Projektbeskrivning

Pac-Man

2021-03-1

Projektmedlemmar:

Nils Arnlund, nilar937@student.liu.se

Anton Bergman, antbe028@student.liu.se

Handledare:

Robin Andersson, roban591@student.liu.se

Innehåll

Projektplan

1. Introduktion till projektet

I kursen TDDE30 - Objektorienterad programmering och Java tänker vi utveckla ett Pac-Man spel som projekt. Pac-Man är ett av det mest kända spelen i världen. Pac-Man släpptes i Japan år 1980. Spelet är ett två dimensionellt labyrintspel där spelaren styr den ikoniska huvudkaraktären som samtidigt jagas av fyra stycken spöken. Målet är att spelaren ska samla så många poäng som möjligt. Poäng samlas igenom *enegizers*, *dots* och bonusföremål som finns i labyrinten. Det finns en *energizer* i varje hörn och gör det möjligt för Pac-Man att äta spöken för poäng. Ytterligare en metod för att samla poäng är *dots*. Varje enskilld *dot* ger poäng och när Pac-Man har ätit alla *dots* går spelaren vidare till nästa nivå. För varje ny nivå blir spökena snabbare och effekten av *enegizers* blir kortare. Figur 1 visar en bild från originalspelet och som kommer att användas som inspiration vid implementationen av vårt projekt.

(1) Bild från Pac-Man orginalet.

2. Ytterligare bakgrundsinformation

För mer bakgrundinformation om Pac-Man se följande wikipedia artikel:

- https://en.wikipedia.org/wiki/Pac-Man

För att prova en version av spelet se följande länk:

- https://www.google.com/logos/2010/pacman10-i.html

3. Milstolpar

För att underlätta implementationsfasen av projektet använder vi milstolpar. Projektet delas upp i en sekvens milstolpar, där varje milstolpe är en funktion. Varje ny milstolpe bygger vidare på föregående med ny funktionalitet. I detta tidiga stadie av projektet är det svårt att avgöra vad som är rimligt att hinna med. Därför kommer vi att ange fler milstolpar än vi kommer att ha tid att implementera. Listan av milstolpar ska fungera som vägledningen igenom projektet och inte som en kravlista.

#	**Beskrivning**
1	Det finns en data typ för en spelplan samt funktionalitet för att med en grafisk komponent visa en spelplan. En testklass kan skapa en spelplan och visa den grafiskt i ett fönster.
2	Implementation av Pac-Man. Se till att det går att skapa en instans och att denna ritas ut på spelplanen. Redan i detta stadie ska Pac-Man figuren ha en väldigt simpel animation.
3	Implementation av rörelse för Pac-Man. Se till att det går att röra sig i alla riktningar. Det sker på en tom spelplan.
4	Implementation av en labyrint med *dots* och den tunnel som finns i originalet. Detta gör det möjligt att testa kommande funktioner.
5	Implementation av spöken, även dessa med en lättare animation. En enklare artificiell intelligens och möjligheten att eliminera Pac-Man implementeras också.
6	Ett poängsystem implementeras. *Dots* och bonusföremål ger nu poäng och poängen visas.
7	Implementation av en timer för spelet. Denna kommer att användas på en rad olika ställen i spelet.
8	Implementation av *energizers* med alla dess funktioner. Hur spökena påverkas, hur spökena kommer tillbaka till liv och hur poäng delas ut.
9	Implementation av en mer kompetent artificiell intelligens för spökena. Här ska även spökena släppas ut i rätt ordning och tillfälle.
10	Implementation av liv för Pac-Man och att spelaren tas till en ny nivå när alla *dots* i labyrinten har konsumerats.
11	Implementation av att spelet blir svårare för varje nivå. Svårighetsgraden ökas igenom att *energizers* blir sämre och att spöken rör sig fortare.
12	Implementation av olika ljud för spelet. Till exempel ljudeffekter när spelaren dör, fångar *dots* och bakgrundsmusik.

