TDDD78 / TDDE30 / 729A85 – Objektorienterad programmering och Java

Projektbeskrivning

The legend of Lelda:
Adventure of Zink

2022-03-22

Projektmedlemmar:

Henry Andersson henan555@student.liu.se

Hannah Bahrehman hanba478@student.liu.se

Handledare:

Philip Rettig phire844@liu.se

Innehåll

Projektplan

1. Introduktion till projektet

Detta är något som ni aldrig tidigare sett. Ett action-äventyrsspel med kluriga pussel, skrämmande monster och ett mer skrämmande bossmonster som hindrar dig från ditt mål. Du spelar riddaren som fått uppdraget att rädda prinsessan Lelda från den onda draken i ondskans grotta. För att kunna besegra draken måste du hitta de vapen som ligger gömda i grottan, bland dom finns också det legendariska mästersvärdet som har kraften att dräpa alla varelser i några få slag.

Projektet är inspirerat av spelet *"The legend of Zelda"* som släpptes år 1986 av Nintendo. Spelet kunde bara spelas på ett licenserad spelsystem från Nintendo vid namn "*Nintendo Entertainment System",* Oftast förkortat till NES. Spelsystemet var driven av ett 8 Bit processor, som under 80-talet var det bästa som då fanns.
"The legend of Zelda" är ett Äventyrsspel där du kontrollerar spelaren från ett fågelperspektiv. Med ett svärd och andra vapen kommer du dräpa monster och rädda prinsessan.

2. Ytterligare bakgrundsinformation

Under projektet kommer vi använda oss av objektorienterad programmering. Objektorienterad programmering är en typ av programmeringsmetodik som använder varierande uppsättning objekt som interagerar med varandra.
En del av objektorienterad programmering är klasser. En klass är som en mall för hur en viss typ av objekt ska fungera och kunna användas, alltså vilken information den ska spara och vilka metoder som kan tillkallas av andra objekt.
Vilka delar av ett objekt som är åtkomligt från andra objekt kan bestämmas i klassen. Detta gör det möjligt för en klass bestämma gränssnittet.
En klass kan ärva av andra klasser. Klassen kommer ha funktionaliteten av klassen den ärver från som inte ligger "gömd" samt kan ha egna metoder och info.
Flera klasser som har samma överklass kan ha samma gränssnitt men olika funktionalitet. Det kallas polymorfism och fungerar på så sätt där gränssnittet bestäms av en överklass men metodiken bestäms av en underklass.

3. Milstolpar

#	Beskrivning
1	Kan visa en ruta på skärmen (spelplanen).
2	Laddat in bilder (sprites). Kan visa bilder på skärmen.
3	Få en bild att röra på sig. Bilden ska röra på sig med instruktioner från tangentbordet.
4	Kollision kan detekteras.
5	Kan byta rum. När en bild/gubbe är på en specifik plats (dörr eller liknande öppning) kommer ett annat rum visas.
6	Implementerat fiender och enkel rörlighet.
7	Implementerat enkelt hälsa-system för fiender och huvudkaraktären.
8	Skapat grundläggande GUI.
9	Karaktärer kan skada och bli skadad (utökat kollision). Huvudkaraktären skadas genom att gå in i fienden.
10	Utökat hälsa-systemet.
11	Utökat spelplanen. Det finns väggar, golv och dörrar.
12	Implementerat inventory.
13	Implementerat svärd
14	Implementerat fler vapen.
15	Kan skada fienden med vapen.
16	Skapa dörrar med nycklar
17	Skapa boss
18	Skapa pussel / utöka spelplanen
19	Skapa slutskärm/slutskärm
20	Spelet är klart :^).

4. Övriga implementationsförberedelser

Till början så spelade vi Zelda och kollade på game play videos på Zelda till att veta vad vi ville ha i vårat spel samt vad som skulle vara realistisk till att implementera till vårat projekt.
Till en början förberedde vi oss genom att skriva ned alla saker vi kunde tänka oss ha med i spelet. Monster, huvudkaraktären, svärd, väggar, dörrar, nycklar mm. Sedan sorterade vi in sakerna i flera grupper som liknar varandra till exempel vapen, personer, objekt. Därifrån blev det lätt att se vilka klasser som behövdes och därmed vilka som kan ärva från samma klass eller interface. Vi började också skapa ett UML-diagram för hela projektet. UML diagrammet blev allt för dålig för att vi ska kunna få någon typ av användning av det under projektet, istället så kommer vi försöka följa våra milstolpar och se vart det kommer leda oss till.

