Projektbeskrivning

Tower Defence

2019-03-08

Projektmedlemmar:
Emil Wiman emiwi425@student.liu.se
Hannes Westander hanew828@student.liu.se

Handledare:
Oscar Lundin osklu414@student.liu.se

Table of Contents

Planering

1. Introduktion till projektet

I det här projekt kommer vi att utveckla ett spel av typen Tower Defence (se bild 1). Detta är en känd spelgenre som går ut på att man placerar ut ”torn” på en spelplan och där tornen sedan ska eliminera fiender som rör sig enligt en given bana på spelplanen. Fienderna är av olika karaktär och om de når slutet av banan tar spelaren skada. Tar man för mycket skada förlorar man spelet. Under spelets gång får spelaren chans att köpa fler och mer avancerade torn för att kunna hantera svårare fiender.

Bild 1: Bloons Tower Defence, ett ikoniskt spel inom genren.

Spelet grundas på olika rundor där de senare rundorna är svårare. Efter en runda kan spelaren köpa till nya torn utifrån de pengar man tjänat på att eliminera inkommande fiender. Om spelaren klarar av alla rundor vinner man spelet.

2. Ytterligare bakgrundsinformation

Tower Defence spel är en subgenre av Strategispelgenren. Det huvudsakliga målet är att försvara spelarens territorium eller tillgångar genom att eliminera inkommande fiender som rör sig enligt givna banor. Genom att placera ut defensiva strukturer av olika slag stoppas fienden innan den når målet. Fiender kommer ofta i vågor med varierande mängd och typ, vilket bidrar till ett varierat och dynamiskt spel där planering och taktik är avgörande faktorer för att vinna.

Vi hänvisar till ”Tower Defence” wikipediasida för vidare information kring spelets uppbyggnad:

https://en.wikipedia.org/wiki/Tower_defense

3. Milstolpar

#	Beskrivning
1	Utveckla en spelplan som visualiseras i konsolen. Spelplanen ska bestå av olika block.
2	Implementera olika ”block”, sådana som vi kan placera ”torn” på och sådana som fiender senare ska kunna gå på.

3	Utveckla en ”fiende” som kan placeras ut på planen på en hårdkodad plats. Skapa en form av ”fabrik” som kan skapa instanser av denna fiende.
4	Utveckla ett fönster där vi kan skriva över vår spelplan och se våra olika block. Utveckla även Level editor som låter oss manipulera planen på ett smidigt sätt genom fönstret.
5	Implementera en tick-metod som uppdaterar spelet och ritar om spelplanen.
6	Implementera så att vår fiende kan röra sig enligt en given väg på spelplanen. Gör även så att när fienden när målet tar vi skada. Här implementerar vi även vårat eget HP.
7	Implementera ett grundläggande torn-klass som de andra tornen ärver. Varje torn bör ha en attackradie, utseende, en position etc
8	Implementera en ”bullet”-klass sedan kan användas för att skjuta fiender.
9	Gör så att tornet kan attackera en fiende när fienden är inom dess attackradie. Tornet ska då avfyra ett skott och vi vill detektera en kollision.
10	Implementera en ”cooldown” så att vi ej attackerar på varje tick i spelet.
11	Utöka så vi kan spela mer än en runda i spelet. Exempelvis genom flaggor.
12	Utöka antalet olika fiender. Ex. Mer liv, rör sig snabbare, annat utseende, skapar fler fiender när den själv dör etc.
13	Utöka antalet olika torn. Gör mer skada, lägre cooldown, större attackradie etc.
14	Lägg till pausfunktion.
15	Lägg till startmeny.
16	Utöka antalet banor. 3 st olika exempelvis. (Lätt, Medium, Svår).
17	Implementera även liv och pengar som ”globala” konstanter.

4. Övriga implementationsförberedelser

Board

Board bör bli en egen klass som ritar upp spelplanen enligt en 2D-array. Board bör kunna rita om sig själv enligt en ”tick-metod” som uppdateras via varje tick. Board bör visas av en annan klass som hantera själva GUI:et. Board bör även kunna berätta för andra klasser var saker och ting befinner sig på skärmen.

Fiende

Vår fiende skulle rimligtvis kunna ärva vissa egenskaper från en superklass. När man sedan vill utöka antalet fiender låter man dessa ärva de egenskaper och metoder de delar. Sedan låter man varje skild fiende ha sina egna specialiteter i sin egen klass.

Torn

Torn bör ha en liknande implementation som fiender. Vi låter varje torn ärva från en superklass och sedan får varje torntyp ha koll på sina egna egenheter.

Bullet

Bör bli en en egen klass i implementation. Vi låter en bullet skapas när våra torn har detekterat en fiende och sedan låter vi tornet skjuta i den riktningen som fienden är. Även här kanske vi behöver skapa en superklass och sedan låta utveckla olika typer av bullets med olika egenskaper till respektive torn.

