Projektbeskrivning

Project: Floppy Borb

2022-03-22

Projektmedlemmar:
Hugo Nilsson <hugni385@student.liu.se>
Christoffer Näs <chrna581@student.liu.se>

Handledare:
Jonathan Falk <jonfa001@student.liu.se>

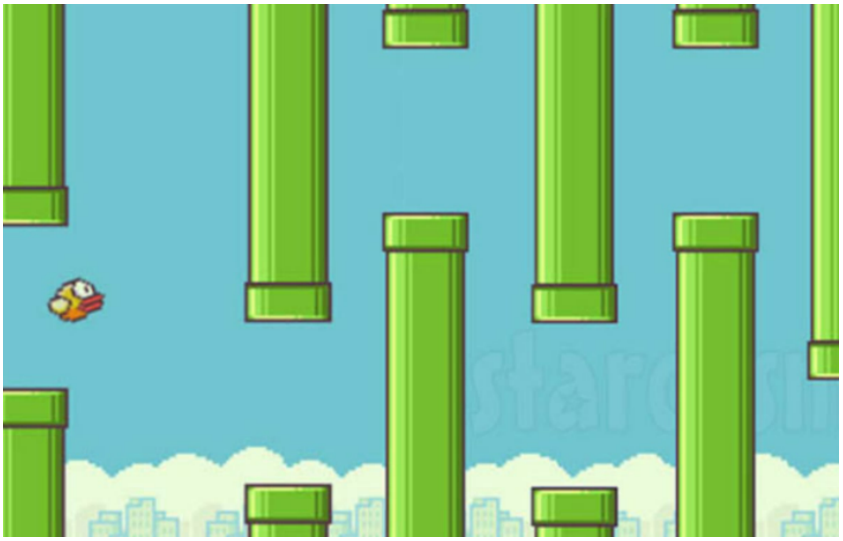
Projektplan

1. Introduktion till projektet

Vi tänker utveckla ett spel som liknar flappy bird, och vi har då utgått ifrån inspirationsprojekt av typen Vertically/Side Scrolling.

vilket innebär att spelarens karaktär rör sig framåt längs banan genom att spelplanen rör sig i motsatt riktning. Vårt mål med spelet är att samla så mycket poäng som som möjligt. Spelet avslutas när man dör. Till spelarens hjälp kommer olika power-ups finnas tillgängliga.

Det slutförda projektet kommer likna något i denna stil, fast med sämre grafik då vi kommer att prioritera ett välfungerande spel över snyggt utseende:



Figur 1.1 Inspirationen bakom vårt projekt.

2. Ytterligare bakgrundsinformation

Vårt spel kommer att vara av typen side-scroller vilket innebär att spelet följer spelaren allt eftersom den förflyttar sig höger eller vänster. Denna genre av spel var som mest populära tidigt i spelutvecklingens historia då hårdvarans begränsningar var större. För ytterligare läsning följ denna länk: https://en.wikipedia.org/wiki/Side-scrolling_video_game

3. Milstolpar

#	Beskrivning
1	Spelplan/Fönster
2	Karaktären
3	Hitbox
4	Hinder
5	Kollision med hinder
6	Spelbart spel
7	Poäng
8	Bytbar Karaktär
9	Power-Ups-Storleksförändring
10	Power-Ups-Skjuta
11	Power-up extra liv
12	Highscore
13	Rörbara hinder
14	Svårighetsgrad

Figur 3.1. *Tabell över milstolpar. Grön/Gul/Röd betyder implementerad/delvis implementerad/inte implementerad.*

Spelplan: Fönstret som spelet spelas i.

Spelkaraktären: En karaktär som spelaren kan styra uppåt och nedåt. Spelplanen rör sig så att spelkaraktären är kvar centrerat i skärmen.

Hitbox: Spelkaraktären och Spelplanens översta och understa kant har nu någon slags fysisk representation, en hitbox.

Hinder: Hinder som spelar karaktären kan kollidera med. Hindren har både grafisk och fysisk representation.