5. Utveckling och samarbete

Planen är att vi skulle främst arbeta med projektet vecka 11 och 12. Förberedelser som till exempel vilket typ av projekt det är, vad vi vill att projektet ska innehålla och hur grundläggande struktur ska se ut ska ske innan.

Vi båda är gemensamma om att sikta mot betyg 5. För att kunna uppnå detta måste vi se till att vi hela tiden förstår vad den andra personen har gjort och hur det fungerar, så att vi eventuellt kan använda varandras kod på ett bra sätt.

Projektrapport

6. Implementationsbeskrivning

6.1. Milstolpar

1. Kan visa en ruta på skärmen.

Helt genomförd. Sker i StartGame-mainfunktionen.

2. Laddat in bilder (sprites). Kan visa bilder på skärmen.

Helt genomförd. Sprites laddas in i AbstractEntity-klasser och AbstractObject-klasser. De ritas ut i drawfunctionen som finns i dem klasserna som tillkallas i ZinkPanels paintComponent.

3. Få en bild att röra på sig. Bilden ska röra på sig med instruktioner från tangentbordet.

Helt genomförd. Klassen KeyHandler ärver från KeyListener så funktionerna keypressed och keyreleased tillkallas när en tangent trycks och släpps.

4. Kollision kan detekteras.

Helt genomförd. Kollision sker mellan olika entitys, mellan entity och gameobject samt mellan entity och tiles som har collision. Vad som sker vid kollision är olika för varje klass beroende på vad som kolliderar och andra villkor. Alla entity och gameobject har en funktion hasCollided.

5. Kan byta rum. När en bild/gubbe är på en specifik plats (dörr eller liknande öppning) kommer ett annat rum visas.

Helt genomförd. I RoomManager laddas en textfil in med hela kartan som i ZinkPanel visas som flera rum. I funktionen moveScreen flyttas “kameran” beroende på spelarens position på kartan.

6. Implementerat fiender och enkel rörlighet.

Helt genomförd. Fienden rör sig genom att slumpmässigt välja mellan fyra olika håll och sedan röra sig det hållet i 2 sekunder.

7. Implementerat enkelt hälsa-system för fiender och huvudkaraktären.

Helt genomförd. När kollision sker mellan en fiende och playern tar playern skada.

8. Skapat grundläggande GUI.

Helt genomförd. Hanteras i WindowManager.

9. Karaktärer kan skada och bli skadad (utökat kollision). Huvudkaraktären skadas genom att gå in i fienden.

Helt genomförd. Fienden tar skada när kollision mellan fiende och playerns vapen sker.

10. Utökat hälsa-systemet.

Helt genomförd. Det finns ett gameobject Heart där när playern kolliderar med ett Heart kommer playern få tillbaka ett liv

11. Utökat spelplanen. Det finns väggar, golv och dörrar.

Helt genomförd. Vägg och golv är Tiles som skapas i RoomManager. Dörrar är en typ av gameobject som öppnas om playern har ett Key-gameobject. Annars beter dörren sig som en vägg.

12. Implementerat inventory.

Helt genomförd. Det finns ett inventory-system skapat i WindowManager där man kan se alla gameobjects man har och välja sitt vapen.

13. Implementerat svärd

Helt genomförd. PlayerSword är en typ av gameobject som läggs till på skärmen när playern attackerar

14. Implementerat fler vapen.

Helt genomförd. Det finns två olika svärd och en pilbåge.

15. Kan skada fienden med vapen.

Helt genomförd. Man kan svinga sitt svärd på dem eller skjuta en pil men pilbågen från en bit bort.

16. Skapa dörrar med nycklar

Helt genomförd. Door och Key är två gameobjects.

17. Skapa boss

Helt genomförd. EnemyDragon är en typ av Enemy som kan skjuta eldklot. Playern dör vid kollision med enemydragon.

18. Skapa pussel / utöka spelplanen

Ej genomförd.

