; i<testamounts; i++){
    spdt1(firstTestVector,secondTestVector);
}
double timeElapsed = System.currentTimeMillis()-currentTime;
System.out.println("Time taken for old first test= " + timeElapsed/1000);
double currentTime2 = System.currentTimeMillis();
for (int i=0; i<testamounts; i++){
    spdt2(firstTestVector, secondTestVector);
}
double timeElapsed2 = System.currentTimeMillis()-currentTime2;
System.out.println("Time taken for new first test= " + timeElapsed2/10
private static MathUtils utils = new MathUtils();

private double spdt2(Vector2 firstPoint, Vector2 secondPoint){
    return
utils.radiansToDegrees*(utils.atan2(secondPoint.y-firstPoint.y,secondPoint.x-firstPoint.x));
}

private double spdt1(Vector2 firstPoint, Vector2 secondPoint){
    if (firstPoint != null && secondPoint != null) { //make sure there is a real vector

        Vector2 vector = new Vector2(secondPoint.x - firstPoint.x, secondPoint.y -
firstPoint.y); //to get a vector, subtract point a from point b
        double hypotenuse = Math.sqrt(vector.x * vector.x + vector.y * vector.y);
        if (hypotenuse == 0) {
            return 0;
        }
        double angle = Math.acos(vector.x / hypotenuse);
        double angleDegrees = Math.toDegrees(angle);
        if (vector.y > 0) {return angleDegrees;}
        return -1 * angleDegrees;}return 0;
    }
}
```

Testresultat:

Programmet skriver ut:

Time taken for old first test= 4.371

Time taken for new first test= 0.071

Enligt testet tar det alltså ca. 4,4 sekunder att köra metoden `getAngle` 10 000 000 gånger med den gamla implementationen. Kör man den nya metoden lika många gånger tar det i stället ca. 0,071 sekunder. Testen gjordes flera gånger, och testades i olika ordning. Samtliga test gav liknande resultat.

Den nya koden är alltså 61 gånger snabbare enligt test.

Bilaga 2 - Precisionstest av libGDX och Java.Math atan2-metoder

av Johan Swanberg, 2015-05-18

Kod som utför testet:

```
//test 1
MathUtils util = new MathUtils();
System.out.println("Java aTan: " + Math.atan2(0.1234, 0.4321));
System.out.println("libGDX aTan:" + util.atan2(0.1234f,0.4321f));
//test 2
System.out.println("Java aTan: " + Math.atan2(0.12238f, 1));
System.out.println("libGDX aTan:" + util.atan2(0.12238f,1));
```

Testresultat:

Följande resultat på beräkningarna skrivs ut:

```
Java aTan:      0.2781773909329711
libGDX aTan:    0.2762185
Java aTan:      0.12177447913107388
libGDX aTan:    0.11756557
```

Ovanstående är alltså resultatet i radianer av två tester med olika siffror.

Detta resultat jämförs med föruträknade resultat med mycket större precision.

Det korrekta svaret för test 1 är:

0.278177390932971129471988408870075544032906382224826819064847... (Beräkning av atan(0.1234/0.4321), 2015)

Det korrekta svaret för test 2 är:

0.121774475847164799167750440264026133632660999227615843353637... (Beräkning av arctan (0.12238), 2015)

Jämför man det korrekta svaret för testerna med resultatet som ges från Javas aTan och libGDX aTan får man följande precision:

Test 1:

```
Java's felaktighet: 0.000000000000000029472...
libGDX felaktighet: 0.00195889093297112947...
```

Test 2:

```
Java's felaktighet: 0.0000000003.2839090808...
libGDX felaktighet: 0.00420890584716479916...
```

Slutsatsen som kan dras utifrån detta är att libGDX atan2 metod har 2 decimalers precision jämfört med Javas atan2-metod som har 8 decimalers precision. Ovanstående tester är dock begränsade i antal och skulle bli mer trovärdiga om ett större antal tester kördes.

Referenslista

Wolfram Alpha LLC (2015) *Beräkning av atan(0.1234/0.4321)*

Hämtad från <http://www.wolframalpha.com/input/?i=atan%280.1234%2F0.4321%29>

Wolfram Alpha LLC (2015) *Beräkning av arctan(0.12238)*

Hämtad från <http://www.wolframalpha.com/input/?i=arctan+%280.12238%29>

Bilaga 3 - Jämförelse mellan Java.Math och libGDX atan2-metoder

av Johan Swanberg, 2015-05-18

För att kontrollera hur stor skillnad libGDX atan2 metod och Java.Math atan2 metod kan returnera, testades metoderna med nedanstående program. Programmet genomför 10 000 000 körningar av metoden atan2 med parametrarna tan1 och tan2. Den första parametern tan1 börjar på värdet "0f" och ökar med 0.0000001 för varje körning. Den andra parametern tan2 är konstant 1.

Programmet skriver till slut ut hur stor den totala skillnaden är på de 1000 körningarna, den största förändringen mellan beräkningarna samt den genomsnittliga förändringen mellan beräkningarna. Resultatet är i radianer.

