Projektbeskrivning

<RPG 2D-Spel>

2013-09-12

Projektmedlemmar:
Dennis Ljung <denlj069@student.liu.se>

Handledare:
Mariusz Wzorek <mariusz.wzorek@liu.se >

Planering

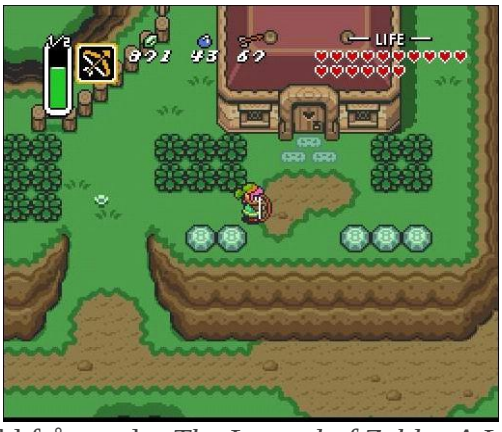
1. Introduktion till projektet

Detta projekt härstammar från slutuppgiften i kursen TDDC69 Objektorienterad programmering och Java. Uppgiften går ut på att implementera ett helt valfritt objektorienterat program i programspråket Java.

2. Ytterligare bakgrundsinformation

Programmet ska bli ett grafiskt spel i 2D-miljö med ett så kallat "top-down" perspektiv. Det menas att spelaren och området runt denne ses från ovan. Ett exempel på ett sådant spel är *The Legend of Zelda: A Link to the Past* som är en insperationskälla för projektet. I figur 1 ses en bild från det spelet.

Mer specifikt ska spelstilen påminna om ett RPG (Role-playing game). Typiskt för den typen av spel är rollspelselement som skräddarsydda karaktärer och utrustning samt mål/uppdrag som ger erfarenhetspoäng till spelaren.



Figur 1: En bild från spelet *The Legend of Zelda: A Link to the Past*

3. Milstolpar

#	Beskrivning
1	Kollisionshanteringen i miljön ska bestå av en osynlig kollisionskarta. En grafisk karta ska vara synlig för spelaren.
2	Allt i miljön ska byggas utifrån en textfil dvs kollisionskartan och vart fiender samt föremål befinner sig.
3	Spelaren ska kunna röra sig i en miljö som sträcker sig utanför det synliga fönstret. Fönstret ska kunna röra sig med spelaren när denne rör sig runt på kartan.
4	Spelaren ska kunna påverka miljön, ha sönder objekt (t ex något som blockerar vägen) plocka upp föremål och utöva skada på fiender.
5	Spelaren ska ha ett eget förråd som föremål kan förvaras i. Det ska gå att se vad som finns där, möjligen kunna välja föremål och göra något med dem.
6	Ett uppdrag ska kunna ges spelaren, kan vara att hämta ett visst föremål och returnera detta. När uppdraget är slutfört ska spelaren belönas med erfarenhetspoäng, antingen uttryckt i siffor eller grafiskt i en tabell.

Figur 2: Milstolpar (planering)

3. Övriga implementationsförberedelser

Det ska finnas en klass som agerar spelets motor. Här ska allt i spelet uppdateras och ritas ut. En uppdateringsmetod och ritametod ska finnas i alla spelelement som sedan körs från spelmotorklassen.

Skapandet av kollisionsmappen för miljön ska ske genom inläsning av en textfil tecken för tecken i en klass. Om tecknet var ett 'x' ska det till exempel indikera en vägg, sedan byggs förslagsvis kartan upp av en array av blockelement som kan integrera med spelaren.

4. Utveckling och samarbete

Jag är tänkte göra projektet ensam, alltså är samarbetsfrågor inte aktuella. Betygsmässigt siktar jag på en trea.

Slutinlämning

5. Implementationsbeskrivning

Nedanför följer implementationsbeskrivningen av programmet. Det innefattar hur programmet är strukturerat och hur de olika funktionaliteterna har blivit implementerade.

5.1. Milstolpar

I planeringsfasen bestämdes milstolpar för tänkt funktionalitet i programmet, se figur 2. Dessa milstolpar har i mer eller mindre grad blivit implementerad, se i figur 3 nedan.