13	Implementation av en artificiell intelligens för de olika spökena. Likadant som i originalet från 1980. Rött spöke jagar Pac-Man igenom att hitta den kortaste vägen. Rosa och blått spöke försöker att positionera sig framför Pac-man. Orangea spöke alternerar mellan att jaga och fly från Pac-Man.
14	Implementation av lista för att se tidigare högst uppnådda poäng.
15	Implementation av startskärm.
16	Implementation av ett verktyg där spelaren har möjlighet att skapa individuella labyrinter och kartor. De kartor som skapas ska kunna sparas i en fil lokalt.
17	Implementation av funktion som gör det möjligt att läsa in och köra de lokalt sparade filerna innehållande individuella kartor.

4. Övriga implementationsförberedelser

Vi kommer under projektets gång behöva ta en rad olika implementations beslut som påverkar utvecklingen och det slutliga resultatet. Grundläggande struktur och implementation är värt att fundera på redan nu för att underlätta kommande arbete.

I implementationen av spelplanen och kartan vill vi undvika för stora likheter med tidigare arbete i kursen. Samtidigt vi vill hitta en lösning som passar och är enkel att arbeta med.

Beroende på hur spelplanen implementeras behöver vi också fundera över hur Pac-Man ska hanteras. Om ett rutnät används som grund till spelplanen så kanske Pac-Man vara 3 rutor hög och 3 rutor bred. Detta skulle göra möjligheten att göra en enkel animation relativt snygg. Möjligheten att placerad *dots* i den mittersta rutan finns också med den här implementationen.

Pac-Man och spöken kommer att dela många attribut och i många situationer bete sig likadant. Här skulle det vara möjligt att dela upp koden med en abstrakt klass och sedan låta *Pacman* och *Ghost* ärva från denna abstrakta klass.

5. Utveckling och samarbete

Vi har samma ambitionsnivå för projektet och kursen. Detta är först och främst att få godkänt men målet är 5 i betyg. Vi har även för avsikt att vara klara med projektet till deadline.

Arbetet planeras att genomföras på de resurstillfällen som erbjuds och på andra lediga tider så som kvällar och helger. Planen är att arbeta mycket tillsammans även om det inte är på samma saker eller delar av projektet. Men försöka att hjälpa varandra och diskutera olika problem och frågor som uppstår.

Projektrapport

6. Implementationsbeskrivning

I detta avsnitt, och dess underavsnitt beskrivs olika delar och aspekter av själva implementationen av projektet. Här beskrivs övergripande design och vilka funktioner som finns. Även uppdelningen och relationen mellan klasser och metoder beskrivs. Vilka milstolpar som är genomförda och inte beskrivs också.

6.1. Milstolpar

Vid start av projekt satte vi upp milstolpar för att underlätta implementationsfasen av projektet. Denna lista av milstolpar har använts som vägledning under projektets gång och har hjälpt oss mycket. Här anges för varje milstolpe om den är genomförd, delvis genomförd eller inte genomförd.

1. Det finns en data typ för en spelplan samt funktionalitet för att med en grafisk komponent visa en spelplan. En testklass kan skapa en spelplan och visa den grafiskt i ett fönster.

   Helt genomfört.

2. Implementation av Pac-Man. Se till att det går att skapa en instans och att denna ritas ut på spelplanen. Redan i detta stadie ska Pac-Man figuren ha en väldigt simpel animation

   Helt genomfört.

3. Implementation av Pac-Mans rörelse. Se till att det går att röra sig i alla riktningar. Det sker på en tom spelplan.

   Helt genomfört.

4. Implementation av en labyrint med *dots* och den tunnel som finns i originalet. Detta gör det möjligt att testa kommande funktioner.

   Delvis genomfört. En karta eller bana är implementerad men utan tunnel funktionen.

5. Implementation av spöken, även dessa med en lättare animation. En enklare artificiell intelligens och möjligheten att eliminera Pac-Man implementeras också.

   Helt genomfört.