Kollision: En Collisionhandler som hanterar kollision mellan spelkaraktären och hinder. Eller kollision emellan spelaren och Spelplanens botten eller toppen.

Spelbart: Ett spelbart spel där man kan i någon utsträckning spela spelet och man kan förlora genom att kollidera.

Poäng: Lägg till något slags poängsystem som räknar poäng för att utföra specifika handlingar.

Bytbar Karaktär: Möjlighet att kunna byta spelkaraktär.

Power Ups: Power Ups är temporära eller permanenta förbättringar som spelaren kan samla på sig. Exempel på det här den röda svampen, fjädern och stjärnan i Super Mario spelen.

Highscore: Som nämnt ovan kommer spelaren kunna samla objekt som i sin tur ger poäng. Dessa poäng kommer sparas och jämföras, varpå en highscore-lista kommer att vara tillgänglig för spelaren på startmenyn.

Kollision: En stor del av spelets funktionalitet bygger på att de olika objekten faktiskt kan kollidera och således interagera på olika sätt. Därför kommer vi att implementera någon form av fysikmotor som kan hantera alla olika typer av kollisioner.

4. Övriga implementationsförberedelser

Vi tänkte att vårat spel skulle kunna byggas upp utav dessa olika klasser, fler kan självklart behövs skapas under utvecklingens gång.

Klass, Spelplan

Klass, Spelplan viewer, Ritar upp en spelplan i ett fönster så spelet kan spelas.

Klass, Spelkaraktär - Beskriver spelkaraktärens, storlek, färg, har fysisk och grafisk representation

Klass, Spelkaraktär maker, skapar en Spelkaraktär

Klass Collisionhandler, hanterar kollision emellan spelkaraktären och andra objekt.

Klass, EpicBirdComponent, används för att rita upp grafik.

Klass, Hinder, har fysisk och grafisk representation

Klass, Hinder maker, skapar upp ett hinder.

Klass, Powerups, har fysisk och grafisk representation

Klass, Powerups maker, skapar upp en powerup.

5. Utveckling och samarbete

Vi planerar på att arbeta tillsammans en stor del av projektet, för att sedan när det går ta på oss egna uppgifter såsom att exempelvis implementera olika powerups. Vi kommer att arbeta med projektet under gemensamma överenskomna tider.

Projektet kommer att ta sin början efter tenta-perioden under VT1 då vi i gruppen har mycket annat som behöver fokus och tid.

Tidsplan:

1 / 4 - 2022 - Vi har påbörjat projektet, Startat upp git repon
- Skapa ett öppningsbart fönster
    - Uppnå effekten av ett sidoscrollande spel genom att få bakgrunden att röra sig bakåt.

1 / 5 - 2022 - Spelbart spel Mål 6. Utan features

17/5 -Färdigt spel.
- Felsökning och polishing.

22 / 5 - Hård deadline
- Demonstrera och lämna in

6. Implementationsbeskrivning

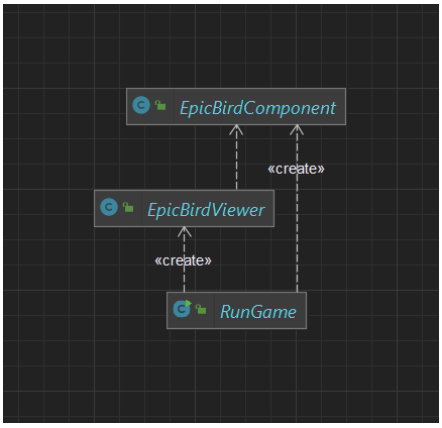
6.1. Milstolpar

Projektets implementation följer till stor del den lista som återfinns under avsnitt 3, milstolpar. Tanken var att successivt bygga upp spelet från grunden genom att en följd med allt mer avancerade funktioner läggs till. Det vill säga, varje funktion som läggs till är beroende av den föregående. Varje artikel på listan är en grov uppskattning och kan innefatta flera mindre och således olistade moment som löses allt eftersom som de uppstår. Se *Figur 3.1* för komplett lista över implementerade milstolpar. Grön betyder att funktionen implementerades. Gul betyder delvis implementerad och röd betyder att funktionen aldrig implementerades.

