Det bortglömda och fantastiska originala Flappy Bird

Flappy Bird

2024-02-22

Projektmedlemmar:

Kevin Magron kevma271@student.liu.se

Handledare:

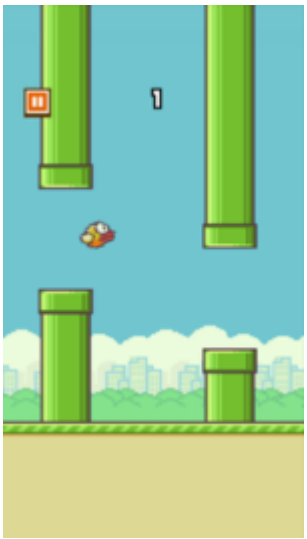
Hampus Holm handledare@liu.se>

Innehåll

Projektplan

1. Introduktion till projektet

Detta spel handlar om en flaxande fågel. Har man spelat Mario kommer man att känna igen miljön för fågeln måste undvika att flyga in i gröna rör som sticker upp från marken och ned från himmeln. Om detta inte låter utmanande nog kommer det då och då att komma upp oförväntade hinder och fågeln kan få konstiga power-ups.



2. Ytterligare bakgrundsinformation

Flappy Bird är ett simpelt koncept. Man använder sig av mellanslagsknappen för att förflytta sig uppåt på brädet. Det gäller att man motverkar gravitationen samtidigt som man undviker att flyga in i rören.

3. Milstolpar

#	Beskrivning
1	Skapa ett bräde.
2	En grafisk komponent som kan visa en spelplan i rätt bakgrund. Det går att stänga fönstret genom att klicka på stängknappen.
3	Skapa fågeln och få den att synas på rätt position.
4	Lägga till funktionalitet för knapp och mustryck.
5	Göra så att knapptryck ger realistiska rörelser med hjälp av gravitation.
6	Implementera kollision med marken och taket.
7	Skapa en “toppipe”. Få rören som ska hänga från luften att synas på ett rimligt sätt och avstånd. De ska skapas i en slumpmässig höjd från taket.
8	Få spelet tickande. Implementera en timer och få pipe’sen att röra sig mot fågeln.
9	Implementera de lägre rören. De ska skapas och röra sig tillsammans med de övre rören.
10	Implementera kollision mellan fågeln och rören.
11	Spelet ska kunna ta slut någon gång.
12	Räkna spelarens poäng och visa det på skärmen.
13	Spelet ska kunna startas om.
14	Det ska gå att pausa spelet.

15	Implementera en power-up för fågeln. Den ska då och då få mycket högre kvaxkraft.
16	Implementera en andra power-up. Gravitationen ska bli kraftigare ibland.
17	Implementera en tredje power-up. Spelet ska bli snabbare. Eftersom takten i flappybird inte ökar ska det sedan gå tillbaks till normalt tillstånd.
18	Implementera ljud. Det ska låta när man hoppar och när det blir en kollision.
19	Skapa en loadingscreen. Den ska visas i antal sekunder innan spelet börjar.
20	Skapa highscores som sparas mellan varje runda. Varje highscore ska vara kopplat till ett användarnamn.
21	Skriv och läs highscores i en fil på datan. Dessa ska följa kursens undantagshantering och kasta undantag uppåt .
22	Skapa dialogfönster som varningar till användaren när det sker en exception.

4. Övriga implementationsförberedelser

Jag tänker ha en klass som har ansvar för det som har med det “fysiska” brädan att göra. Här ska alla metoder finnas som gör att spelet går framåt och är spelbart. En klass för att köra spelet och ansvara för att starta om. Ett spel som visar och ritar upp allt visuellt. En klass för mina rör. En för highscores. Ett gränssnitt för lyssnare. Klasser för mina power-ups.