6. Ett poängsystem implementeras. *Dots* och bonusföremål ger nu poäng och poängen visas.

   Delvis genomfört. Bonus föremål är inte implementerat.

7. Implementation av en timer för spelet. Denna kommer att användas på en rad olika ställen i spelet.

   Helt genomfört.

8. Implementation av *energizers* med alla dess funktioner. Hur spökena påverkars, hur spökena kommer tillbaka till liv och hur spelaren får poäng för detta.

   Inte genomfört.

9. Implementation av en mer kompetent artificiell intelligens för spökena. Här ska även spökena släppas ut i rätt ordning och tillfälle.

   Helt genomfört.

10. Implementation av liv för Pac-Man och att spelaren tas till en ny nivå när alla *dots* i labyrinten har konsumerats.

    Helt genomfört.

11. Implementation av att spelet blir svårare för varje nivå. Svårighetsgraden ökas igenom att *energizers* blir sämre och att spöken rör sig fortare.

    Inte genomfört.

12. Implementation av olika ljud för spelet. Till exempel ljudeffekter när man spelaren dör, fångar *dots* och bakgrundsmusik.

    Inte genomfört.

13. Implementation av en artificiell intelligens för de olika spökena. Likadant som i originalet från 1980.

    Rött spöke jagar Pac-Man igenom att hitta den kortaste vägen.

    Rosa och blått spöke försöker att positionera sig framför Pac-man.

    Orangea spöke alternerar mellan att jaga och fly från Pac-Man.

    Inte genomfört.

14. Implementation av lista för att se tidigare högst uppnådda poäng.

    Inte genomfört.

15. Implementation av en startskärm.

    Delvis genomfört. Det finns en väldigt enkel startskärm. Dock inte av den kvalitén som var planerat när milstolpen sattes upp.

16. Implementation av ett verktyg där spelaren har möjlighet att skapa individuella labyrinter och kartor. De kartor som skapas ska kunna sparas i en fil lokalt.

    Inte genomfört.

17. Implementation av funktion som gör det möjligt att läsa in och köra de lokalt sparade filerna innehållande de individuella kartorna.

    Inte genomfört.

6.2. Dokumentation för programstruktur, med UML-diagram

För att förstå och kunna vidareutveckla behöver programkod dokumenteras. Denna dokumentation sker i detta avsnitt och dess underavsnitt. Dokumentation av mindre karaktär som berör enskilda metoder, fält och klasser finns i programkoden. Först beskrivs övergripande programstruktur och sedan finns det underavsnitt för de klasser som kräver mer dokumentation.

6.2.1 Övergripande programstruktur

Vårt spel styrs av en timer-tick. Vid varje timer-tick kontrolleras först inputs från spelaren, kontrollen sker i PacmanComp klassen. Sedan är det klassen StepMaker som driver spelet framåt varje timer-tick.

Denna input resulterar i en förflyttning av Pac-Man men först sker en kontroll om den önskade förflyttningen resulterar i en kollision med labyrinten eller ett spöke. Metoden *hasCollison* i den abstrakta klassen *Entity* hanterar kollision mellan Pac-Man och labyrinten. Medans en kollision mellan Pac-Man och spöke kontrolleras av metoden *hitGhost()* i *Pacman* klassen. Skulle den önskade förflyttningen inte resulterar i en kollision så sker en förflyttning av Pac-Man i önskad riktning. Skulle den önskade förflyttningen resultera i en kollision med en vägg stannar Pac-Man. Skulle den önskade förflyttningen resultera i en kollison med ett spöke förlorar spelaren ett liv och banan startas om.

