Rogue Runner

Design och utveckling av ett spel i Java

2023-03-10

Projektmedlemmar:

Samuel Åkesson <samak519@student.liu.se>

Daniel Alchasov <danal315@student.liu.se>

Handledare:

Linus Gardshol <linga754@student.liu.se>

Innehåll

Projektplan

Läs först:

- https://www.ida.liu.se/~TDDD78/labs/2023/project/intro
- https://www.ida.liu.se/~TDDD78/labs/2023/project/select
- https://www.ida.liu.se/~TDDD78/labs/2023/project/documents

Ni väljer själva vilken sorts projekt ni vill utföra, men det finns vissa begränsningar som man behöver tänka på. Läs om detta på ovanstående sidor. Det är viktigt för att ni inte ska måla in er i ett hörn med ett projekt som inte är lämpligt för kursen.

Projektplanen skriver ni i samband med första inlämningen, gärna under tiden ni arbetar på sista labben för att göra det möjligt att få kommentarer innan projektstart. Små kompletteringar kan göras senare, men försök få med så mycket som möjligt redan från början.

§ **Använd denna mall**, i t.ex. LibreOffice eller Word -- inte en egentillverkad

§ **Låt alla rubriker vara kvar**, med samma **numrering** och **rubriktext**

§ **Ta bort de övriga instruktionerna**, till exempel denna text och ”Beskriv relativt kortfattat…” nedan, när ni har gjort vad instruktionerna säger!

1. Introduktion till projektet

Vi kommer att skapa ett non-linear RPG-spel där spelaren “levelar upp” genom att tjäna poäng när den dödar monster, löser problem, gräver upp kistor etc. Spelaren kommer att plocka upp mer och mer utrustning och varje föremål kommer att ha olika attributer som hur ändrar hur mycket du skyddas mot skador / hur mycket skada du gör mot fiender.

Spelet kommer att ha en underliggande berättelse, men kommer inte att vara speciellt utvecklad eller relevant för projektet. Störst fokus kommer ligga på att spelaren kan plocka upp nya svärd / rustning för att bli starkare, och det kommer att finnas olika typer av fiender som beter sig annorlunda.

Projektet kommer att vara väldigt inspirerat av The Legend of Zelda:

2. Ytterligare bakgrundsinformation

Det är ett topdown RPG-Spel i pixel art som man ska gå genom en kortfattad story och bli starkare och levela upp. Kanske tar en liten inspiration från JRPGS och andra RPGS mekaniker för att uppfylla alla kriterer. Vi tänker låta AI skapa storyn för spelet och programmera efter det.

3. Milstolpar

Tänk efter **hur ni kan utveckla programmet stegvis** så att ni hela tiden kan testa koden, verifiera att den fungerar, och se konkreta resultat! Programmering är roligt, men att verkligen se ett fungerande program är ännu roligare.

Konkretisera stegen genom att dela upp projektet i en sekvens **milstolpar,** där varje milstolpe anger en viss funktionalitet som ska finnas i projektet. Tanken är att ni anger ett antal sådana milstolpar under planeringen och därefter använder dem som ledning för er själva under implementationsfasen. Varje steg bygger vidare på de tidigare stegen med ny funktionalitet. Efter varje steg, inklusive det första, ska det finnas en testbar ”produkt” som har någon form av meningsfull funktionalitet. **Detta är något som många tidigare studenter säger sig ha haft mycket nytta av!** Exempel ges nedan.

Det är väldigt svårt att i förväg veta exakt hur mycket som är rimligt att implementera under kursens gång. Därför vill vi att ni anger ett antal milstolpar där ni tror att de inledande **med lätthet kommer att hinnas med**, medan de mot slutet av listan kommer att **ta betydligt mer tid** än ni har under kursen. För de flesta projekt bör det gå relativt lätt att dela upp funktionaliteten i minst **20-25** steg, eller varför inte **40-50**?

Listan är alltså **inte en kravlista** där någon kommer att klaga om ni inte uppnår allt, utan är helt och hållet tänkt att vara en vägledning till er själva (och ge labbhandledarna en möjlighet att kommentera och ge förslag, så klart). Genom att tänka igenom stegen i förväg, och även ta med sådant som ni troligen inte hinner med, får ni ett bättre underlag när ni ska designa grunden i ert projekt. Att veta att man kanske någon gång i sitt spel ska implementera nätverksstöd eller låta flera användare spela på samma gång kan till exempel påverka även er första uppbyggnad av spelarobjekt och liknande. Dessutom har ni garanterat att ni har ett sätt att gå vidare om projektet visade sig vara enklare än ni trodde, så ni behöver mer funktionalitet för att få tillräcklig omfattning.