6.2. Dokumentation för programstruktur, med UML-diagram

6.2.1 Spelplan/Fönster

Med spelplan/fönster menas det faktiska fönstret i vilket spelet syns och således spelas. Detta sköts främst av klasserna *EpicBirdViewer* och *EpicBirdComponent* men också i mindre utsträckning *RunGame* då den fungerar som en brygga mellan det grafiska och det logiska. Klassen *EpicBirdViewer* är den klass som skapar det faktiska fönstret i vilket spelet syns och köres. *EpicBirdComponent* sköter all grafisk utritning. Mer om grafiken återfinns under rubrik 6.2.2.1. *RunGame* initialiserar alla spelobjekt och lägger in dem i spelet. Dessa objekt ritas sedan ut av *EpicBirdComponent*. I figuren nedan syns tydligt hur *EpicBirdComponent*, *EpicBirdViewer* och *RunGame* samarbetar.



Figur 6.2.1.1 *UML-Diagram över spelfönstrets uppbyggnad.*

6.2.2 Karaktär och spelobjekt

Varje objekt består av två delar, en grafisk och en logisk. Den grafiska är den som syns på skärmen och den logiska hanterar det fysiken, däribland rörelse och kollisioner. Alla objekt ärver sina egenskaper från interface-klassen *EpicBirdObject.* Detta innefattar fiender, meteorer, rör och power-ups. Klassen *EpicBirdObject* är av typen interface, vilket bäst kan beskrivas som en typ av “regelbok” för vilka metoder som varje objekt-klass måste innehålla.