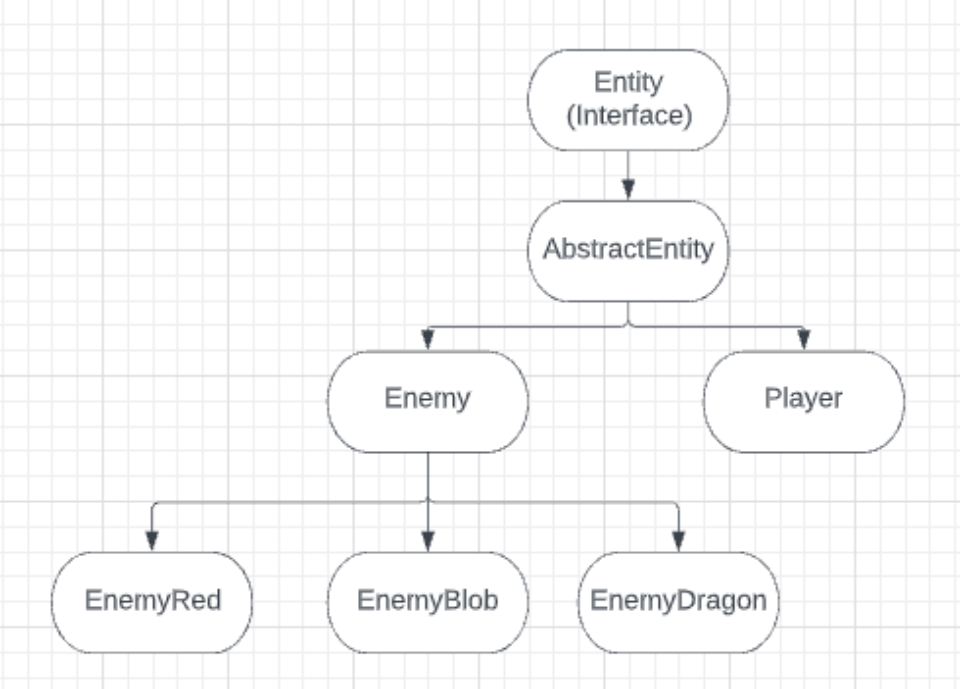
19. Skapa startskärm/slutskärm

Delvis genomförd. Det finns en startskärm skapat i WindowManager. “Slutskärmen” är bara ett meddelande som syns på skärmen när playern kolliderar med prinsessan.

20. Spelet är klart :^).

6.2. Dokumentation för programstruktur, med UML-diagram

6.2.1 Klasser och struktur



UML-diagram 1

UML-diagram 1 visar hur entity-klasserna hänger ihop. Entity i det här projektet är alla saker på skärmen som “lever”, allt som rör på sig samt behöver någon typ av högre funktion än att bara stå still. Detta beskriver alla fiender och spelaren. Det finns ett gränssnitt med metoder “attack” och “draw” och fler metoder som alla entitys använder. AbstractEntity är en abstrakt klass som implementerar Entity. Den definierar det flesta av metoderna i Entity. Alla entitys är ganska lika så det mesta av funktionaliteten för dom ligger i AbstractEntity. De funktioner som finns i AbstractEntity används av alla entitys, De funktionerna är så gemensamma så det gör att alla klasser som härstammar från AbstactEntity blir väldigt liten. Det som finns i dom som frånskiljer dem är de bilder som laddas in.
Vi spenderad mycket av våran tid på att få en bra struktur för oss om underlättade implementation