Stegen i listan bör till största delen bestå av **mätbara krav** (även om det alltså inte är någon som kommer att titta vilka ”krav” som har uppfyllts). Med andra ord, man bör kunna svara objektivt "ja" eller "nej" på om ett steg är uppnått eller inte, vilket man till exempel kan för "Spelet ska kunna styras både med tangentbord och med mus". Om ni skriver att "spelet ska vara snyggt" är det svårare att bedöma – men eftersom det inte är en ren kravlista kan ni t.ex. skriva ner att ”I detta steg lägger vi lite mer tid på spelets utseende”.

Skriv ner milstolparna genom att bygga ut tabellen nedan. Exempel ges för Tetris-spelet.

Använder ni ett inspirationsprojekt? I många fall är de angivna milstolparna i beskrivningen väldigt preliminära och ungefärliga. Ni behöver *vidareutveckla* dem för att beskriva i mer detalj vad som ska göras och hur just *ni* kommer att realisera dem. Ni kan också vilja ta bort, lägga till eller ändra milstolpar.

#	**Beskrivning**
1	Det första som behövs är en grafisk ramverk för att hålla alla våra komponenter. Som uppdaterar varje tick med repaint. Ordningen saker paintas: 1. Map tiles 2. Entities 3. GUI
2	Vi ska skapa datatyper för polymorphism. En enum “items”, enum “enemies” eller “characters”.
3	**Enum ItemType** kommer att användas för att seperera objekt i klasserna liknande som det var i tetris “squaretypes” så under “Item”kommer det vara necklace, weapon osv…
4	Vi kommer behöva ett “**visible gameobjects**” där objekt i omvärlden läggs i och andra klasser inheritar den. Samt ge objekt kollision eller inte.
5	Vi måste ha fält **EXP** på characters som bestämmer vilken LVL karaktären är. Kanske att HP och DMG ska räknas ut direkt från LVL.
6	Med att det finns olika former av enteties som kommer att behöva ha olika stats men de alla kommer att inherita “base stats” från base objektet.
7	Bygga upp **maps** på ett enkelt sätt, kanske skapa en metod som omvandlar bilder med färgade pixlar till typer av tiles.

8	Skapa movement och keybinding till de. Movements som: - Framåt - backåt - sidledes Samt ska det kunnas inputa framåt och sidledes för att gå horonsintelt
9	Ett sätt att tillhöra objekt i spelet till bilder.
10	Skapa enkla items och karaktärer samt enemeies för testning.
11	Skapa ett GUI som visa HP, EXP, DMG och kanske även inventory, stats skärm och annat.
12	Skapa action based combat. Idén är att spelaren enemies gör attacker som går ur karaktären och har kollision med omvärlden. Olika enemies har olika weakness (weak to blunt / weak to arrows / weak to magic / resistant to explosives)
13	Lägga inputs för att attackera.
14	Tillägg olika items som delvis ger dig andra moves eller högre stats.
15	Skapa någon sorts skill path liknande till rougelites. Som du får välja mellan när du levlar och/eller vid specilla ställen vid storyn.
16	Spela upp ljudeffekter / musik

17	Spel intro. Som introducerar spelaren intill spel världen.
18	
19	
20	
21	
22	
23	
...	

4. Övriga implementationsförberedelser

Klasser för objekt i spelet som kommer vara “huvudklassen” för de flesta subklasser som enemes, character items och objekt som går ej med att inraktrera med. Förutom det behövs någon enkel sätt att skapa maps hittas.

5. Utveckling och samarbete

Liknande ambitionsnivå och vi planerar träffas då och då (troligtvis en gång i veckan) och gå genom det båda har kodat själva. Vi tänker jobba ganska separat och distribuerat, men kommer att berätta vad vi håller på att implementera just nu så vi aldrig gör dubbelt arbete och får merge-conflicts.

(Resten av dokumentet ska inte lämnas in förrän projektet är klart, men titta ändå genom allt för att se vilka delar ni behöver arbeta med och fylla i *kontinuerligt* under projektets gång!)

Projektrapport

Även om denna del inte ska lämnas in förrän projektet är klart, är det **viktigt att arbeta med den kontinuerligt** under projektets gång! Speciellt finns det några avsnitt där ni ska beskriva information som ni lätt kan glömma av när veckorna går (vilket flera tidigare studenter också har kommenterat).