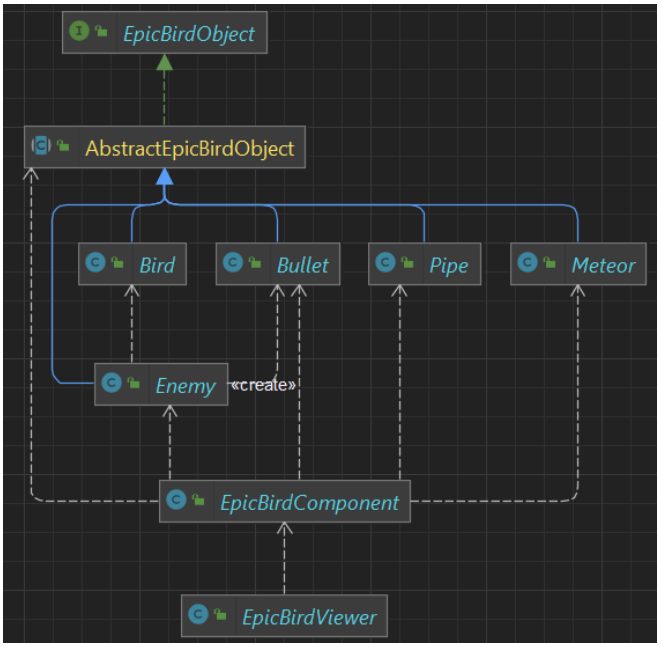
6.2.2.1 Grafik

Spelets karaktär, officiellt benämnt fågeln, består av en tvådimensionell array. Genom att strategiskt fylla varje plats i arrayen med en rut-typ är det möjligt att framställa en form som efterliknar en fågel. Alla föremål i detta spelet använder sig av denna metod. Tvådimensionella arrays valdes främst eftersom de innebär ett överskådligt tillvägagångssätt till utritning. Varje cell i arrayen innehåller ett typ av block, kallat SquareType, som bestämmer vad som ska ritas ut. Där finns två olika övergripande typer av SquareType; EMPTY eller FILLED. FILLED är ingen egentlig SquareType utan kan delas upp i ett flertal olika subtyper som berättar för utritningsmodulen *EpicBirdComponent (EBC)* vilken färg som ska ritas ut. Detta sker genom en for-loop i *EBC* som successivt går igenom objektets array. Se figur 6.2.2.1.1 nedan för referens.

```
public SquareType[][] bird1() {
    SquareType[][] birds1 = { { SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY },
            { SquareType.EMPTY, SquareType.BLACK, SquareType.EMPTY, SquareType.BLACK, SquareType.EMPTY },
            { SquareType.BLACK, SquareType.EMPTY, SquareType.BLACK, SquareType.EMPTY, SquareType.BLACK },
            { SquareType.BLACK, SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY, SquareType.BLACK },
            { SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY, SquareType.EMPTY } };

    return birds1;
}
```

Figur 6.2.2.1.1 *Uppbyggnaden för en av de spelbara karaktärerna i en 2-dimensionell array.*



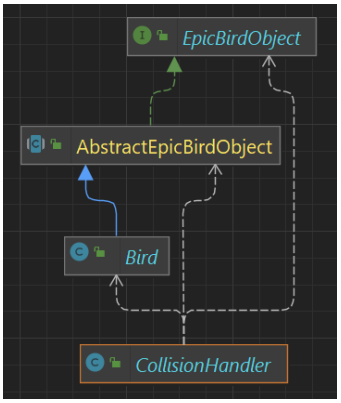
Figur 6.2.2.1.2 *Diagram över de spelobjekt som ritas ut på skärmen.*

6.2.2.2 Logik

Spelets logik hanteras i ett flertal olika klasser och består nästan exklusivt av rörelser eller kollisioner. Kollisioner hanteras i en egen klass benämnd *CollisionHandler*. Denna fungerar på ett liknande sätt som *EpicBirdComponent* och går under varje tick igenom fågelns array med en for-loop för att kolla ifall den kolliderar med ett annat objekt. Om sådant är fallet stoppas spelet och en flagga, GameOver, sätts till sann. För att bespara processorn onödig beräkningskraft kollas enbart objektets yttre kanter eftersom dessa kommer kollidera först.

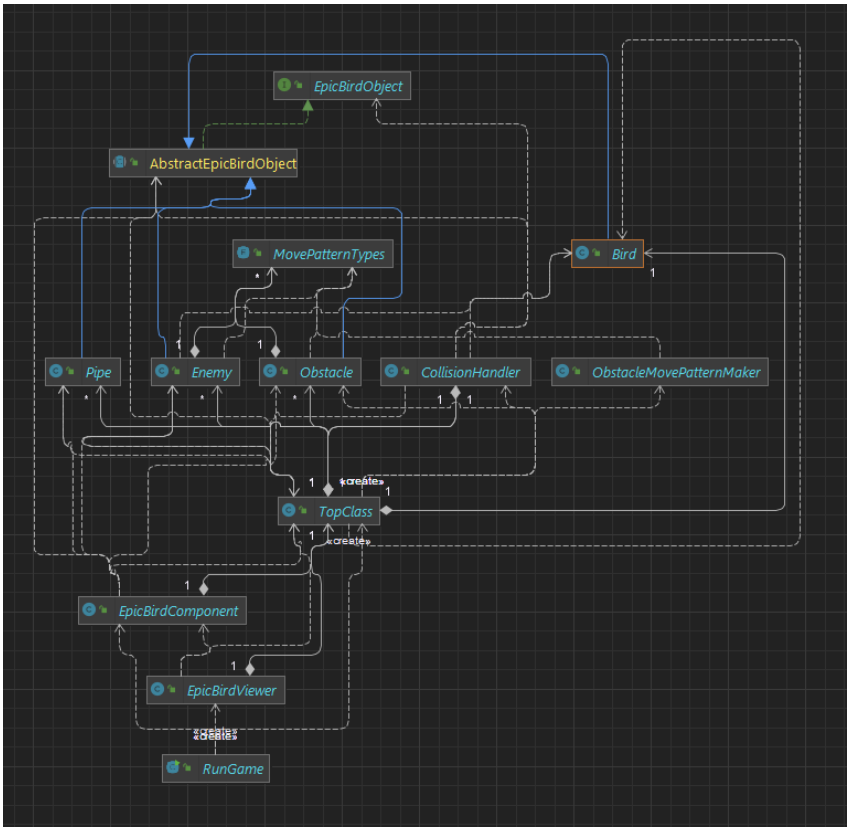
Spelets rörelse går att dela in i två kategorier; spelarstyrd och förutbesämd. Den spelarstyrda är av enkel konstruktion. En KeyListener reagerar på att mellanslagstangenten trycks och kallar på funktionen *moveUp* i *TopClass* vilket förflyttar fågeln uppåt i y-led. Fiender rör sig utefter ett förutbestämt mönster som specificeras i klassen *ObstaclesMovePatternMaker*. Dessa rörelsemönster är enkla listor med riktningar. Varje rörelse är definierad i enum-klassen *MovePatternTypes* för bättre kodöversikt och återanvändning.