5. Utveckling och samarbete

Jag utförde detta projekt själv och tycker att det har gått bra. Jag hade ingen plan på när jag skulle göra vad utan gjorde det som behövdes när tiden fanns. Totalt skulle jag gissa på att den rekommenderade tiden för projektet på ungefär 180 timmar har lagts ner. Totalt sätt skulle jag påstå att arbetet flöt på bra.

6. Implementationsbeskrivning

6.1. Milstolpar

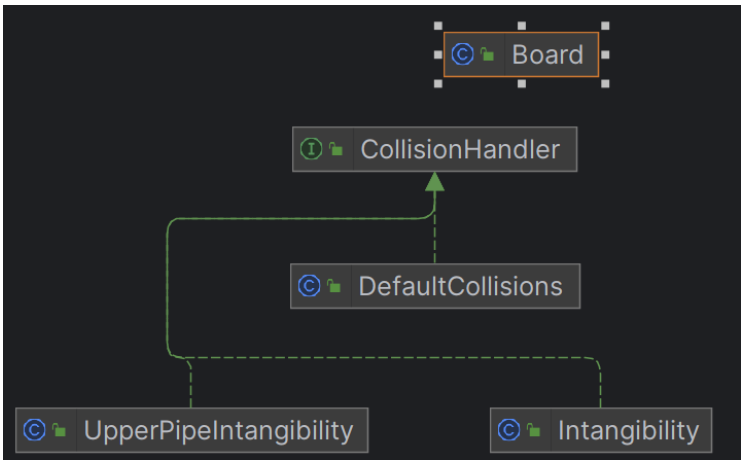
Milstolple 1-3 är helt implementerade. Den fjärde milstolpen implementerades inte då det inte ansågs vara nödvändigt med mustryck. Gravitation, kollision, övre rör, tick, lägre rör, vidare kollision, spel slut, poäng och omstart (milstolpe 5-13) är alla fullt implementerade. Milstolple 14 – pausa spelet – implementerades inte. Inte heller milstolpe 15 implementerades. Milstolpe 16 implementerades men inte 17 och 18. Resterande miilstolpar är implementerade (19-22).

6.2. Dokumentation för programstruktur, med UML-diagram

Spelet styrs från klassen GameEngine som är ansvarig för att start, omstart och avslut av spelet. Det är även här som timern – vilken är satt på 60 tick per sekund (60 FPS) - initieras och körs. Denna timer kallar på en tick-metod i Board klassen som ansvarar för att alla objekt - fågeln och rören - förflyttas samt att det fysiska spelmekaniken fungerar. För att representera detta visuellt används GameScreen som kallas på från brädet varje gång något har förändrats - vilket är varje tick i detta spel. Den ritar upp brädet och alla objekt på rätt position.

Board och GameEngine är huvudklasserna för programmet kan man säga. Det är de som instansierar de flesta klasserna och det är genom brädet som de flesta klasser hämtar sin information om spelläget. Det är därför som de flesta klasser tar in brädet i sin konstruktor, för att det endast finns ett bräde åt gången och på så sätt får dem tillgång till dess fält genom metoder.

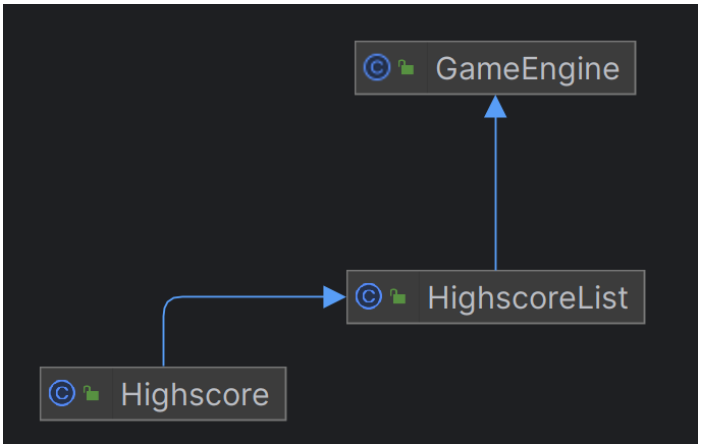
Collisions



I collisions paketet finns det 4 klasser. Den första är ett gränssnitt som fungerar som ett kontrakt för att definera och bestämma hur alla subklasser ska fungera. Det är 2 metoder som de bör innehålla. En