Tänk på att ligga på lagom ambitionsnivå! En välskriven implementationsbeskrivning (avsnitt 6) hamnar normalt på **3-6 sidor** i det givna formatet och radavståndet, med ett par mindre UML-diagram och kanske ett par andra små illustrerande bilder vid behov. Hela projektrapporten (denna sista halva av dokumentet) behöver sällan mer än 10-12 sidor.

6. Implementationsbeskrivning

I det här avsnittet, och dess underavsnitt (6.x), beskriver ni olika aspekter av själva *implementationen*, under förutsättning att läsaren redan förstår vad *syftet* med projektet är (det har ju beskrivits tidigare).

Tänk er att någon ska vidareutveckla projektet, kanske genom att fixa eventuella buggar eller skapa utökningar. Då finns det en hel del som den personen kan behöva förstå så att man vet *var* funktionaliteten finns, *hur* den är uppdelad, och så vidare. Algoritmer och övergripande design passar också in i det här kapitlet.

Bilder, flödesdiagram, osv. är starkt rekommenderat!

Skapa gärna egna delkapitel för enskilda delar, om det underlättar. **Ta inte bort några rubriker!**

Även detta är en del av examinationen som visar att ni förstår vad ni gör!

6.1. Milstolpar

Ange för varje milstolpe om ni har genomfört den helt, delvis eller inte alls.

Detta är till för att labbhandledaren ska veta vilken funktionalitet man kan ”leta efter” i koden. Själva bedömningen beror inte på antalet milstolpar i sig, och inte heller på om man ”hann med” milstolparna eller inte!

1. Helt
2. Delvis. Vi har skapat klasser som Entity som har underklasser, men ingen klass Item som skulle kunna fungera som en superklass till de olika föremålen spelaren kan interagera med.
3. Delvis. Enumerable visade sig fel sätt att tänka kring att bestämma typen på föremålet eftersom varje föremål kunde ha ett ganska annorlunda beteende från

varandra. Vi skapade överklassen Weapon som innehåller Projectile och Sword, samt Potion.

4. Helt. Alla objekt som är synliga och har kollision är av typen Entity.
5. Helt. Alla objekt av typen Character har två fält: exp och level. När spelaren dödar fienden ökar spelarens exp, och om kraven är uppfyllda, också dess level.
6. Helt.
7. Helt.
8. Helt.
9. Helt.
10. Helt.
11. Delvis
12. Delvis
13. Helt
14. Delvis
15. Delvis
16. Inte alls.
17. Inte alls.

6.2. Dokumentation för programstruktur, med UML-diagram

Programkod behöver dokumenteras för att man ska förstå hur den fungerar och hur allt hänger ihop. Vissa typer av dokumentation är direkt relaterade till ett enda fält, en enda metod eller en enda klass och placeras då lämpligast vid fältet, metoden eller klassen i en Javadoc-kommentar, *inte här*. Då är det både enklare att hitta dokumentationen och större chans att den faktiskt uppdateras när det sker ändringar. Annan dokumentation är mer övergripande och saknar en naturlig plats i koden. Då kan den placeras här. Det kan gälla till exempel:

- **Övergripande programstruktur**, t.ex. att man har implementerat ett spel som styrs av timer-tick n gånger per sekund där man vid varje sådant tick först tar hand om input och gör eventuella förflyttningar för objekt av typ X, Y och Z, därefter kontrollerar kollisioner vilket sker med hjälp av klass W, och till slut uppdaterar skärmen.

- **Översikter över relaterade klasser** och hur de hänger ihop.
  - Här kan det ofta vara bra att använda **UML-diagram** för att illustrera – det finns även i betygskraven. Fundera då först på vilka grupper av klasser det är ni vill beskriva, och skapa sedan ett UML-diagram för varje grupp av klasser.
  - Notera att det sällan är särskilt användbart att lägga in hela projektet i ett enda gigantiskt diagram (vad är det då man fokuserar på?). Hitta intressanta delstrukturer och visa dem. Ni behöver normalt inte ha med fält eller metoder i diagrammen.
  - **Skriv sedan en textbeskrivning av vad det är ni illustrerar med UML-diagrammet**. Texten är den huvudsakliga dokumentationen medan UML-diagrammet hjälper läsaren att förstå texten och få en översikt.
  - IDEA kan hjälpa till att skapa ett klassdiagram som ni sedan kan klippa och klistra in i dokumentet. Högerklicka i en editor och välj Diagrams / Show Diagram. Ni kan sedan lägga till och ta bort klasser

med högerklicksmenyn. Exportera till bildfil med högerklick / Export to File.