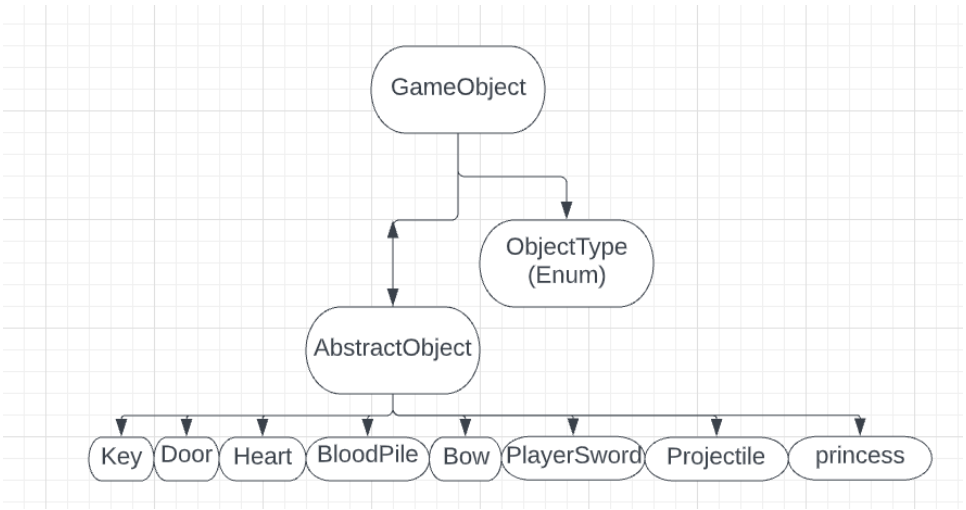
av fiender. Tyvärr har vi väldigt få eftersom vi spenderade mer tid på att programmera än att rita. Om vi skulle vilja lägga till fler fiender så skulle det inte ta någon tid alls om vi hade alla bilder tillgängliga.
I AbstractEntity så finns det funktioner gällande kollision, När kollision sker mellan en entity och en Tile så bakar man enityn lika mycket som den går i tilen.
I Entity så finns det en funktion gällande kollision med andra Entitys vid namn “takeDamage”. Som tar bara in en int som argument för att kolla hur mycket skada den entityn ska få.
I AbstractEntity klassen så tar den hand om allt gällande sprites, Den ändrar sprites beroende på hur en entity går, den börjar blinka när en entity tar skada, den ger en delay gällande hur ofta en entity kan attackera, den ritar alla entitys på skärmen.
Den ändrar vilken sprite som vissas gällande input. Den gör det genom att man sickar in en EntetyInput (en Enum) som senare rör entity i den riktningen samt ändrar bilden till rätt bild för att gå åt det hållet.

Klassen Player är entityklassen för den användarstyrda entityn, huvudkaraktären “Zink”. Spelaren styr Player genom KeyHandler (Beskriven i mer detalj senare i texten). Genom att få de olika inputs så kommer Player klassen läsa dessa och göra det som behövs. Så när man håller nere “W” så kommer den skicka in “UP” i funktionen definierad i AbstractEntity som senare gör att spelaren rör sig uppåt. Det som definieras i Player som inte görs i AbstractEntity är vad som händer när spelaren trycker på attack och Spelarens ryggsäck. När spelaren trycker på attack så kollar klassen om ett vapen är aktiv och om fördröjningen mellan attacker är tillräckligt liten, sen attackerar den med det vapen som är aktivt. Om det är ett svärd attackerar spelaren med svärdet, Om det är Pilbågen så kommer spelaren skjuta ut en pil.

Klassen Enemy är den abstract-klassen som de andra Fienderna i spelet ärver från. I Enemy så tar den hand den slumpmässiga rörelsen hos fienderna. Den gör det genom att ta fram en slumpmässig riktning och sen sicka in den i Move-funktionen som är definierad i Entity. I samma spår så slumpar klassen när en Enemy ska attackera. När Enemy attackerar så skjuter den ut en projektil. Att skjuta är definierad i AbstractEntity eftersom vi vill att alla Entitys ska ha möjligheten till att skjuta.

Den enda speciella fienden är draken, Eftersom vi villa att draken bara kunde röra sig upp och ner samt bara kunna skjuta år höger behövde vi överskriva dessa funktioner i EnemyDragon. Det var lätt att göra eftersom de funktionerna vi skrev var väldigt korta eftersom det ska bara uppfylla en sak.

De andra två fienderna överskriver bara vilka bilder de ska använda. De använder bara klassen Enemy, vilket är väldigt bra för om vi vill implementera fler fiender så behöver vi knappt skriva någon kod alls. Vilket var hela poängen med att skriva Enemy och AbstractEntity.



UML-diagram 2

Strukturen för GameObjects (UML-diagram 2) liknar den av Entity (UML-diagram 1). Det finns ett gränssnitt GameObject och en abstrakt klass AbstractObject som implementerar gränssnittet. I AbstractObject definieras metoden setImages där bilden för ett GameObject läses in och moreValues där beroende på vilken EntityInput man skickar in som parameter vrids bilden mot riktningen. Det finns även metoder så som draw och två getters.