#	Beskrivning
1	Kollisionshanteringen har blivit implementerad som en karta. Den är även den grafiskt synliga kartan för spelaren.
2	Allt i världen byggs utifrån en separat textfil.
3	När spelaren rör sig utanför fönstret flyttas kartan (och spelaren) så att den nya delen av kartan visas.
4	Spelaren kan plocka upp föremål och skada fiender men inte ha sönder objekt.
5	Spelaren har ett förråd som de upplockade föremålen visas i. Det går dock inte att göra något med föremålen.
6	Uppdrag (quests) finns i spelet. Uppdragen kombineras ihop av en eller flera delmål. När uppdraget är slutfört belönas spelaren med pengar och erfarenhetspoäng som visas grafiskt.

Figur 3: Milstolpar (Slutinlämning)

5.2. Dokumentation för programkod

Programmets kodstruktur är uppdelad i fyra paket: *Game, World, Player och Quest.*

5.2.1 Paketet Game

I paketet *Game* ligger främst kod för startandet och uppdateringen av spelet. Paketet består av två klasser: *Game* och *GamePanel.* Det är i klassen *Game* som programmets main-metod befinner sig. I klassens konstruktur skapas ett programfönster och en enkel meny till denna samt objektet *GamePanel* som kan ses som spelets huvudmotor. *GamePanel* har dessa huvudsakliga funktioner:

- Skapa objekten:

  - World : Världen där spelaren rör sig runt i. Innehåller kollisionskartan, fiender m.m.

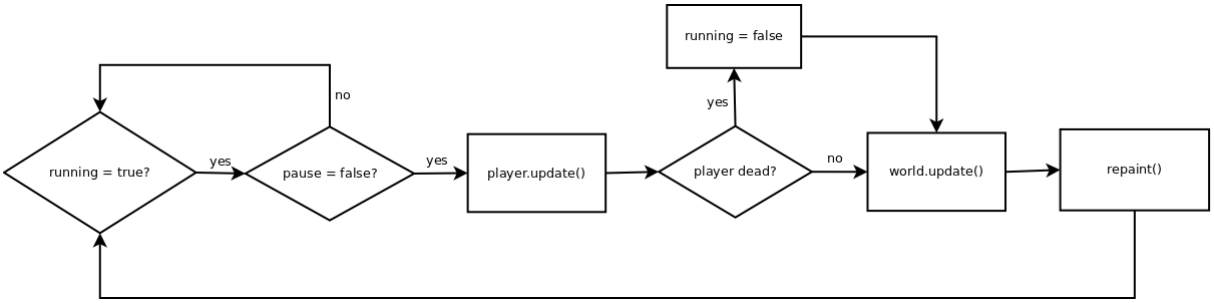
- Player : Spelaren.

- QuestSystem : Skapandet/hantering av uppdrag

- HUD : Statuspanelen som visar spelarens hälsa, erfarenhetspoäng, uppdrag m.m.

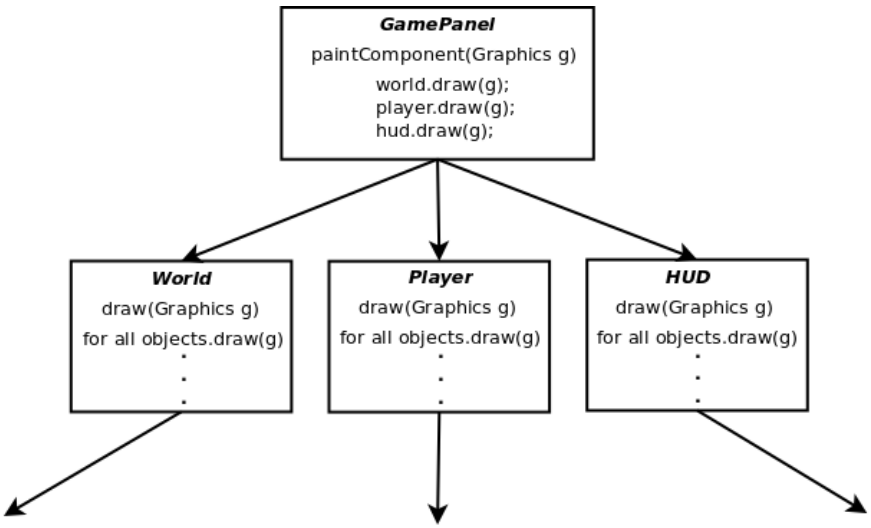
- Uppdatera objekten Player och World.
- Hantera indata från tangentbordet.