I det här avsnittet har ni också en möjlighet att visa upp era kunskaper genom att diskutera koden i objektorienterade termer. Ni kan till exempel diskutera hur ni använder och har nytta av (åtminstone en del av) objekt/klasser, konstruktorer, typhierarkier, interface, ärvning, overriding, abstrakta klasser, subtypspolymorfism, och inkapsling (accessnivåer).

Labbhandledaren och examinatorn kommer bland annat att använda dokumentationen i det här avsnittet för att förstå programmet vid bedömningen. Ni kan också tänka er att ni själva ska vidareutveckla projektet efter att en annan grupp har utvecklat grunden. Vad skulle ni själva vilja veta i det läget?

När ni pratar om klasser och metoder ska deras namn anges tydligt (inte bara ”vår timerklass” eller ”utritningsmetoden”).

Framhäv gärna det ni själva tycker är **bra/intressanta lösningar** eller annat som handledaren borde titta på vid den senare genomgången av programkoden.

Vi räknar med att de flesta projekt behöver runt **3-6 sidor** för det här avsnittet.

6.2.1 Övergripande struktur

Sättet som spelet fungerar följande. Det instansieras ett Viewer-objekt, vilket sköter allting med spelet så fort man kallas på dess metod “show”. På den här nivån är det som “ryska-dockor” (Matryoshka). Ett Viewer-objekt innehåller ett GameComponent-objekt som i sin tur innehåller ett Game-objekt. Ett Viewer-objekt rent praktiskt tar fram en JFrame och lägger en GameComponent ovanpå den, med hjälp av Swing:s egna syntax. Viewer-objektet sköter också själva timing i spelet, genom att kalla på dess GameComponent:s frameChanged-metod, samt GameComponent:ens Game-objekt:s tick-metod vid jämna tidsintervall.

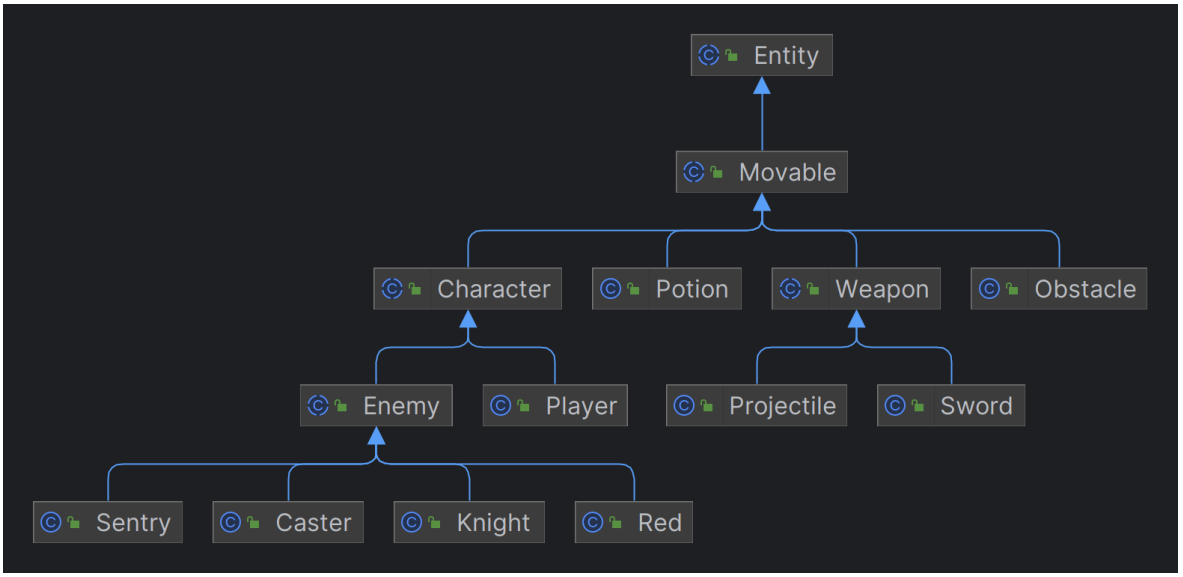
“frameChanged-metoden” helt enkelt kallar på repaint som är Swing:s egna metod som vi overidar.

Game-objektet:s tick-metod hanterar helt enkelt allt som sker i spelet, exempelvis att Entity:s ska förflyttas enligt sin hastighet, samt puttas ifrån varandra eller ta skada om de kollideras.