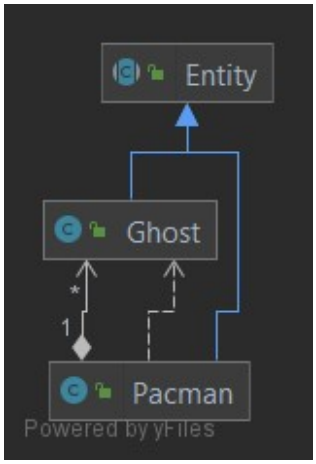
Varje timer-tick uppdateras också spökets artificiella intelligens för att hitta den kortaste vägen till Pac-Man. Detta sker i *Ghost* klassen med metoden *findShortestPath()* och med hjälp av *getNeighbours*. Utifrån denna kortaste väg förflyttas spöken tackvare metoderna *setNextMove()* och *moveToNextStep()*. Mer om klassen *Ghost* och dess artificiella intelligens finns i avsnitt 6.2.4 Ghost.

Den simpla animationen för Pac-Man och spökena uppdateras också varje tick. Det kontrolleras också om Pac-Man har ätit en *dot*, i så fall tas den bort från spelplanen och poäng delas ut. Varje timer-tick kontrolleras också om alla *dots* på spelplanen har konsumerats och nivån därmed är över. Metoderna som hanterar detta är *eatDots()* respektive *allDotsEaten()* och finns i *Pacman* klassen.

Tillslut så uppdateras skärmen med eventuella förändringar. I klassen *PacmanComp* sker sedan ut ritningen av Pac-Man, spökena och spelplanen.

6.2.2 Entity

Entity är en abstrakt klass som är representerar ett generellt entity objekt. I vårt fall är ett entity objekt Pac-Man eller ett spöke. Vi låter sedan klasserna *Pacman* och *Ghost* ärva den abstrakta klassen *entity*. Figur 2 visar ett UML-Diagram som illustrerar detta förhållande mellan klasserna *Pacman*, *Ghost* och *Entity*. I *entity* hanteras det som alla entity objekt har gemensamt så som koordinater, riktning, hastighet och förflyttning. Riktningen definieras i enum klassen *direction* och har värden RIGHT, LEFT, UP, DOWN och NONE. Entity hanterar också kollisioner mellan entity objekt och labyrinten. Detta görs med metoden *hasCollison*. Hjälpfunktioner för entity objekt som *isInMiddelOfSquare()* och *getIndex()* finns också.



(2) UML-Diagram nr. 1

6.2.3 Pac-Man

Pacman klassen är en sub-klass av *entity* och innehåller alla attribut och metoder som är specifika för *entiy* objektet Pac-Man. I *Pacman* klassen hanteras också animationen av Pac-Man, poäng, liv och kontroll om spelet är över. Kollisoner mellan spöken och Pac-Man kontrolleras ochså här av metoden *hitGhost()*

6.2.4 Ghost

Ghost klassen är en sub-klass av *entiy* och innehåller allt som är specifikt för entity objektet *Ghost*. I *Ghost* klassen hanteras animationerna av spökena och deras färger.

Ghost klassen hanterar också spökets artificiella intelligens. Den artificiella intelligensen använder sig av en BFS eller breadth first search för att hitta den kortaste vägen till sitt mål. Den artificiella intelligensen visualiserar spelplanen som en matematisk oviktad graf. Nedan finns pseudokod för att hitta den kortaste vägen.

```
Lägg till spökets nod i kön
Så länge kön inte är tom:
    Ta bort noden lägst fram i kön
    Om denna nod är Pac-Mans position:
        Spara vägen till denna nod som kortaste vägen och avsluta.
    För varje granne till denna nod:
        Om grannen inte har besökts tidigare:
            Lägg till noden i kön
            Lägg till noden i besökta noder
            Spara vägen till denna nod
```

Metoden *getNeighbours* returnerar en EnumMap med grannar för ett givet index. *findShortestPath()* är metoden som sedan kommer fram till den kortaste vägen. *moveToNextStep()* och setNextMove() gör förflyttningen av ett spöke möjligt.

6.2.5 Animation

Pac-Mans animation görs möjlig av metoderna *loadImages()*, *getImage()* och *pacAnimCounter()* i *Pacman* klassen. Animationen bygger på att alternera mellan tre olika bilder för att skapa Pac-Mans ikoniska animation. Metoden *loadImages()* laddar in alla bilder som används i animationen och lägger till dem i rätt lista beroende på riktning. Medans *pacAnimCounter()* bestämmer vilken av de tre bilderna i listan som ska visas. Sedan kontrollerar *getImage()* riktningen Pac-Man har och vad *pacAnimCounter()* kommit fram till för att returnera korrekt bild för att visa på skärmen.