GamePanel implementerar gränssnittet *Runnable* och metoden *run()* för att kunna köra trådat. I figur 4 ses flödesschema för huvudtråden.



Figur 4: Flödesschema för huvudtråden

Så länge som spelaren inte är död kommer huvudtråden fortsätta köra. Sist kallas metoden *repaint().* Eftersom *GamePanel* ärver utritningsfunktionalitet från *JPanel* kan *repaint()* användas och den kallar i sin tur på utritningsmekanismen genom metoden *paintComponent(Graphics g).* Det är i *paintComponent(Graphics g)* som all kod för utritning befinner sig. Det som sker i *GamePanel* är att grafikobjektet *g* skickas vidare till objekten World, Player och HUDs utritningsmetoder som kallas *draw.* Dessa objekt kallar i sin tur på deras skapade objekts *draw*-metoder, se figur 5.



Figur 5: Utritningsmetoder

För hantering av indata från tangentbordet används gränsnittet *ActionListener* och den abstrakta klassen *KeyAdapter.* Med hjälp av metoden *keyPressed(KeyEvent e)* och *e.getKeyCode()* kan olika tangentbordstryck fångas. För ett visst knapptryck kallas olika

funktioner: För vänster-, höger- , upp- och nedpil sätts objekten Player och World's riktning med *player/world.setXDirection(int dir)* och *player/world.setYDirection(int dir).* Det gör att spelaren och världen kommer röra sig i någon riktning. Andra knapptryck gör andra saker, till exempel får spelaren att röra sig snabbare eller visa spelarens förråd och nuvarande uppdrag.

5.2.2 Paketet World

I paketet *World* ligger all kod som tillhör spelvärlden, dvs allt som spelaren kan interagera med. Paketet består av sex klasser: *World, Item, CollectibleItem, Coin, Enemy och Friend.*

I klassen *World* skapas kartan och allt världen ska innehålla. Denna information fås från en textfil, se figur 6.

Figur 6: Textfil för spelvärldens uppbyggnad.

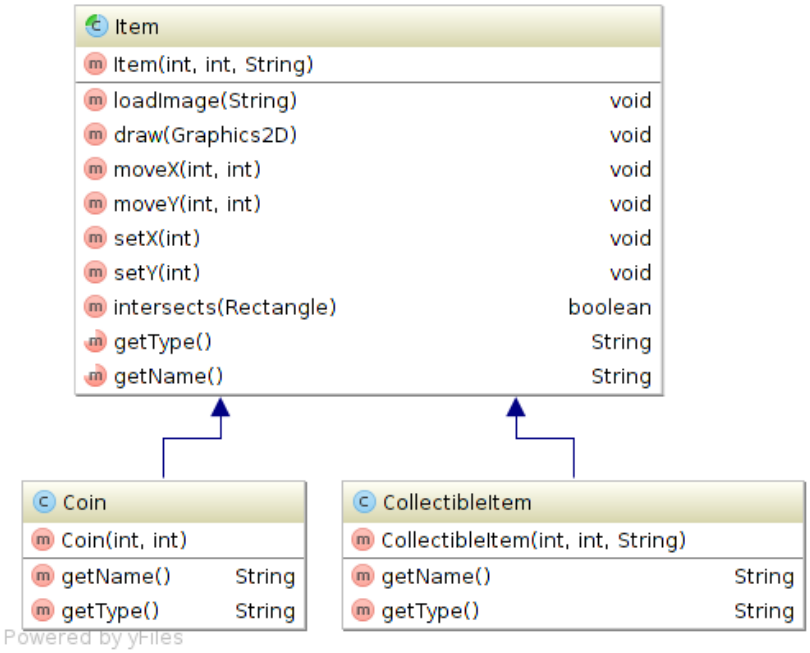
För varje inläst tecken skapas ett *Rectangle*-objekt som läggs i listan *mapCollision[]* och får värdet *true* eller *false* i listan *mapSolid[]* beroende på om spelaren ska kunna röra sig där eller inte. Andra objekt kan också skapas beroende på textsymbol. Alla objekt är representerade av *Rectangle*-objekt. Nedan följer teckenkodningen:

- 'x' : Här kan spelaren inte röra sig utan att kollidera.
- '-' : Här kan spelaren röra sig.
- 'c' : Ett *Coin*-objekt.
- ' e' : Ett *Enemy*-objekt.
- 'w' : Ett *Friend*-objekt av typen *Walk.*
- 'i' : Ett *Friend*-objekt av typen *Idle.*
- 's' : Ett *Friend*-objekt av typen *Stand.*
- 'm' : Ett *CollectibleItem*-objekt.

- 'R' : Används för att få kartans bredd.
- 'E' : Används för att veta när textfilen är slut.

Alla dessa objekts uppdaterings- och utritningsmetoder kallas på i metoderna *update() och draw()* i *World.*

Klassen *Item* är abstrakt och används av subklasserna *Coin* och *CollectibleItem.* Nedan följer ett UML-diagram för dessa, se figur 7.



Figur 7 : UML-diagram för *Item, Coin* och *CollectibleItem*

Klasserna *Enemy* och *Friend* är simpla implementationer av agenter.

5.2.3 Paketet Player

I paketet *Player* ligger kod som är relaterad till spelaren. Paketet består av fyra klasser: *Player, Inventory, HUD och PlayerListener.*

I klassen *Player* skapas objektet för spelaren och det är här den huvudsakliga funktionaliteten ligger. Konstruktorn tar ett argument vilket är ett *World*-objekt. *World*-objektet behövs för att få tillgång till kollisionskartan, föremål, vänner och fiender. Då världen är uppbyggd av *Rectangle*-objekt är även spelaren ett sådant.

5.3. Användning av fritt material

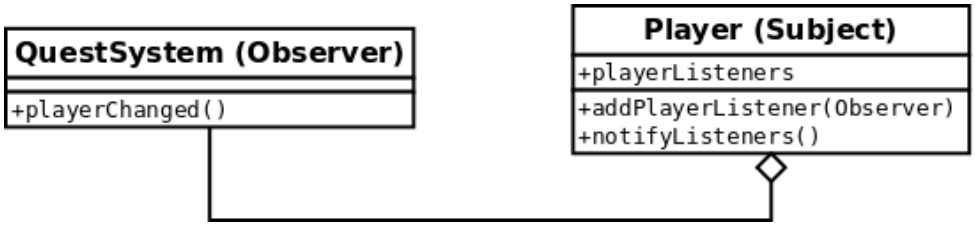
I projektet har det inte använts något bibliotek utöver det som redan finns inkluderat i Java 7.

5.4. Användning av designmönster

- Observer

Detta designmönster används för att uppdatera *QuestSystem* när spelaren har gjort något som kan påverka en pågående *Quest*, till exempel då ett föremål har plockats upp eller en fiende har dödats. Genom att använda detta designmönster blir det effektivare kod då *QuestSystem* endast blir notifierad när det har hänt något av nytta.

Ett UML-diagram för implementationen ses i figur ? .



Figur ? : UML-diagram för observer-mönstret

Klassen *QuestSystem* lägger till sig själv i listan *playerListeners.* Klassen *Player* är det observerade subjektet och den notifierar klassen *QuestSystem* med metoden *notifyListeners()* då spelaren har dödat en fiende, plockat upp ett föremål eller pratat med en vän. Metoden *notifyListeners()* kallar på objektet i *playerListeners* metod *playerChanged().* I metoden *playerChanged()* kallas i sin tur metoden *update().*