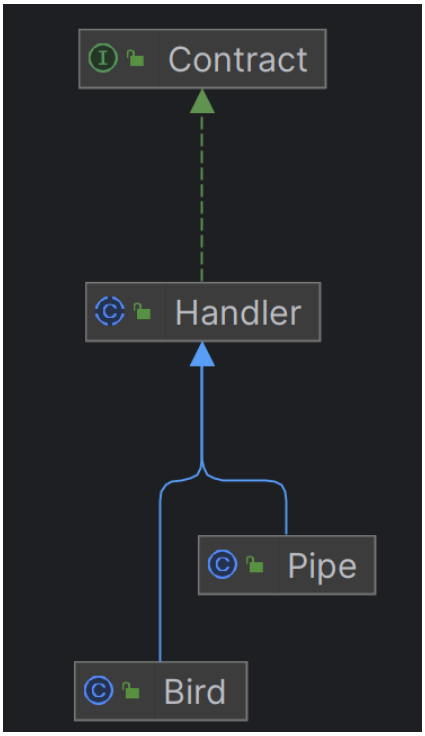
metod för att kolla kollisioner samt en för att returnera vilken typ av kollision som används. I brädet skapas ett objekt av typen CollisionHandler som till en början sätts till DefaultCollisions. Detta objekt byts ut allt eftersom spelet rullar.

Highscore



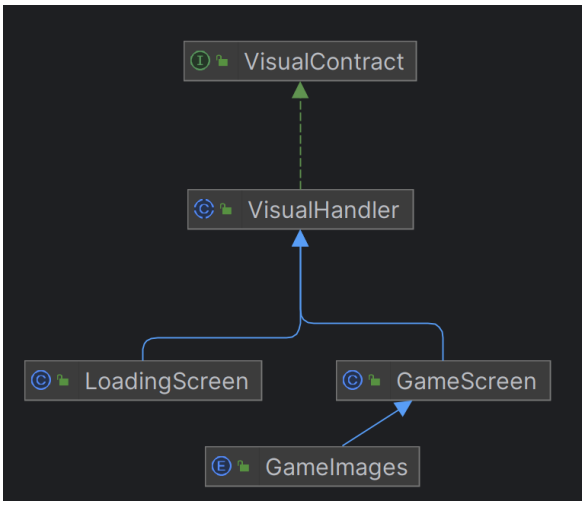
Paketet med highscores har också en enkel struktur som skulle kunna beskrivas med diagrammet ovanför. GameEngine instansierar en highscorelist som används tills att programmet avslutas. Den instansierar även highscore för varje avslutat runda. Dessa läggs till i highscorelistan om de har tillräckligt högt värde. Eftersom programmet utnyttjar en inkapslad struktur skickas varje instans av highscore med till highscorelist för att sedan hämta korrekta värden.