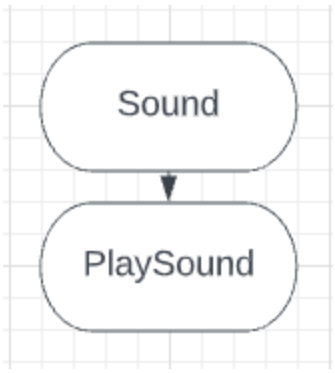
Det som främst skiljer mellan olika GameObjects är vad som sker vid kollision. När kollision sker mellan Player och något av objekten förutom BloodPile och Princess-objekten kommer objektet försvinna från skärmen. Om objektet var en Key, Bow eller PlayerSword kommer objektet hamna i playerns inventory-lista. Med kollision med Heart kommer playern få ett till liv, om inte playern redan har maxantal liv.

Det som händer vid kollision med en Projectile beror på vilken entity som kolliderar och vilken entity som skapade Projectilen. Om en Projectile skapad av en enemy kolliderar med playern tar playern skada och tvärtom om playern skapade Projectilen kommer enemyn ta skada. När en Projectile skapas skickas antingen PLAYER_BOW eller ENEMY_BOW in och sparas i fältet gameObject som finns i AbstractObject så att whenCollided-metoden kan kolla från vilken entity Projectilen skapades.

En BloodPile skapas när en Enemy dör och placeras på den Tile där Enemyn dog.

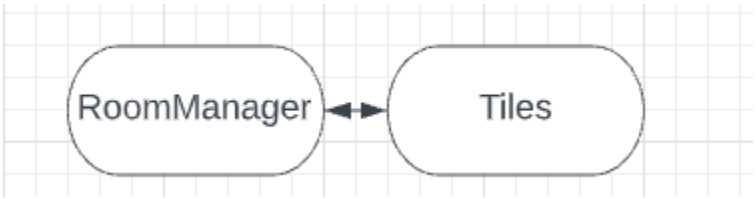
När en Player kolliderar med en Door sker olika saker beroende på om playern har någon Key i sitt inventory. Om playern inte har någon Key kommer Door agera som en vägg genom att putta tillbaka playern när den försöker gå genom. Om playern har minst en Key kommer Door försvinna när playern kolliderar med den och en Key kommer försvinna från playerns inventory.

Vid kollision med en Princess tillkallas metoden showWinScreen i WindowManager vilket gör att det visas ett meddelande på skärmen att man har räddat prinsessan. Spelet kommer också sluta uppdateras, alltså spelet tar slut när man kolliderar med prinsessan.



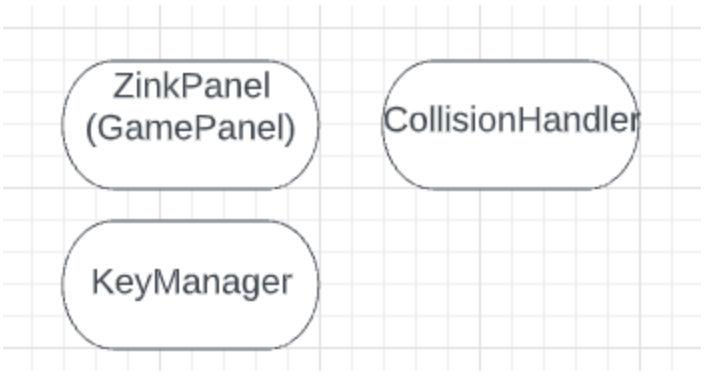
UML-diagram 3

I Sound klassen så laddar vi in alla ljud vi vill ha i en array med olika index. Genom att göra det så sparar vi ljuden med dessa index, så när vi vill spela ett ljud så tillkalar vi indexet.
Vi sedan skapar en klass PlaySound som ärver Sound. Dessa funktioner kan ligga i Sound men vi valde att skapa en till klass så att vi kan separera när vi ska spela ett ljud och ladda in ljud.
Nu efterhand kanske det skulle vara bättre att ha en Enum istället för nummer för att tillkalla vilka ljud vi vill spela så att det blir bättre att läsa koden.
Vi tar hand om Catch Fallthrough genom att de olika värden i array blir null, då dessa ljud aldrig spelas och spelet fortsätter samt andra ljuden spelas.