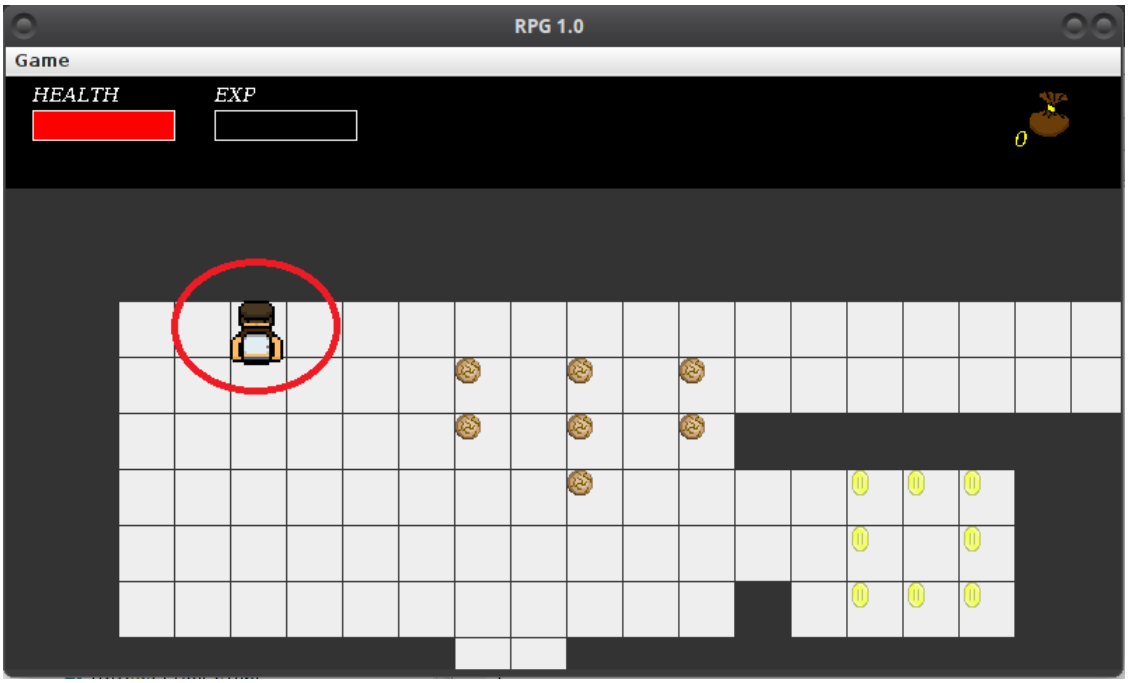
5.5. Användning av objektorientering

- Polymorfism

Med den abstrakta klassen *Item* kan subklasserna *CollectibleItem* och *Coin* hanteras som om de vore instanser av superklassen *Item.* Detta har lett till effektivare kod då de kan läggas i samma lista och kallas med samma metoder.

6. Användarmanual

När programmet har kompilerats startas det genom att köra filen *Game.java.* Användaren möts då av vyn nedanför, se figur ?. Spelaren som användaren kan styra är inringad i rött.



6.1. Kontroller

För att styra spelaren och interagera med omvärlden används följande tangentbordskommandon:

- Piltangenterna : Styr spelaren upp, ned, höger eller vänster
- c : Attackera
- x : Plocka upp föremål/starta *quests*
- shift : Rör sig fortare.
- i : Visa spelarens *inventory.*
- q : Visa aktiva och avklarade *quests.*

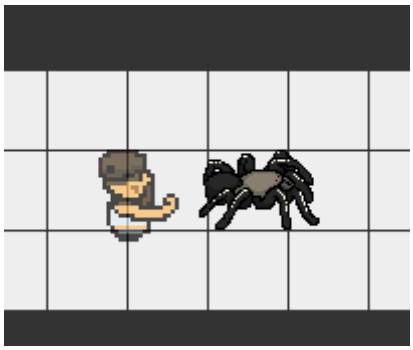
6.2. HUD

Överst i spelfönstret visas spelarens livsnivå (HEALTH) och erfarenhetspoäng (EXP). Överst i högra hörnet visas hur mycket pengar (*Coins)* spelaren har. Se figur ?.