```
MathUtils util = new MathUtils();
double biggestError = 0;
float tan1 = 0f;
float tan2 = 1;
double total = 0;
System.out.println("Start-testing number: aTan(" + tan1 + ")");
int runs = 10000000;
for (int i = 0; i < runs; i++) {
    double javaATan = Math.atan2(tan1, tan2);
    double libGDXATan = util.atan2(tan1, tan2);
    double error = Math.abs(javaATan - libGDXATan);
    if (error > biggestError) { biggestError = error; }
    tan1 += 0.0000001;
    total += error;
}
System.out.println("End-testing number: aTan(" + tan1 + ")");
System.out.println("Biggest difference = " + biggestError);
System.out.println("Average difference= " + total / runs);
```

Utskriften blev:

```
Start-testing number: aTan(0.0)
End-testing number: aTan(1.0647675)
Biggest difference = 0.00787384740718035
Average difference= 0.003089513161458096
```

Programmet kördes flera gånger med olika ökningsvärde på tan1 men resultatet på "Biggest difference" (största skillnaden) blev alltid 0.007873848338445186 radianer vilket motsvarar ca. 0.451138° (grader). Den genomsnittliga skillnaden efter en av körningarna landade på 0.003089513161458096 radianer motsvarar ca 0.177016° (grader).

libGDX metod atan2-metod är baserad på en tabell av värden och eftersom tabellen är skapad med ett konstant mellanrum är det troligt att 0.007873848338445186 är mitt mellan två tabellresultat, och att det därför aldrig blir större skillnad. (libGDX, 2015)

Slutsats som kan dras utifrån detta är att den största skillnaden som uppmättes var cirka 0.45 grader, den genomsnittliga skillnaden är ca 0.18 grader.

Referenslista:

Class MathUtils (2015) i *libGDX API*

Hämtad 2015-05-21 från

<http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/math/MathUtils.html>

Bilaga 4 - Jämförelse av metodhastighet i for-loop

av Johan Swanberg, 2015-05-25

Nedan kod skrevs för att testa for-loops påverkan av hastighet för att utföra en metod. Testet gick ut på att köra samma metod några få antal gånger, och anteckna hur lång tid det tog att utföra 10 test, och sedan köra signifikant fler gånger och anteckna hastigheten.

Testkod:

```
ProximityVector v1 = new ProximityVector(100,100);
ProximityVector v2 = new ProximityVector(321,123);

double time = System.currentTimeMillis();
for (int i = 0; i<1000; i++){
    PointCalculations.distanceBetweenNoSqrt(v1,v2);
}
double time2 = System.currentTimeMillis();
System.out.println("Time for 10 calculations: " + (time2-time)/100);