Spökens animation fungerar på exakt samma sätt förutom att det finns spöken av fyra olika färger och endast två bilder bygger upp animationen och att det sker i klassen *Ghost*. Metoden *loadImages()* laddar in alla bilder som används i animationen och lägger till dem i rätt lista beroende på riktning och färg. Medans *ghostAnimCounter()* bestämmer vilken av de två bilderna i listan som ska visas. Sedan kontrollerar *getImage()* riktning på ett spöke av given färg och vad *ghostAnimCounter()* fram till för att returnera korrekt bild att visa på skärmen.

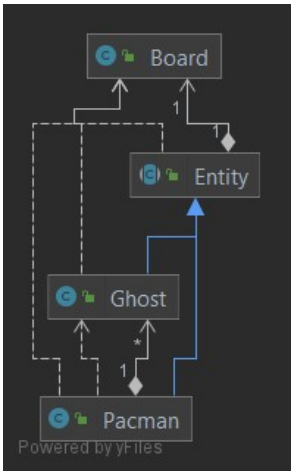
6.2.6 Board

Klassen *board* representerar ett board objekt som i vårt fall är vår labyrint. *Board* innehåller själva labyrinten och metoder för att uppdatera och återställa den. Kartan är i grunden en lista med heltal mellan 0 och 16 som sedan ritas ut med metoden *drawMap* i klassen *PacmanComp*. Heltals siffran i listan indikerar hur just den kvadraten av labyrinten ska ritas ut. Varje siffra indikerar exakt vilken kant som ska ritas ut. Se nedan:

1 = Vänster 4= Höger 2 = Övre 8 = Undre 0 = Tom 16 = *Dot*

Sedan är det möjligt att addera siffror för att skapa kvadrater som innehåller mer än en vägg, t.ex. 14 (= 4 + 2 + 8) ger en kvadrat med vägg till höger, över och under men tom till vänster. Metoden *drawMap* i *PacmanComp* sköter sedan den grafiska representationen av labyrinten. *drawMap* itererar över listan och utför en korrekt utritning beroende på siffran på de indexet.

Värt att notera är att *Board* har på många sätt en nära relation till Entity, Ghost och Pacman klasserna och deras respektive objekt. Alla dessa objekt inverkar på och kräver ett *board* för att fungera. Figur 3 visar ett UML-Diagram som illustrerar detta förhållande mellan klasserna.



(3) UML-Diagram nr. 2

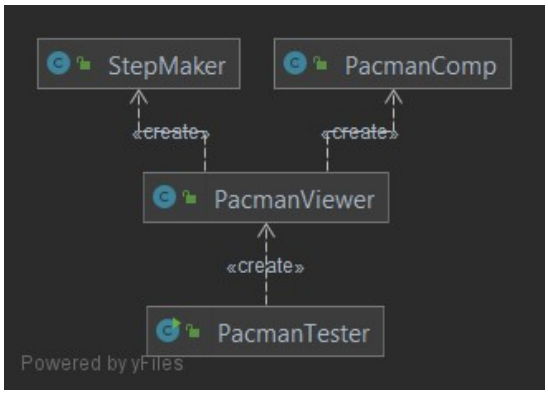
6.2.7 StepMaker

Som tidigare nämnt så är det klassen *StepMaker* som driver spelet framåt och bestämmer vad som ska hända varje timer-tick. I StepMaker kallas *tick()* metoderna som finns i *Pacman, Ghost* och *Board* vid varje tick. *Pacman.tick()* kallar i sin tur på metoder i *Pacman* som ska utföras varje timer-tick. *Ghost.tick()* kallar på metoder i *Ghost* som ska utföras vid varje timer-tick. *Board.tick()* kallar på de metoder i *Board* som ska utföras vid varje timer-tick. *StepMaker* hanterar även när de olika spökena ska släppas ut i labyrinten.