UML-diagram 4

RoomManager tar hand om att läsa in kartan från map.txt filen. Vi läser in data på kartan genom först definiera gränserna på våran spelyta i Zinkpanel sedan skapar vi en 2D array med som representerar våran karta. Vi fyller 2D arrayn med olika Tiles som vi har tidigare definierat med deras egenskaper som bild och om rutan har kollision. Varje nummer i våran map.txt representerar en tile, nu i efterhand så skulle det vara bättre om vi använde oss av Enum för att definiera vilka Tiles vi menar istället för 1,2,3 som vi har nu vilket kan bli svårläst. Men det dyker bara upp i RoomManager så man kan läsa det från sammanhanget.



UML-diagram 5

I keyhandler så ärver den “KeyAdapter”, en standardklass i Java. Hur våran KeyHandler funkar är att vi har en Map med alla våra tangenter som vi vill använda oss av. I mapen så är alla tangenter kopplad till false, sen när man håller nere en tangent blir den tangenten true tills man släpper den då den blir false igen.
Vi valde att ha de så eftersom då kan användaren hålla nere tangenter så gubben kan fortsätta gå tills användaren släpper tangenten, samt att man kan låtsas hålla nere tangenter med kod genom att ha en variabel som adderar varje x sekunder som simulerar att man håller nere tangenten.

CollisionHandler är klassen som ser till att kollision sker. Kollision mellan Entitys och object eller Entitys med andra Entitys var lätt att implementera. Det gjordes genom en inbyggd funktion i java som heter Intersects, som returnerar True om de är över varandra. Genom att iterera över alla object on entitys och kolla om något kolliderar, inte svårt alls 😛.
En av det svåraste med detta projekt var att kolla kollisionen mellan Entitys och Tiles. Vi löste det genom att ange varje hörn i våran rektangel till en koordinat och sedan kolla två av dessa koordinater beroende på den sista inputen av Entityn för att få reda på om kollision sker.
Vi kan inte använda Intersects som tidigare för att vi behöver veta vilken sida som kollisionen sker för att kunna putta tillbaka entityn åt rätt håll.

ZinkPanel är den klassen som kopplar ihop allt i spelet. I den finns det en Timer som tillkallar en metod tick som uppdaterar spelet och tillkallar metoden draw. Det är i ZinkPanel som alla viktiga tal finns såsom storleken på skärmen, storleken på en Tile mm. Metoden draw tillkallar på metoden draw som finns i RoomManager, AbstractObject, Enemy, Player och WindowManager i den ordningen.

6.2.2 Intressanta lösningar

Vissa delar av koden är vi extra stolta över. En av dessa delar är klassen CollisionHandler. Som beskrivet i avsnittet innan är vi stolta över hur kollision detekteras.

Metoden moveScreen i ZinkPanel löste problemet över hur vi skulle byta “rum”. Slutresultatet blev att det som ändras är vilken del av kartan som visas på skärmen.

6.2.3 Varningar i kodgranskningen

Kodanalysen visar ett problem med namnet ZinkPanel. Vi har valt att ignorera varningen då anledningen till att klassen heter just ZinkPanel är för att den entity användaren spelar som heter Zink och ZinkPanel är den centrala klassen som kopplar ihop alla klasser i detta projekt.

En **CatchFallthrough** problem som finns i RoomManager, AbstractObject och AbstractEntity är inte ett problem. Problemet som uppstår är när vi försöker läsa in en fil, vi hanterar problemet genom att om filen inte finns eller inte kan läsas så ersätter vi bilden med en grå ruta. Genom att göra det så kan man se att det finns ett problem med att läsa in felen samt så kommer inte spelet avslutas och fortsätta spela.

7. Användarmanual

Spelet börjar med att visa en startskärm. Här väljer du mellan de olika valen genom att använda tangenterna W och S och sedan bekräfta valet med ENTER-tangenten. Om du har valt START GAME kommer spelet börja direkt.

Du flyttar på din spelare med WASD-tangenterna. Objekten som ligger på marken kan tas upp genom att bara gå in o dem.

Du kan se vilka objekt du har genom att öppna din inventory med ENTER. Tryck samma tangent för att stänga inventory. För att döda fiender behöver du ha ett vapen. Om du har tagit upp ett eller flera vapen så kan du välja att det vapen du vill använda genom att öppna din inventory, markera vapnet och trycka SPACE. Vapen som är omringad av en gul kvadrat är det vapnet du använder just nu.

När du är riktad mot en fiende kan du trycka SPACE för att attackera den. Tänk på att om du håller i ett svärd måste du stå nära fienden.