5. Utveckling och samarbete

Vi har tänkt oss att jobba tillsammans i början för att hjälpas åt med själva grunderna i spelet och den grundläggande mekaniken och implementationen. Senare kommer vi att jobba mer enskilt där det passar, exempelvis utveckla olika fiender med mera. Vi tänker oss att vi kommer försöka boka in tider med varandra under vardagar mellan 8–17 då vi jobbar. Vår ambitionsnivå är framförallt att klara av projektet så vi blir godkända på den tid som projektet bör ta. Finns mer tid att tillgå när projektet är godkänt kan vi mycket väl lägga den tiden men annars låter vi det vara.

Slutinlämning

6. Implementationsbeskrivning

Härvid redogör vi för de olika implementationsdetaljer som kan vara av värde för vidareutveckling eller underhåll av vårt program.

6.1. Milstolpar

Varje milstolpe (se Kap 3) redogörs vid korresponderade siffra i listan här under.

1. Milstolpen här helt genomförd. Den är dock borttagen ur koden då den enbart fyllde ett syfte i den tidiga utvecklingen av spelet.
2. Milstolpen här helt genomförd.
3. Milstolpen är delvis genomförd, en fiende har utvecklats men vi valde att inte implementera en fabrik för fienden utan istället en abstrakt klass som ger varje fiende mer generella funktioner.
4. Milstolpen är delvis genomförd. Vi utvecklade ett fönster där vi kunde se vår spelplan och fiende men valde att inte utveckla en ”*Level editor*” så tidigt i spelet då det inte var nödvändigt för att få spelet att fungera.
5. Helt genomförd.
6. Helt genomförd.
7. Helt genomförd. Vi implementerade en abstrakt tornklass som de andra tornen ärver.
8. Helt genomförd.
9. Helt genomförd.
10. Helt genomförd.
11. Helt genomförd.
12. Ej genomförd.
13. Ej genomförd.
14. Ej genomförd.
15. Ej genomförd.
16. Delvis genomförd. Vi utvecklade en bana till för att ha en mer ”utmanande” bana för att kunna testa funktionaliteten hos våra olika mobs (fiender) och torn.
17. Delvis genomförd. Vi har implementerat vårt eget liv och pengar men ej som globala konstanter utan som fält i klassen Board.

6.2. Dokumentation för programstruktur, med UML-diagram

Härvid redogör vi för de olika övergripande strukturer och metoder som kan vara av värde för vidareutveckling och underhåll. Vi tittar även närmare på hur de olika klasserna hänger ihop.

Programstrukturer med UML-diagram

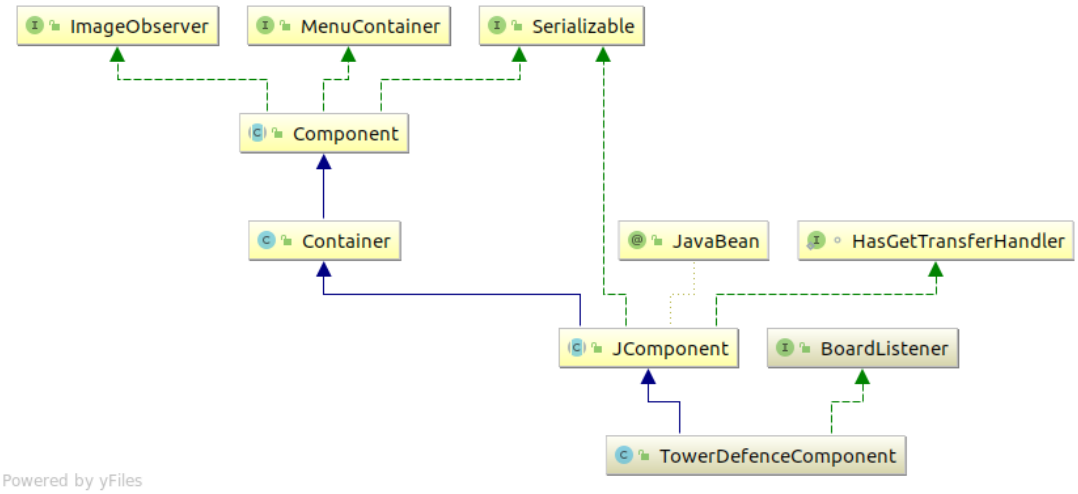

Bild 2. UML-diagram över hur TowerDefenceComponent interagerar med gränsnittet BoardListener.

BoardListener är ett Interface som vi använt oss av för att mer frekvent kunna måla om skärmen (se bild 2). Klassen *TowerDefenceComponent* är den klass som har hand om uppritning av spelplanen, fiender, torn och bullets. Genom att anropa BoardListener när vi ändrat spelplanen kommer vi rita om spelplanen oftare vilket medför att spelets animering kommer flytta på smidare och bli mindre ”hackigt”.