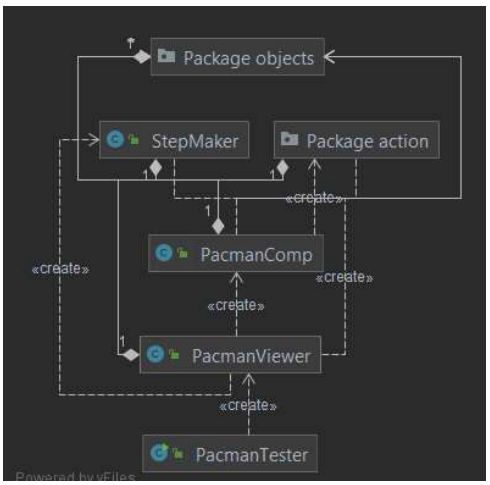
6.2.8 PacmanViewer, PacmanComp och PacmanTester

Klasserna *PacmanViewer* och *PacmanComp* hanterar det grafiska för spelet och *PacmanTester* används för att köra och testa spelet. *PacmanViewer* skapar själv rutan där spelet visas med metoden *show()*. *PacmanComp* är en sub-klass till JComponent och sköter utritningen av Pac-Man, spökena och kartan. *PacmanViwer* innehåller även metoden *initGame()* som skapar instanserna av spöken, Pac-Man och spelplanen samt initialiserar hela spelet och sätter i gång StepMaker. *PacmanTester* kallar sedan på *show() och initGame()* för att starta och köra spelet.

Figur 4 visar ett UML-Diagram som illustrerar förhållandet mellan klasserna *PacmanTester, PacmanViewer, StepMaker och PacmanComp*. Medans figur 5 visar ett UML-Diagram som visar hur klasserna i fig 4 samverkar med vissa av de resterande delarna av projektet. *Package objects* innehåller klasserna *Ghost, Pacman, Entity och Board*. Medans *Package action* innehåller *MoveAction, QuitAction och StartAction*.



(4) UML-diagram nr. 3



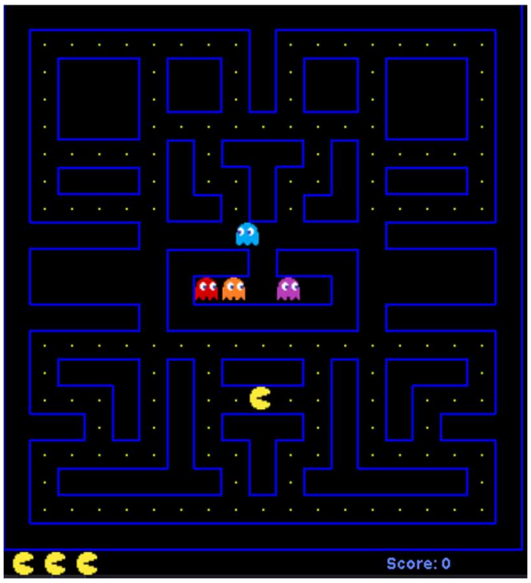
(5) UML-Diagram nr. 4

7. Användarmanual

I detta avsnitt finns en manual som förklara hur programmet fungerar. Underavsnittet *spelupplägg* behandlar de olika funktionerna och målet i spelet. Medans underavsnittet *att spela spelet* förklarar de olika kontrollerna i spelet samt hur det startas.

7.1. Spelupplägg

Pac-Man är ett två dimensionellt spel som tar plats i en labyrint. Spelaren styr Pac-Man och målet är att äta alla *dots* och gå vidare till nästa nivå. *Dots* finns placerade runt om i labyrinten och varje *dot* ger 10 poäng styck. I labyrinten finns det också fyra stycken spöken som försöker att förhindra att Pac-Man tar sig till nästa nivå. Om något av spökena lyckas komma ifatt Pac-Man förlorar spelaren ett av sina tre liv. När spelare har förlorar alla sina liv är spelet över. Till en början jagar endast ett spöke Pac-Man men för varje 300 poäng spelaren samlar släpps ytterligare ett spöke ut. När alla *dots* har fångats går spelaren vidare till nästa nivå.