GameComponent extendar JComponent och implementar vår interface FrameListener som har vår frameChanged-metod. GameComponent i enklaste nivå är objektet med hela spelet visuellt på sig. Fältet “game” är helt enkelt spelet där gameComponent tar allt information från för att rita det visuella. Iallafall använder GameComponent swings paintComponent metod för att rita allting. Metoden kallar på alla våra metoder vi skapade som ritar varje aspekt av spelet, som paintMapLayer, paintEnteties och så vidare. Ordningen på paintComponent är baserat på sättet spelaren ska se på spelet så att mappen inte är över spelaren. GameComponent hanterar även keyinputs med AbstractActions. Här hanterar vi key presses och key releases på så sätt när man trycker in och släpper sker saker som att starta movements eller stoppa dem.

Banor laddas in på ett väldigt flexibelt sätt. Det finns en klass “RoomFileLoader” som tar in ett filnamn till en .tsx-fil skapad av tredjepartsprogrammet “Tiled”. Klassen använder sig av det externa biblioteket “JSoup” för att läsa filen, och tolka den som ett Room-objekt, som är vår klass som innehåller information om en bana.

6.2.2 Entity



Figur 1: UML-diagram över Entity och alla dess underklasser. Här syns det att varje ny klass är en mer specifik implementering än sin “förälderklass”.

Entities är en abstrakt klass som fungerar som huvudklassen för alla andra klasser som ärver från den. Entities-klassen innehåller de fält och metoder som behövs för entiteter som kan röra sig eller inte. Varje entitet har en "coord" som bestämmer deras position i spelet och representeras av en Point2D.Float. En viktig egenskap för varje entitet är deras hitbox, som är ett område som beskriver entitetens utbredning i spelet och representeras av en Rectangle2D.Float. Entity-objektet hanterar också hur entiteter ska tas bort. Varje entitet har ett boolean-fält, isGarbage, som talar om för Game-objektet om entiteten ska tas bort eller inte. När detta fält sätts till true av olika skäl, kan Game:s tick-metod iterera genom listan "Movable" och ta bort de entiteter som är "garbage", samtidigt som den tar hand om "loot" som hjärtan som spelaren får när den dödar fiender.

Nästa klass som ärver från Entity är Movable. Movable är också en abstrakt klass som bygger på Entity och hanterar entiteter som kan röra sig. Vi använder två floats, velX och velY, för att hantera entitetens hastighet i x- och y-led. Metoden "nudge" hanterar huvudsakligen all rörelse genom att använda setlocation och skicka nuvarande x- och y-värden samt dx och dy, som motsvarar velX och velY. Sedan uppdateras entitetens koordinater och hitboxen anpassas till de nya koordinaterna. Movable har också en tick-metod som helt enkelt flyttar objektet med "nudge"-metoden.

Nästa klass som ärver från Movable är "Character", som också är en abstrakt klass och fungerar som den slutliga abstrakta klassen för entiteter. Här finns fält och metoder som är mer generella, till exempel vilken bufferedImage som ska användas beroende på typen av karaktär och vilket håll den tittar åt. Det speciella här är att vi skriver över tick-metoden. Denna tick-metod ändrar bufferedImages om det behövs och kontrollerar om ticksInvincible eller ticksAttackCooldown behöver minskas. Förutom det innehåller Character-klassen själva ramverket som spelare och fiender kommer att ärva. Målet med alla dessa abstrakta klasser var att uppnå en viss nivå av abstraktion och polymorfism. De klasser som ärver från Character har dock sina egna unika metoder. En intressant klass är "Enemies", som också är en abstrakt klass som ärver från Character. Enemies har metoder som är unika för fiender, till exempel deras storeSpriteFrames som använder delbilder och en förskjutning (offset), eller deras checkIfPlayerIsInFront-metod.

6.2.3 Weapon

Weapon ärver Movable och är lik sin superklass på många sätt, såsom att den är abstrakt, men har till exempel ett nytt fält “lifeSpan”. En vapens lifeSpan säger hur länge den kommer att existera innan den markeras för borttagning. Ett annat fält som varje Weapon har är “owner”, vilket kommer att peka på ägaren till vapnet, och därmed göra det enkelt att veta vem som inte får ta skada av vapnet, samt vem som ska få poäng för att det har dödats någonting. Weapon har dessutom ett fält “sprite” av typen BufferedImage som bestämmer vilken bild som vapnet ska använda vid utritning.

Det finns två konkreta underklasser till Weapon: Projectile och Sword. Namnen är ganska självförklarande, ett objekt med typen Projectile har beteenden för att flyga rakt i den riktning som ägaren tittade när den attackerade, och ge skada till det som den kolliderade med. Ett Sword-objekt kommer att “följa” sin ägare under hela sin livstid, för att ge illusionen att ägaren håller i svärdet. Sword-objektet kommer att skada allting som kommer i kontakt med den under sin livstid.