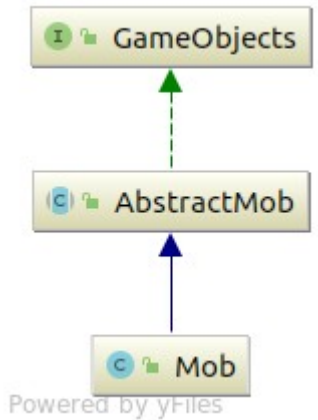

Bild 3. UML- diagram över hur Mob ärver från sin superklass AbstractMob.

Högst upp i klasshierarkin har vi ett interface kallat *GameObjects (se bild 3)* där vi har placerat metoder som delas av våra olika objekt på spelplanen. Detta interface är framförallt användbart vid vidareutveckling av spelet då metoder som delas av flera olika ”GameObjects” kan samlas på ett och samma ställe. Vidare har vi valt att implementera *AbstractMob* som en abstrakt klass där vi definierar olika fält och metoder som vi anser ska tillhöra alla mobs såsom koordinater, räckvidd, kostnad med mera. Slutligen har vi implementerat en *Mob* (fiende) klass som är den vi använder oss av i spelet. Denna klass ärver då superklassens konstruktor samt metoder och fält. Genom att samla flera metoder och fält i den abstrakta klassenen kan fler mobs utvecklas på ett smidigt genom att låta de ärva superklassens konstruktor och sedan låta de specifika metoderna vara definierade i klassen.

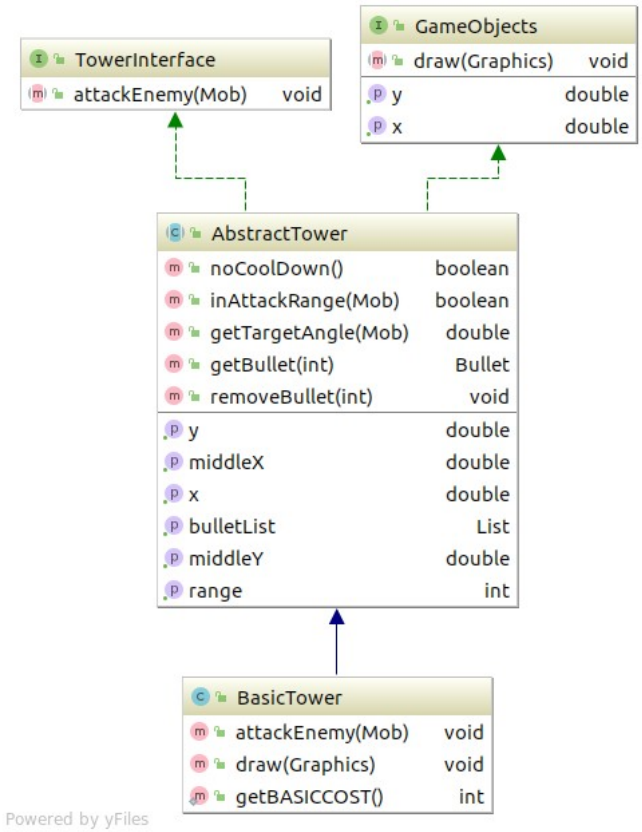

Bild 4: UML-diagram över hur BasicTower implementerar gränssnitteten TowerInterface och ärver superklassen AbstractTower.

På samma sätt som hos vår Mob implenterar ”Abstract Tower” ett interface GameObjects som innehåller gemensamma metoder för alla GameObjects (se bild 4). Vidare implementerar även vårt torn *TowerInterface* som innehåller metoder som alla torn bör ha tillgång till. AbstractTower innehåller i sin tur fält och metoder som alla torn kan ha nytta av. Slutligen använder vårt *BasicTower* detta genom att ärva från AbstractTower. Även här är det lätt att vidareutveckla fler torn genom att ärva från superklassen. Exempelvis kan man se att i vårt TowerInterface finns metoden *attackEnemy*. Detta är en metod som vi anser att alla torn kommer behöva däremot kan den vara utformad olika hos varje torn.

Objektorientering inom projektet

Objekt/klasser

Inom detta projekt av vi utnyttjat att Java är ett objektorienterat språk. Genom att dela upp koden i olika klasser kan man på ett strukturerat sätt bilda olika typer av objekt. Varje klass fungerar som en ritning till ett visst objekt där vi berättar vilka attribut objektet behöver, vad som krävs när vi skapar den samt vilken funktionalitet den ska ha. Ett exempel är vår klass ”BasicTower” som ärver sina fält från superklassen ”AbstractTower”. Detta gör den då dessa fält kan ses som generella för alla torn. Varje torn behöver en position, en viss attackradie, en s.k. *cooldown* (dvs en viss tid då vi ej kan avfyra några skott) och en lista med *bullets* (skott) då vi vill att varje torn ska ha kontroll över sina egna bullets. Givet detta krävs endast två koordinater för att skapa ett ”BasicTower”. Vidare kan sedan varje Tower ha sin egen unika funktionalitet i sin egen klass exempelvis, kraftigare skott, skjuter snabbare, målsökande skott etc.