double time3 = System.currentTimeMillis();
for (int i = 0; i<100000; i++){
    PointCalculations.distanceBetweenNoSqrt(v1,v2);
}
double time4 = System.currentTimeMillis();
System.out.println("Time for 10 calculations: " + (time4-time3)/10000);
```

Testkoden kördes upprepade gånger på en Lenovo Yoga 2 Pro med i7 processor, resultatet förblev konsekvent.

Resultat, utskrift:

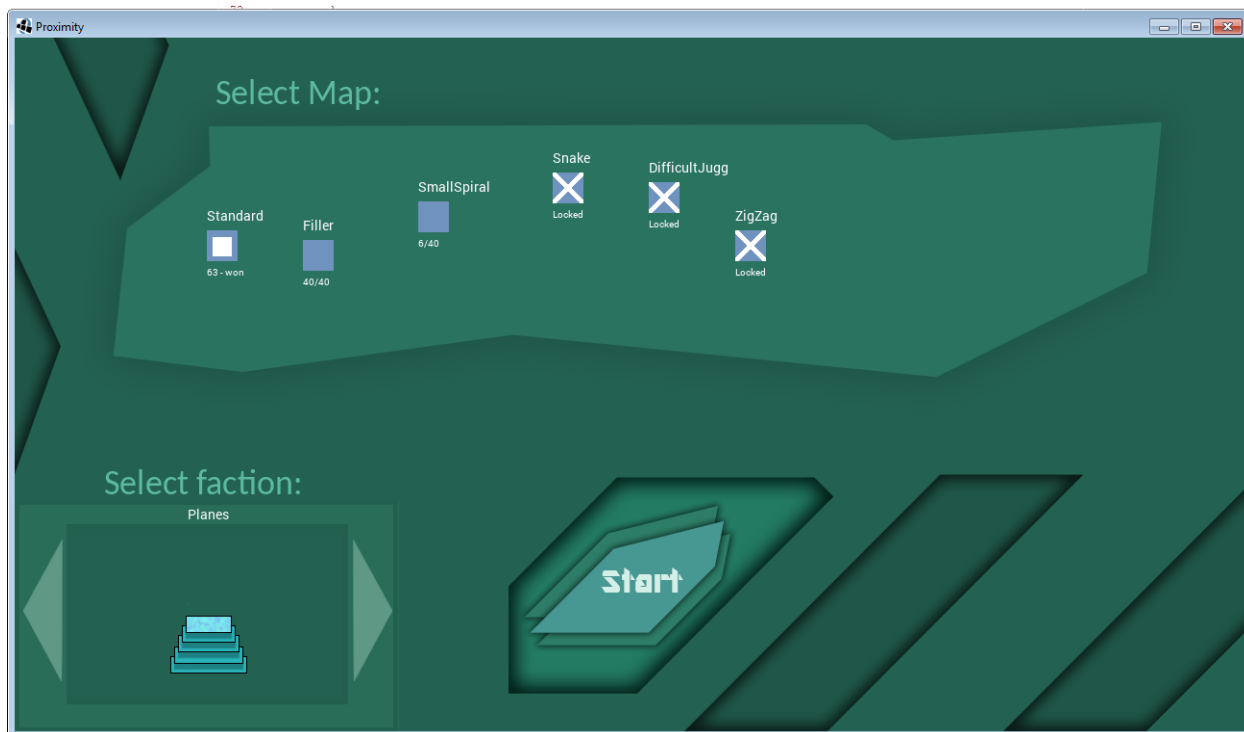
Time for 10 calculations: 0.01

Time for 10 calculations: 0.0004

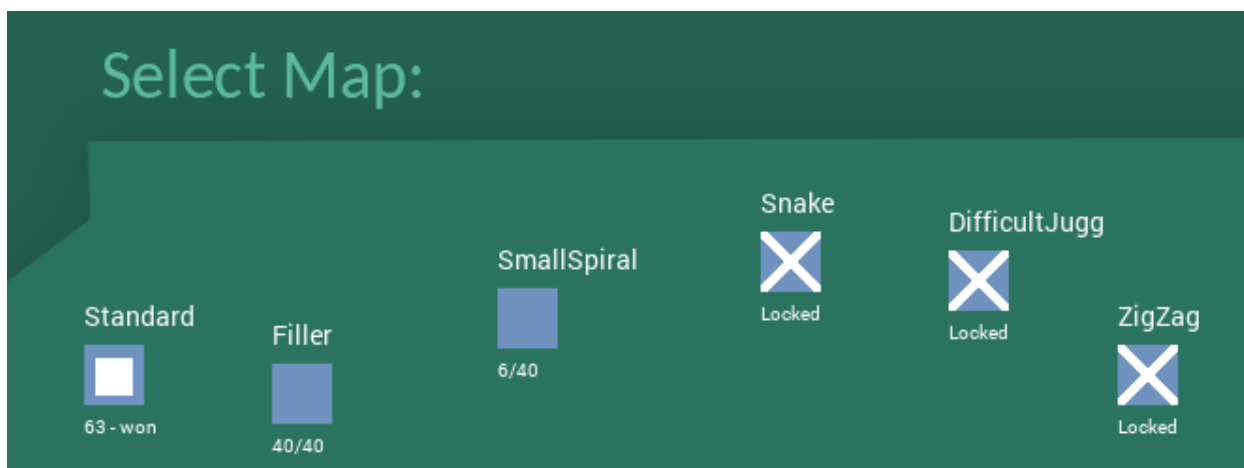
Slutsatsen är att tiden att utföra en metod minskar signifikant då man kör metoden upprepade gånger.

Bilaga 5 - Bilder av den färdiga versionen av applikationen

av Linda Evaldsson, 2015-05-26



Figur 1: Startmenyn i Proximity