7. Användarmanual

När ni har implementerat ett program krävs det också en manual som förklarar hur programmet fungerar. Ni ska beskriva programmet tillräckligt mycket för att en labbhandledare själv ska kunna *starta det, testa det och förstå hur det används*.

Inkludera flera (**minst 3**) **skärmdumpar** som visar hur programmet ser ut! Dessa ska vara ”inline” i detta dokument, inte i separata filer. Sikta på att visa de relevanta delarna av programmet för någon som *inte* startar det själv, utan bara läser manualen!

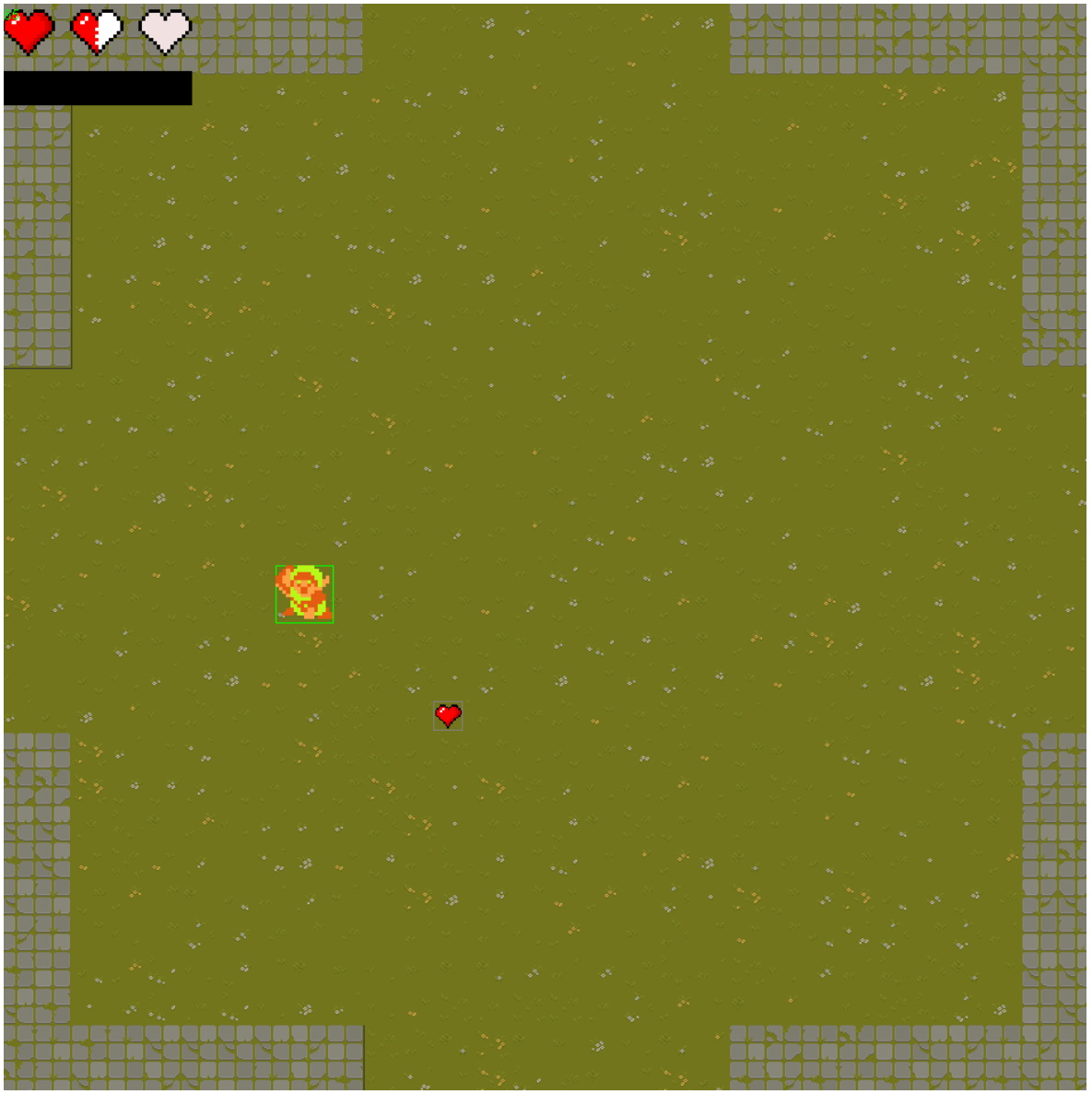
(Glöm inte att ta bort våra instruktioner, och exportera till PDF-format med korrekt namn enligt websidorna, innan ni skickar in!)