Typhierarkier

Vidare har vi även använt oss av typhierarkier på olika sätt. Klassen AbstractTower implementerar TowerInterface, dvs AbstractTower implementerar de olika metoder som finns i gränssnittet (Interface). Detta blir då en del av ett kontrakt, vi lovar att varje AbstractTower kommer ha dessa metoder. I vårt fall har vi valt att lägga metoden *attackEnemy* i gränssnittet då varje Tower bör ha en attackmetod. Vidare använder vi oss av *Overriding* vid

denna metod det vill säga vi implementera metoden men har sedan vår egen unika implementation av den metoden i vår klass. Detta kan vara användbart ifall vi vill att torn ska kunna attackera fiender på olika sätt exempelvis genom att skjuta flera skott samtidigt eller genom att skjuta specifika skott. För vidare utveckling av spelet kan fler metoder komma att läggas till i gränssnittet.

Subtyppolymorfism

Vi har i vår klass Controller utnyttjat subtypspolymorfism genom att låta de listor som har hand om torn och mobs ta in deras Abstrakta klasser. På så vis spelar det ingen roll vilket torn vi skapar så länge det ärver AbstractTower för då är den även av typen AbstractTower. På det sättet kan vi lägga dem i samma lista och manipulera enbart den listan istället för en lista för varje typ av Tower.

Inkapsling

Inkapsling är ett sätt att hålla fälten i en klass privata. Ingen utomstående ska kunna ändra på dem utom klassen själv. Vi använder oss av *Setters* eller *Getters* när vi vill ha tag på ett värde eller ett element i en lista istället för att kalla på det via klassen exempelvis genom att skriva ”BasicTower.x”.

Övriga implementationsdetaljer

Timer-metod

Det spel som vi utvecklat drivs framåt av en så kallad ”timermetod” det vill säga en funktion som går i en loop och anropar sig själv med ett givet intervall. Metoden ligger i klassen *TowerDefenceStart* och metoden heter *main*. I sagda funktion anropar vi metoden *tick* i klassen *Board* som vid varje tick (anrop från timermetod) uppdaterar spelet. Givet att spelet inte är förlorat kommer tickmetoden att uppdatera positionerna för våra *mobs* (fiender) samt uppdatera våra torn genom att söka igenom deras attackområde efter mobs. Ifall en mob skulle påträffas kommer tornen att skjuta mot nämnda mob. Vidare uppdaterar vi vårt eget liv ifall vi tagit skada samt uppdaterar ifall vi fått några pengar från eventuellt eliminerade mobs. Slutligen uppdaterar vi den aktuella *waven* (grupp av fiender) för att se ifall den waven är avklarad.

EnumKlasser för Block, Buttons och Directions

Olika strukturer inom programmet bygger på olika *Enum*-klasser, det vill säga klasser som består av flera olika objekt. Detta är speciellt användbart då vi enbart vill tillåta olika typer av objekt. Vi har använt oss av detta på flera ställen, exempelvis när vi utformat spelplanen och de olika block som ska ingå i terrängen. Härvid blir det väldigt smidigt med enums då vi slipper kontrollera att vi får in data av rätt typ när vi skapar ett *Map*-objekt. Vidare finns då endast en viss uppsättning av block till förfogande vilket medför att det blir omöjligt att få in block som inte skapats av utvecklaren. Vi använder oss även av enums när beräknar olika riktningar för vår mob samt när vi trycker på olika knappar i menyn.

Kollisioner och navigering genom spelplanen

Vi har framförallt använt oss mycket av avståndsberäkningar för att kunna simulera kollisioner och navigering. När varje mob rör sig kontrollerar den ifall den nått mitten av nästa block. För att veta när vi nått mitten använder sig varje mob av en metod för att beräkna avstånd till mitten av nästa block, *middleDistance*. När vi är inom ett givet intervall från mitten av nästa block börjar vi leta efter en ny *Direction* (En enum, se klassen Direction) som vi sedan rör oss i.

På samma sätt beräknar även BasicTower ett avstånd till alla fiender på spelplanen. Befinner sig fienden inom ett visst avstånd till tornet kommer tornet att skjuta. BasicTower gör detta med sin egen implementation av *attackEnemy.* Vidare använder sig varje Bullet av en metod *withinDistance* för att beräkna ifall de är tillräckligt nära en mob för att skada den och sedan själva försvinna.