6.3. Fiender

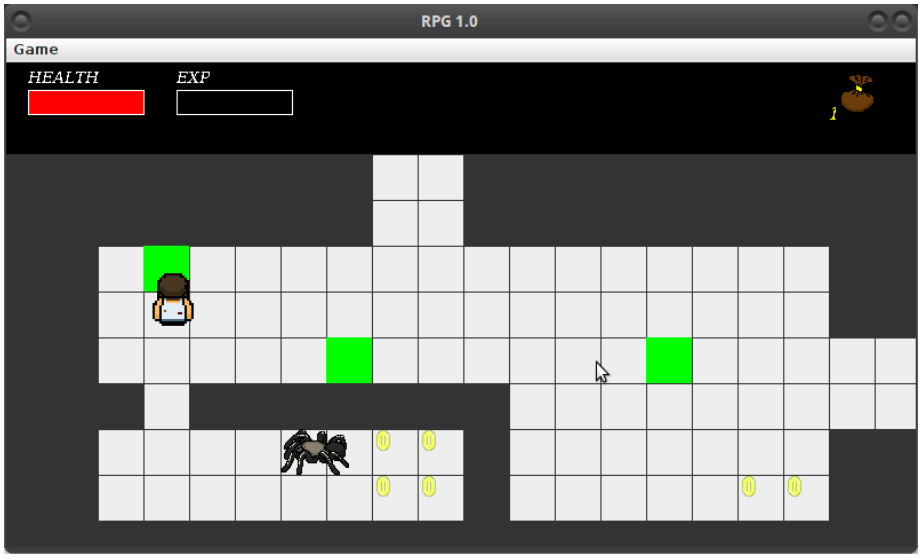
Fienderna i spelet kan skada spelaren och om livsnivån (HEALTH) tar slut är spelaren död och spelet är över. Spelaren kan skada och döda fienderna genom att attackera dem. Om en

fiende dödas belönas spelaren med erfarenhetspoäng (EXP) . I figur ? syns en bild på en fiende (till höger) som blir attackerad av spelaren.



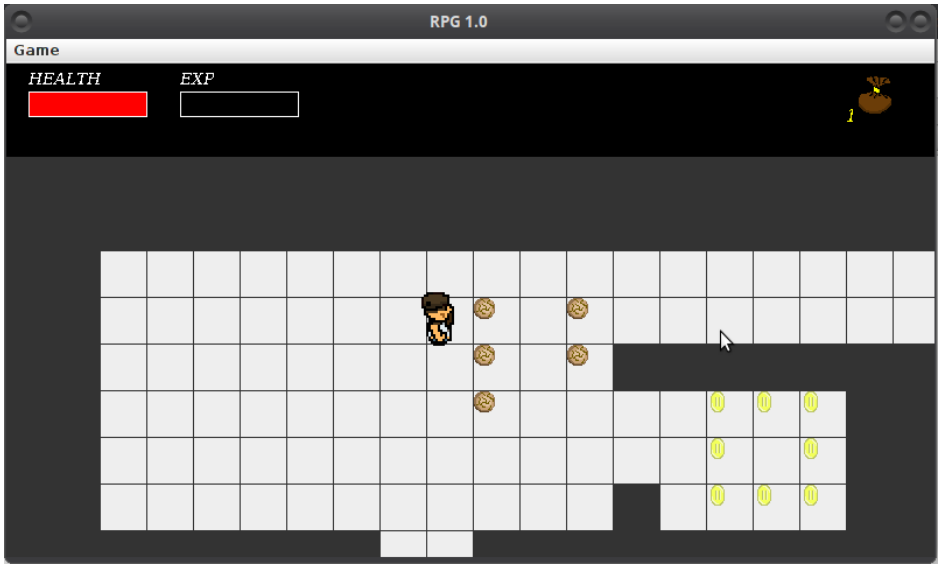
6.3. Quests

Spelaren kan starta uppdrag (*quests)* genom att gå fram till en vän (gröna lådor) och trycka på 'x'. Det är bara vissa vänner som har *quests.* När en *quest* är avklarad går spelaren tillbaka till vännen som gav denna och får då en belöning i form av pengar (*coins)* och erfarenhetspoäng (EXP). I bilderna nedan visas ett exempel på hur en *quest* utförs.