Spelet är inspirerat av en procedurellt genererad roguelike, där spelaren utforskar nya rum som är fyllda med slumpmässiga fiender.

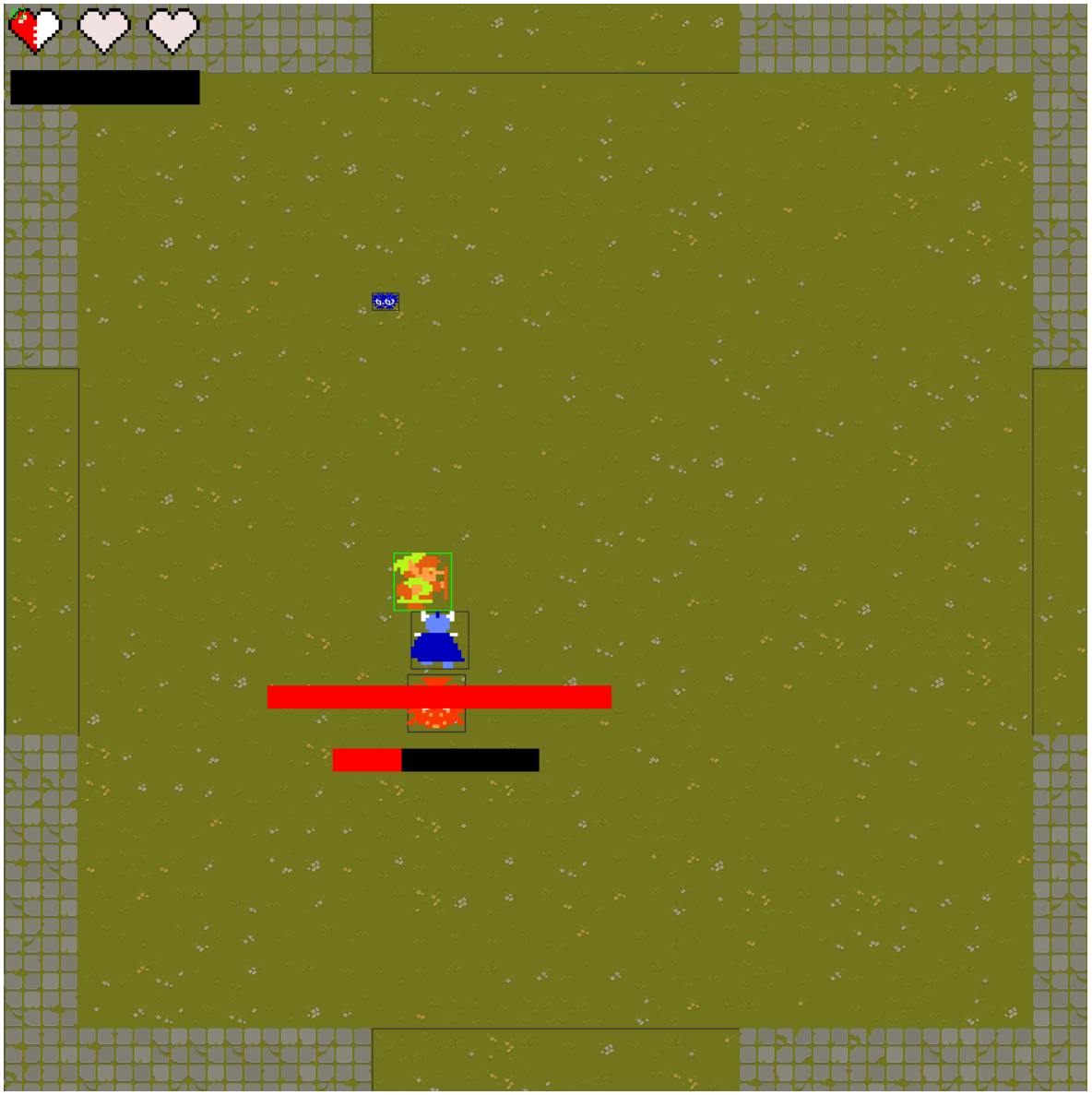
I spelet har man två sätt att attackera: det första är med ett svärd som används genom att trycka på mellanslag, och det andra är att skjuta projektiler med knappen Z. För att röra sig använder man piltangenterna på tangentbordet.

När man besegrar fiender blir man starkare genom att gå upp i nivå och erhålla "Decrees", vilka ger olika bonusar som till exempel större projektiler eller mer hälsa med mera. För tillfället erhåller man dessa "Decrees" genom att trycka på F2, där man får välja mellan två stycken. Spelupplevelsen går ut på att starta ett rum, besegra fiender och bli starkare för varje gång man spelar.