6.3. Användning av fritt material

Vi har i vårt projekt enbart använt hos av Java 11 samt MigLayout för grafik.

6.4. Motiverade designbeslut med alternativ

Numrerade svar:

1. Till vår spelplan önskade vi kunna använda oss av olika typer av ”block” för att på ett smidigt sätt kunna kontrollera var våra fiender får förflytta sig samt var vi som spelare får placera ut våra torn. För att realisera detta skapade vi en Enum-klass *BlockType* där våra olika block sparas som enum-konstanter. En alternativ lösning till detta hade varit ha låta varje block representeras av en siffra eller av en textsträng. Fördelen med enums är att det blir objekt vilket passar utmärkt då vi jobbar objektorienterat. Vi kan med andra ord behandla dem som objekt. Exempelvis blir det lättare att kontrollera att vi får in rätt objekt i en metod då den måste vara av typen *BlockType*. Ifall vi använt oss av textsträngar hade vi behövt kontrollera varje gång att det som vi tog in verkligen beskrev ett block. Genom att använda enums försäkrar vi oss om att vi inte kan få in ”fel” typ av information.

2. Vi ville kunna skapa banor på ett modulärt och generellt sätt. Det funktionalitet vi sökte var att varje bana skulle bli ett eget objekt som i sin tur lätt skulle gå att manipulera. Antingen för att kunna modifiera existera banor eller för att med lätthet kunna skapa nya. För att uppnå detta skrevs klassen *MapMaker* samt klassen *Map*. I klassen MapMaker finns metoden *mapSelector* som givet en siffra väljer en specifik bana ur en switchsats. Varje spelplan är då uppbyggd som en 2D-array fylld med BlockTypes där vi använder s.k. ”*array initialiserzs*” för att kunna placera in block i förväg i själva kartan. Vidare skickar vi 2D-arrayn till konstruktorn i klassen Map för att på så sätt göra listan till ett objekt av typen Map. Slutligen tar klassen Board:s konstrukor in ett objekt av typen Map. Härvid itererar vi över Mapen och skapar alla olika block. Ett alternativ hade varit att direkt i board skapa själva banan genom att iterera igenom listan och placera ut ett givet block på varje position. Det som gjorde vår lösning bättre är att vi kan placera flera olika block på olika platser på ett betydligt mer lättöverskådligt sätt än att behöva modifiera olika for-loopar.

3. Vi ville uppnå en bättre struktur på koden och framförallt en grund på som lätt gick att bygga vidare på. Det behandlade framförallt våra towers och våra mobs. För att uppnå detta skapade vi en Abstract klass till vardera klass för att på ett mer strukturerat sätt hantera den kod vi ansåg vara generell. Ett alternativ till detta hade varit att låta varje klass ha hand om sina egna fält och metoder, dock hade detta varit onödigt då Java (och framförallt objektorientering) tillåter oss att ärva. Därmed är det bättre att utnyttja detta.

4. Vi ville på något sätt försöka samla ihop de olika listor som vi har våra spelobjekt i. För att strukturera upp detta skapade vi klassen Controller som kort sagt har hand om alla listor samt uppdaterar dem. Ett möjligt alternativ till detta hade antingen varit att placera samtliga listor i Board och låtit dem tillhöra just den klassen. Vi bestämde oss dock för att detta inte passade då Board i sig bara ska ha hand om själva spelplanen, inte objekten på den. Därmed passade det bättre med en kontrollklass som har hand om detta och som då får en tydlig roll.

5. Vi ville på något sätt skapa ett gränssnitt som användaren kunde använda för att interagera med spelet. Vi implementerade därmed ett antal knappar som spelaren kan använda för att köpa nya torn, sälja befintliga eller kalla på nästa våg av fiender. Dessa knappar kan hittas i en panel till höger om spelplanen. Härvid kan vi även se

vårt eget liv, hur mycket pengar vi har samt vilken våg vi är på. Ett möjligt alternativ hade kunnat vara att istället binda olika tangenter till de olika funktionerna. Det hade dock medfört att man hade behövt någon form av manual för att kunna tyda vilka knappar som gör vad. Vidare blir det mer intuitivt att klicka på knappar då de förklara sig själva samt att det är lättare att utöka knappar och menyer än tangenter. Tangenterna tar tillslut slut och det blir lätt för komplicerat.

6. Vi önskade rita ut de olika objekt som finns i vårt spel däribland torn, mobs och skott. För att göra detta samlande vi den koden som krävdes i klassen *TowerDefenceComponent* som hanterar uppritning av de olika objekten. Kort sagt itererar den över listorna med objekt och ritar ut dem. En alternativ lösning hade varit att låta varje objekt ha en *draw*-metod där de ritar ut sig själva. Denna metod hade sedan kunnat finnas i ett interface som samtliga objekt kan implementera. Vi valde dock att använda vårt sätt då uppritning ser likadan ut för alla våra objekt och då vårt projekt är rätt litet i omfattning. Vi inser dock att ur ett objektorienterat perspektiv hade det varit ”mer” korrekt att låta varje objekt rita sig själv.