Spelets side-scrolling effekt skapas av att funktionen *movePipe* i *TopClass* anropas en gång varje tick. Denna funktion förflyttar alla rör bakåt i x-led tills det att de är utanför skärmen vilket resulterar i illusionen att fågeln flyger framåt.



Figur 6.2.2.2.1 *Diagram över hur kollisioner och fågeln hör ihop. Notera hur inga andra klasser är inkluderade i diagrammet då det enbart är fågeln som undersöks för kollision.*

6.2.2.3 Sammanslaget UML-Diagram över karaktär och spelobjekt



Figur 6.2.2.3.1 *UML-Diagram över spelets olika objekt och hur de samarbetar med de klasser som hanterar grafik och fysik. Notera här hur de olika spelobjekten, d.v.s pipe, enemy och obstacle ärver från AbstractEpicBirdObject och EpicBirdObject.*

6.2.3 Tick

Som många andra spel är även detta baserat på ett fast tickande vilket är jämförbart med hur en klocka tickar framåt i tiden. Implementationen består av två komponenter; en timer och en funktion *tick*. Denna timer återfinns i klassen *RunGame* under namnet *gameTick*. Funktionen *gameTick* ropar sedan på *tick*-metoden i klassen *TopClass* som i sin tur ropar på en rad metoder vars syfte är att uppdatera de olika objekten i spelet, exempelvis fiender och rör. Metoden *gameTick* kallar även på *updateFrame* i klassen *EpicBirdViewer* som i sin tur kallar på PaintComponents inbyggda *repaint*() metod. Den sistnämnda metoden, *repaint*(), är ansvarig för att objekten som ritas ut på skärmen faktiskt uppdateras.

```
final Action gameTick = new AbstractAction()
{
    public void actionPerformed(ActionEvent e) {
        //Tick, updates
        topclass.tick();
        viewer.updateFrame();

    }
};
final Action timerSpaceJmp = new AbstractAction() {
    public void actionPerformed(ActionEvent e) {

    }
};
final Timer clockTimer = new Timer(TIME_DELAY, gameTick);
clockTimer.setCoalesce(true);
clockTimer.start();
```

Figur 6.2.2.1 *Uppbyggnaden av timer-funktionen som är ansvarig för spelets tickande.*

6.2.4 Övriga funktionaliteter

6.2.4.1 Poängsystem

Spelet räknar poäng som spelaren samlar när den framgångsrikt flyger förbi och undviker ett rör. Detta sker genom att fågelns x-koordinat jämförs med rörets och ifall dom är ekvivalenta läggs ett poäng till.

```
public void points() {
    for (Pipe pipe : pipes) {
        if (bird.getPositionx() == pipe.getPipePositionX(pipe)) {
            points += 1;
        }
```