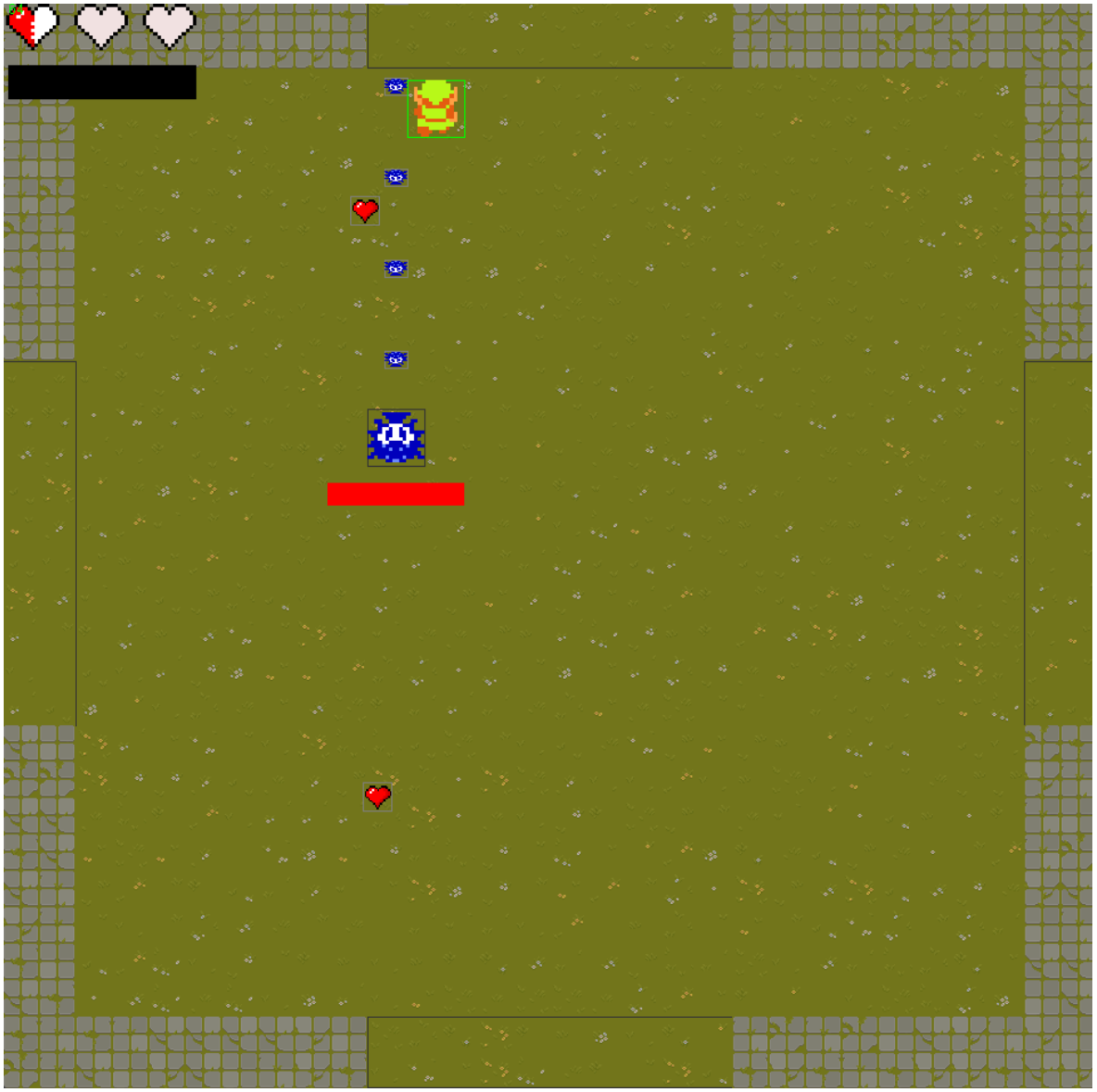
Man måste besegra alla fiender i rummet för att grindarna i alla fyra väderstreck ska öppnas. När man lämnar rummet så kommer man till ett nytt rum med nya fiender som också måste dödas för att ta sig vidare.



Figur 1: Spelaren har precis rensat hela rummet från fienden, och kan därmed lämna.



Figur 2: Spelaren vidrör, och tar därmed skada från fiendetypen “Knight”.



Figur 3: Spelaren flyr från fiendetypen “Sentry”:s projektiler.