7. Användarmanual

Hur spelas Tower Defence?

”Tower Defence” är ett klassiskt spel inom strategispelsgenren. Spelet går i huvudsak ut på att fiender kommer röra sig enligt en given väg på spelplan. När fienderna når sitt mål kommer de göra skada på spelaren och tar spelaren för mycket skada förlorar man spelet. Vi som spelare har 10 HP (hälsopoäng, d.v.s. hur mycket skada vi tål innan vi dör). Spelarens uppgift blir därmed att placera ut olika defensiva strukturer för att stoppa fienderna från att nå sitt mål. Dessa defensiva strukturer kan köpas via köpmenyn för pengar som spelaren tjänar ihop genom att eliminera fiender. Spelet är uppbyggt kring så kallade ”*waves*”, det vill säga att varje ny nivå innebär att en ny grupp fiender anfaller spelaren. Efter varje avklarad *wave* får spelaren en ”wave-bonus” det vill säga 50 guld för att kunna köpa fler torn. *Klarar* man av alla tio nivåer på en bana vinner man spelet.

Spelplanen och terräng

Spelet innehåller olika föremål som går att interagera med samt olika typer av terräng. Nedan listar vi dessa och berättar om deras olika funktioner.

Spelplanen

Gräs


Det gröna blocket representerar gräs och är ett vanligt förekommande block. Härvid kan spelaren placera sina torn.

Vatten


Det ljusblå blocket representerar vatten och är mer sällsynt. Härvid kan inte spelaren placera sina torn.

Väg


Det beigea blocket representerar vägen som våra fiender kommer röra sig längs. Härvid kan spelaren inte placera sina torn.

Sten


Det gråa blocket representerar sten. Härvid kan spelaren placera sina torn.

Skapare

Det rosa blocket representerar vårt ”skaparblock” det vill säga det blocket som kommer skapa våra olika fiender. Från detta block kommer fienderna sedan röra sig längs med väg-blocket (se ovan). Härvid kan spelaren ej placera sitt torn.

Mål

Det blå blocket representerar målet för våra fiender. När våra fiender når detta block kommer det göra skada på spelaren och sedan försvinna. Härvid kan spelaren inte placera sina torn.

Mur

Det svarta blocket representerar vår mur som omringar banan. Muren ramar in banan och visar dess storlek. Härvid kan vi inte placera några block och vi kan inte röra oss bortom muren. Träffar ett skott muren kommer det försvinna.

Objekt

Fiender

Det röda blocket representerar våra fiender. Våra fiender kommer röra sig längs den givna vägen på spelplanen mot målet. Når en fiende målet kommer den skada spelaren genom att minska dess liv med ett. Fienden har 8 HP.

Torn

Det gula blocket representerar våra torn. Varje torn kostar 100 guld och kan placeras på antingen gräs eller sten. Runt varje torn visas dess attackradie i form av en svart cirkel. Kommer en fiende inom denna cirkel kommer tornet skjuta attackera den genom att avfyra skott mot den. Varje skott skadar 1 HP.

Spelets gång och funktioner

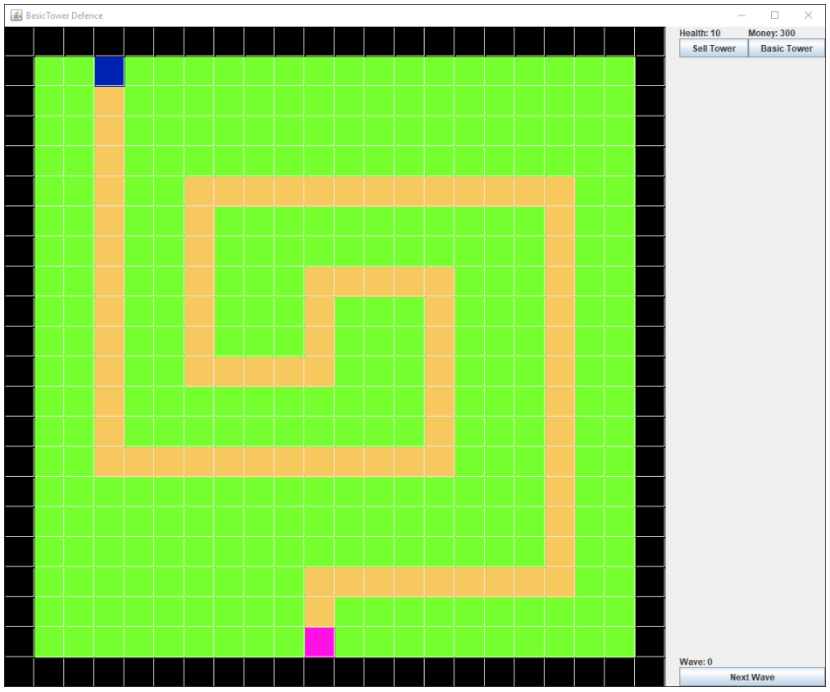

Bild 5: Startskärmen. Spelplanen till vänster och köp- och säljmenyn till höger.