Figur 2: Kartväljaren på startmenyn



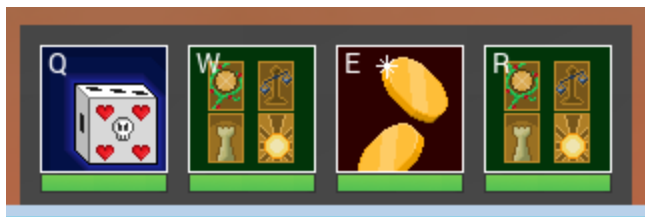
Figur 3: Tillhörighetsväljaren (Faction chooser) på meny



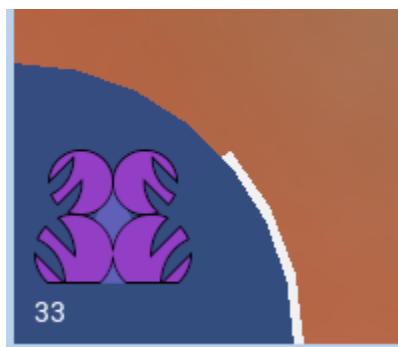
Figur 4: Bild av en karta (Map) i Proximity



Figur 5: Kontrollpanel som visar spelarens liv och resurser. Kontrollpanelen erbjuder också möjligheten att köpa torn samt uppgradera eller sälja befintliga torn. Vidare finns det längst ner även funktioner för att kontrollera hastigheten i spelet.



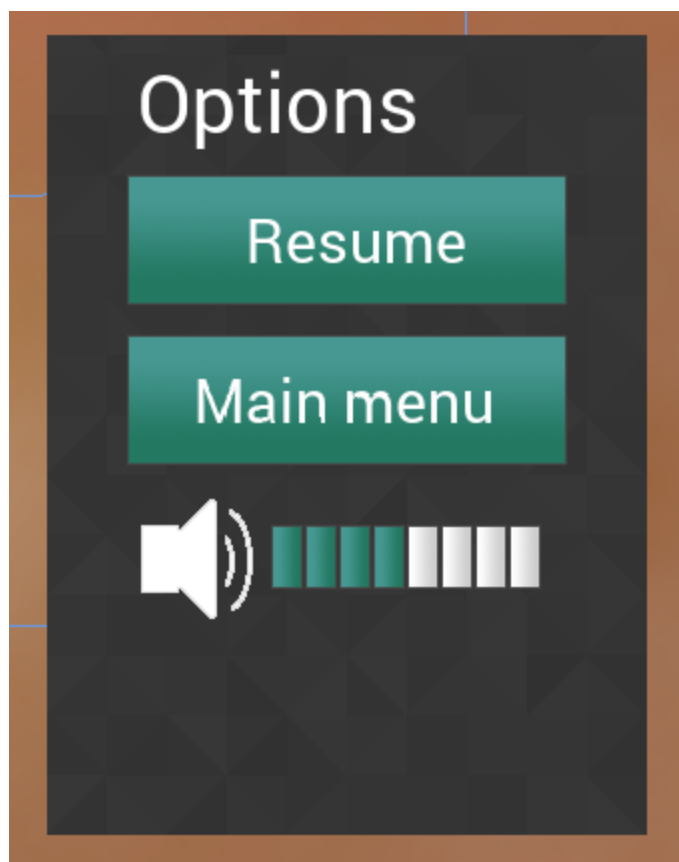
Figur 6: Panel med magiska förmågor som kan användas på kartan



Figur 7: Panel som visar spelarens nivå (level) och en bild på den tillhörighet (faction) som valts.



Figur 9: Våg-indicator som visar vilken våg (wave) som har besegrats.



Figur 10: Meny inuti spelet