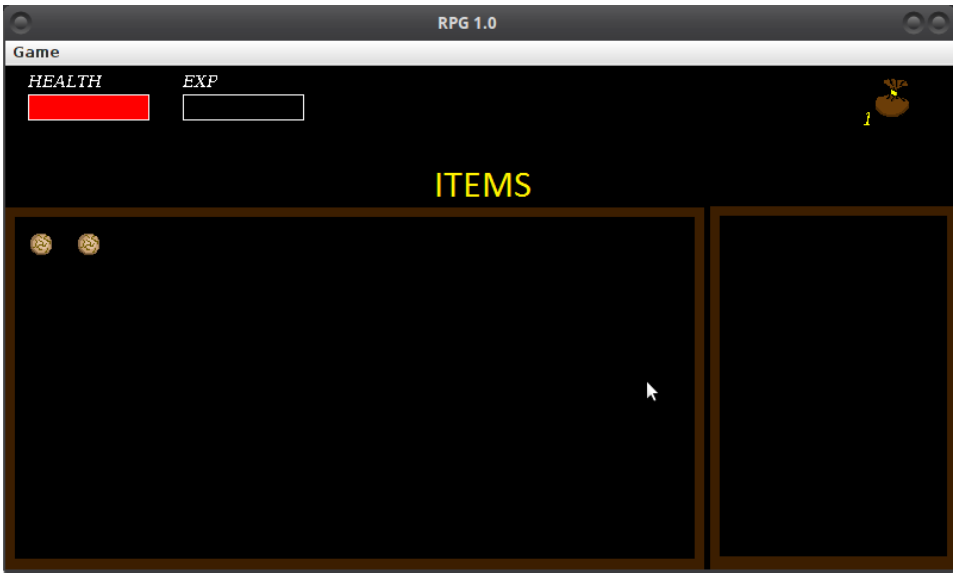


Figur ? : En quest startas...

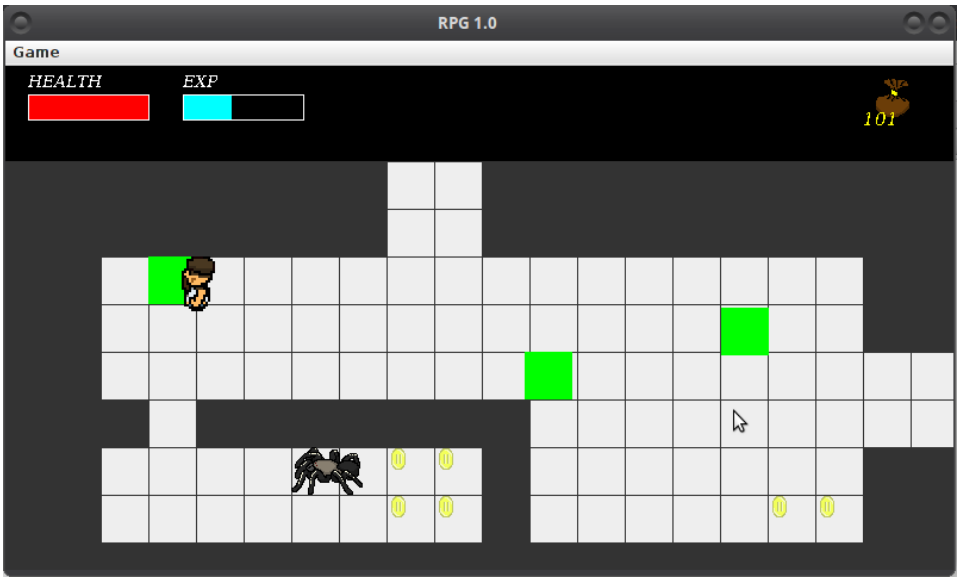
Figur ? : ...den aktiva questen kan nu ses...



Figur ? : ...utför questen och plockar upp 2x spider eggs...



Figur ? : ...föremålen ligger i *inventory*...



Figur ? : ...går tillbaka och questen är avklarad!

6.4. Spelmeny

I spelets meny (uppe i vänstra hörnet under *Game)* kan spelet börjas om med *New Game* och avslutas med *Exit.* Spelet kan även avslutas genom att klicka på krysset uppe i högra hörnet.

7. Betygsambitioner

Betygsmässigt siktar jag på en trea.

8. Utvärdering och erfarenheter

Den färdiga produkten är jag stolt över, dock kodmässigt är jag inte lika stolt. Jag borde ha lagt mer tid på att planera och strukturera koden, tänka över designmönster och designbeslut. Kontinuerlig kommentering av kod är något jag även har lärt mig är viktigt, dels för min egen förståelse av koden, dels för att inte behöva stressa ihop allt på slutet.