Precis när man startar spelet möts man av spelets startskärm (bild 5). Härifrån kan vi se spelplanen och köp-menyn till höger.

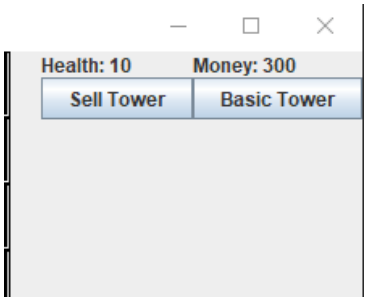

Bild 6. Köp-meny, HP och Guld

Tar vi en närmare titt på köpmenyn (bild 6) ser vi vårt eget HP, hur mycket guld vi har samt om vi vill sälja eller köpa torn. Genom att klicka på ”Basic Tower” kommer spelaren sättas i köpläget vilket tillåter oss att placera ut torn genom att helt enkelt klicka på den ruta vi önskar placera tornet. Spelaren kommer kunna placera ut torn så länge vi har tillräckligt med guld. Önskar spelaren sälja sina torn klickar vi på ”Sell Tower”. Då placeras spelaren i säljläget och genom att då klicka på ett torn kommer tornet försvinna och vi återfår våra de guld vi investerat.

Bild 7. Starta nästa runda.

I den nedre delen av menyn finner vi en knapp som startar nästa *wave (bild 7)*. Genom att klicka på den knappen kommer fiender börja skapas och vi kan inte längre köpa eller sälja torn. Vi kan även se vilken nivå vi är på.

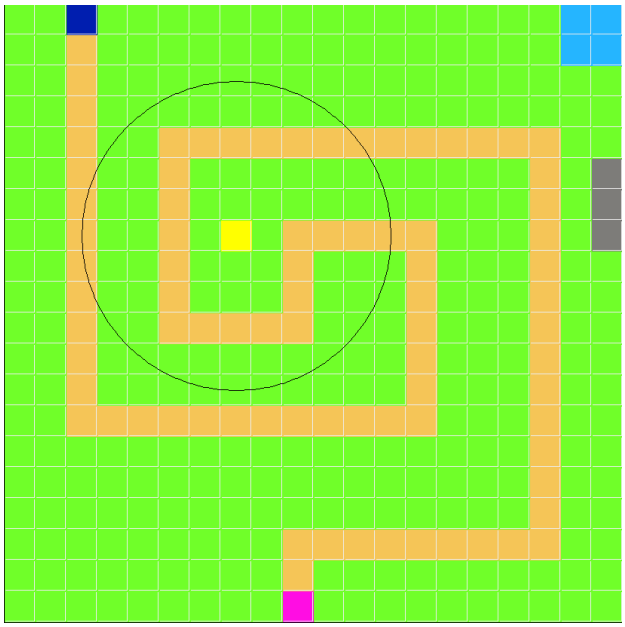

Bild 8. Ett torn utplacerat.

Inledningsvis kan vi placera ut ett torn på spelplanen genom att köpa det via menyn (bild 8). Den svarta cirkeln runt tornet representerar dess attackradie. Kommer en fiende inom cirkeln kommer tornet attackera den.