(5) Labyrint vid start.

Figur 5 visar starten av spelet. Det blå spöket jagar Pac-Man medans resterande spökena är kvar i sitt näste. Vid 300,600 och 900 poäng släpps ytterligare ett spöke ut i labyrinten. De gula prickarna sprida runt banan är *dots*. I nedre högra hörnet visas poängen som hitils samlats. Spelarens liv syns i form av Pac-Mans i det nedre vänstra hörnet.

7.2. Att spela spelet

För att starta spelet krävs det att *PacmanTester.java* filen körs. Spelaren möts av en väldigt enkel startskärm, se figur 6. För att börja spelet trycks S tangenten. Kontrollerna för spelet är:

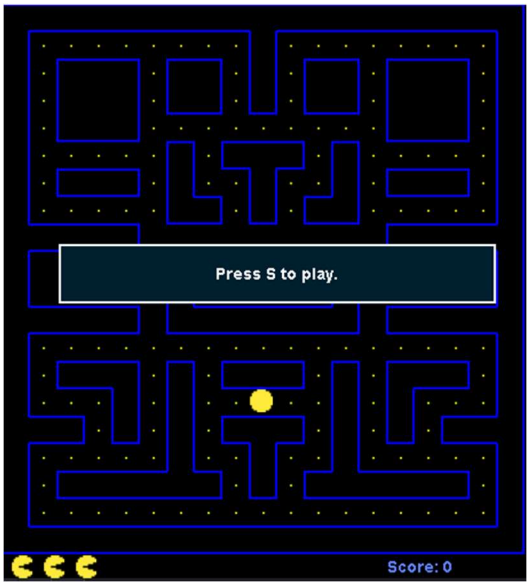
↑, för att förflytta sig uppåt.

↓, för att förflytta sig nedåt.

←, för att förflytta sig åt vänster.

→, för att förflytta sig åt höger.

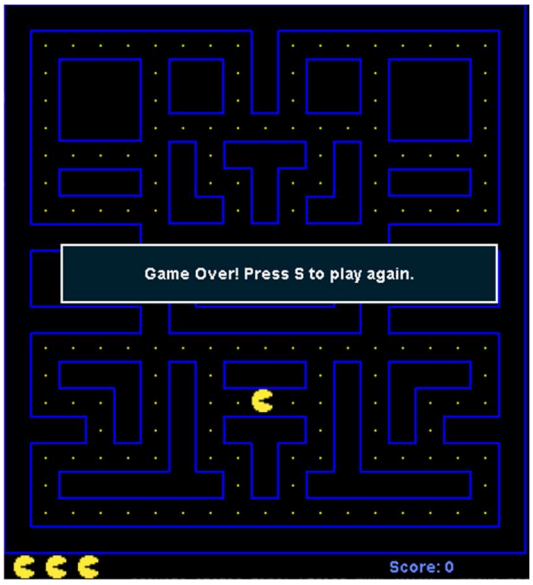
Esc, för att stänga spelet.



(6) Enkel startskärm.

Värt att notera är att i originalversionen från 1980 går det endast att byta riktning vid korsningar. I vår version går det emellertid att byta riktning var som helst.

Skulle spelare bli fångad av ett spöke och förlora ett liv så flyttas både Pac-Man och alla spöken till sina startpositioner. Samma antal spöken som jagade Pac-Man då livet förlorades kommer att fortsätta försöka fånga Pac-Man. Skulle alla tre liv förloras så börjar spelet om. Spelaren möts då av en väldigt enkel startskärm, se figur 7. Där med tappar spelaren alla sina poäng, labyrinten och spökena återställs.



(7) Enkel Game Over skärm