Figur 6.2.4.1.1 *Uppbyggnaden av poäng-funktionen.*

6.2.3.2 Restart funktion

För ett ordentligt spelbart spel ställdes det krav på att efter förlust kunna starta om. Alla objekt i spelet läggs vid sin skapelse till i en för objektet specifik lista. Efter att objektet utfört sin uppgift eller efter det lämnat skärmen raderas objektet från listan och försvinner således från skärmen. Funktionen *resetGame* återställer spelets alla listor och om-initialiserar hela spelet vilket gör det möjligt för spelaren att fortsätta spela efter förlust.

```
public void resetGame(){
    obstacles.clear();
    pipes.clear();
    enemies.clear();
    bullets.clear();
    meteors.clear();
    powerups.clear();
    points = 0;
    bird.position.x = 700;
    bird.position.y =350;
    gameover = false;
    initializeGame();
}
```

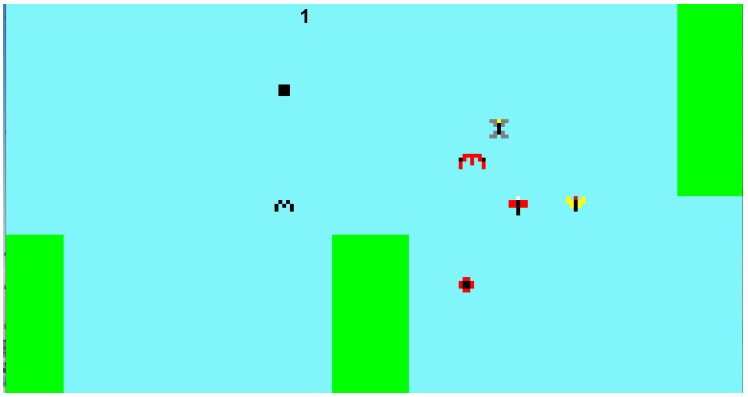
Figur 6.2.4.2.1 *Strukturen av points-funktionen som är ansvarig för spelets poängräkning.*

6.2.4.3 Powerups

En av spelets utökade funktioner som stod med på listan över milstolpar är “powerups”. Powerups är objekt med vilka spelaren kan kollidera för att ta del av särskilda förmågor som kan vara till fördel. Exempel på en sådan powerup är den svart-gröna cirkelformade formen som ibland uppenbarar sig på spelplanen och ger spelaren 10 extra poäng.

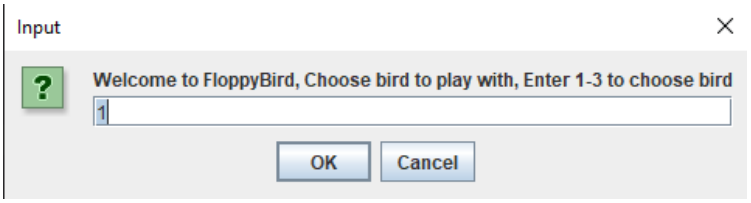
7. Användarmanual

Målet i detta spel är att flyga så långt som möjligt utan att krocka med något av spelets hinder. Dessa hinder uppenbarar sig främst i rör som förflyttar sig bakåt horisontellt men även i rörliga hinder i olika former och storlekar. Notera poängen som visas i överkanten av skärmen.



Figur 7.1. *Spelets utseende. Fiender är de former i olika färger till höger. En av spelets olika powerups är här synlig precis ovanför mitten av röret till höger av fågeln och uppenbarar sig som en rund, svart-röd prick.*

För att starta spelet måste användaren köra metoden *main* i klassen *RunGame* eller alternativt dubbelklicka på jar-filen. Fågeln styrs med mellanslagstangenten. Användaren kommer då bli presenterad med valet 1-3 där varje val innebär en unik fågel av olika storlekar.



Figur 7.2 *Rutan i vilket spelaren uppmanas mata in en siffra mellan 1-3 för att välja fågel.*

Skulle spelaren förlora genom att krocka med ett av hindrena är det möjligt att klicka på tangenten ‘r’ och således börja om från början.

Figur 7.3 *Spelaren har kolliderat. Tryck på ‘r’ för att spela igen.*