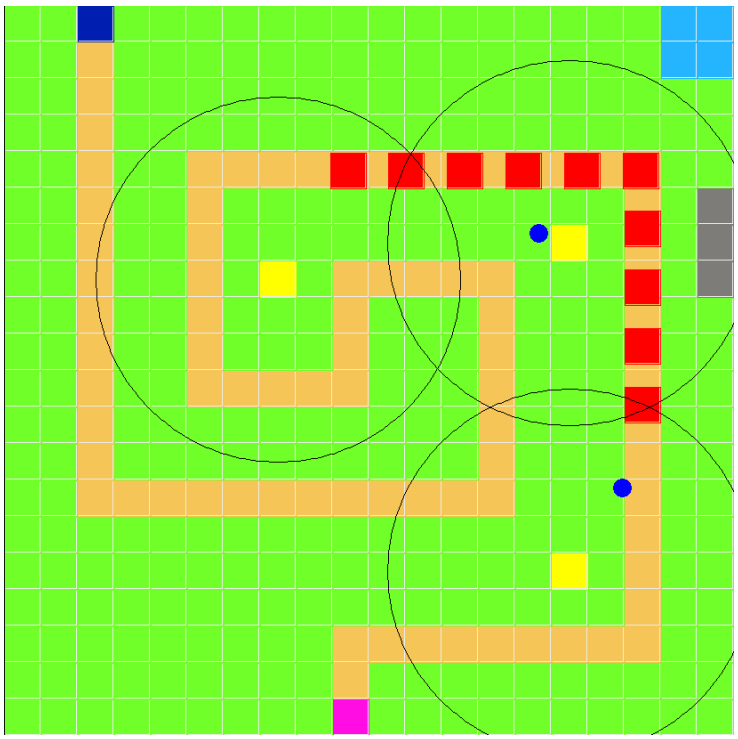

Bild 9. Fiender på ingång och torn som skjuter. De blå cirklarna representerar våra skott.

Fienderna kommer röra sig enligt den givna banan och tornen kommer försöka stoppa dem genom att skjuta på dem (bild 9). Varje karta består av 10 nivåer där varje ny nivå innebär större grupper av fiender. Placera ut dina torn taktiskt, hushåll med guld och ha framförallt kul när du spelar vårt Tower Defence.

8. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
  Samarbetet har fungerat väldigt bra mellan oss. Vi insåg rätt snabbt att det var bättre att jobba tillsammans för att kunna hjälpas åt och för att vi både skulle förstå hur koden var utformad. Vi har båda varit lagom ambitiösa och siktat på att få en fungerade produkt som det ska vara enkelt att bygga vidare på. Vi har även fått bra hjälp av vår handledare och mycket bra input för att kunna utveckla spelet i rätt riktning. Det som möjligt har fungerat ”dåligt” har varit att få in alla de timmar som projektet krävt även om vi planerat och skrivit ner den tid vi suttit.

- *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har ”hjälpt” bäst?*
  I början av kursen gav föreläsningarna en bra grund för att få förståelse för själva objektorienteringen vilket såklart är viktigt då man programmerar i Java. Under projektets gång har vi främst använt oss av internet då man ofta söker specifika lösningar på olika problem till exempel hur får vi muspekaren att interagera med programmet. En av oss har läst lite i boken (Thinking in Java) men kände rätt snabbt att mycket blir irrelevant eller att man snabbare kan hitta mer specifik information på internet så man slipper bläddra igenom 1000 sidor text. Vi har gått på alla handledda labbar och ställt mycket frågor för att framförallt få råd då vi har flera olika sätt att implementera en funktionalitet i vår kod till exempel hur våra fiender ska kunna röra sig i en given bana. Det som hjälpt bäst skulle jag vilja säga är internet och de

handledda labbarna för specifika lösningar medan föreläsningarna gav en bra grund för objektorientering men ej så mycket specifik information.

- *Har ni lagt ned för mycket/lite tid?*
  Det som var rekommenderat var ca 60 timmar och vi har lagt runt den tiden. För vår del har det räckt för att färdigställa spelet och finputsa koden. Vi hade kunnat lägga mer tid på att öka antalet fiender och torn för att göra spelet mer dynamiskt men inget av dessa är direkt nödvändiga för att spelet ska kunna spelas.
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
  Arbetsfördelning har varit jämn då vi alltid har programmet tillsammans och då vi turats om med att vara den som just knackar koden.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
  Det som framförallt har varit viktigt är själva milstolparna för att ha någon form av plan vart man vill med projektet. Blir man klar med något vet man snabbt vad nästa steg blir. Även att skriva ner i början hur vi tänkt att de olika klasserna ska hänga samman har hjälpt för att få ihop helheten. Det kanske inte hjälpt just under spelets gång med det är viktigt att tidigt börja tänka hur man kan göra och vad som passar bäst.
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
  Arbetet har fungerat bra mellan oss. Oftast brukar vi diskutera hur vi ska göra och vilka för- och nackdelar som finns med de olika implementationerna. Kommer vi inte fram till något brukar vi fråga laborationsassistenten för råd.
- *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*
  Ingenting har väl direkt var problematiskt för vår del, vi har alltid hittat något ställe samt ledig tid.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
  Gör som oss, gör ett textdokument i själv projektet där ni skriver ner era designbeslut och hur mycket tid ni lägger samt hur mycket ni borde lägga, väldigt bra för att hålla koll på sig själv. Utöver det försök att varje pass boka in när ni ska sitta nästa pass så det aldrig blir oklart eller i värsta fall ogjort.
- *Har ni saknat något i kursen som hade underlättat projektet?*
  Nej inte vad vi kan komma på.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
  Kanske lite rekommenderande alternativa sätt att lära sig objektorientering och Java. Med andra ord istället för böcker hade det varit kul med lite bra YouTube-kanaler eller andra bra hemsidor om man vill lära sig mer och inte läsa en 1000 sidor lång bok. En YouTubekanal jag lärt mig mycket av är ”The Coding Train”.