Objects



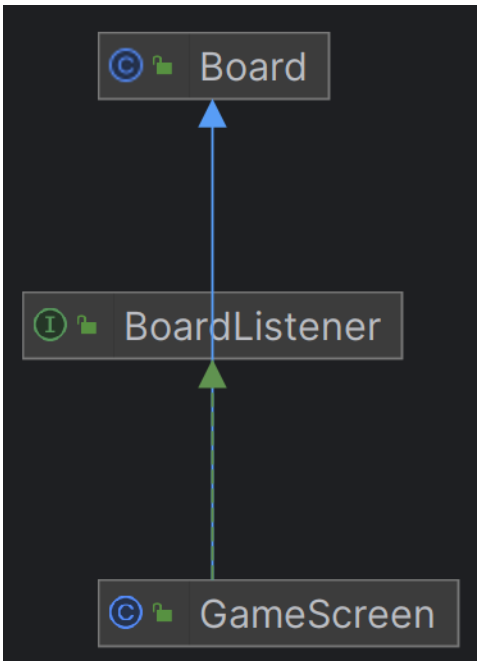
I object paketet finns alla fysiska objekt i spelplanet med. Det finns ett gränssnitt för att definera och nedärva relevanta metoder som krävs av alla subklasser. Alla dessa objekt instansieras i brädet. I mitten av hierarkin finns den abstrakta klassen Handler som innehåller gemensamma fält och metoder för att uppfylla en “DRY” kod. Dessa nedärvande klasser gör inte så mycket för sig, men gör programstrukturen mer objektorienterad och lättförståelig. Hade man i stället skapat dessa direkt i brädet, skulle andra klasser som instansierar objekten att hela tiden kommunicera genom brädet för att komma åt dem. Dessutom skulle klassen för brädet bli alldeles för lång och stökig. En annan fördel är att koden blir självförklarande och logisk. För att hämta värdet för fågelns hastighet skriver man bird.getSpeed(). På detta sätt blir även fält inkapslade och tillgängliga endast genom givet objekt.

Visuals



I visuals paketet existerar alla klasser som har ett visuellt fönster och ritar ut något. Högst upp i hierarkin existerar ett gränssnitt som återigen definierar ett kontrakt för alla implementerande klasser. Här finns de metoder som alla klasser bör ha ifall de ska rita ut ett visuellt fönster. Näst högst upp i hierarkin finns en abstrakt klass som innehåller gemensamma metoder för dem implementerande klasserna. Då dessa visuella klasser skapar, ritar och stänger ner sina fönster har de vissa metoder som är exakt likadana. För att undvika upprepning finns dessa i den abstrakta klassen. Näst längst ner i hierarkin är dessa visuella klasser som implementerar kontraktets metoder som inte redan är definierade i den abstrakta klassen. Sist finns Enum klassen GameImages som innehåller namn på de olika bilder som används. Den är användbar i klasser med flera bilder där java maps används. På så sätt undviker vi att koppla varje bild till en sträng, som hade lett till mer svårläst kod samt större risk för fel.

BoardListener



I detta program används endast en klass för att ruta och uppdatera det visuella fönstret. Denna klass (GameScreen) är en BoardListener. Varje sådan bör implementera 2 metoder. Den ena som har med power-ups att göra och den andra som har med förändringar i spelet att göra. Dessa boardlisteners blir informerade genom brädet.

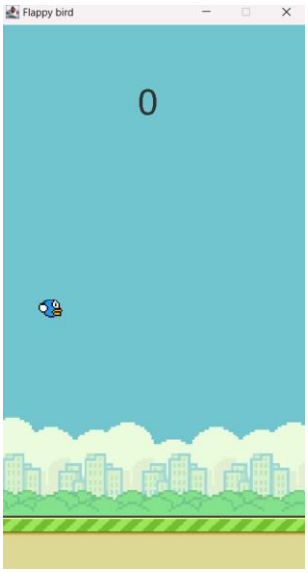
I spelmotorn (GameEngine) hanteras alla undantag som kastas. Det hanteras på så vis för att följa javapraxis för undantagshantering där man hanterar de på högsta nivån. För att förbättra läsbarheten och strukturen på programmet existerar ErrorHandler, en klass för alla olika fel som uppstår när programmet körs. Alla undantag anropar statiska metoder från denna klass. I o med att metoderna är statiska behövs heller ingen instans av ErrorHandler.

Till sist finns det 2 enum klasser (Direction, PowerUp) som definerar olika rörelseriktningar samt power-ups. Detta gör koden mer läsbar, bättre strukturerad och minskar risken för fel, jämfört med att använda strängar.

7. Användarmanual

När man startar programmet möts man av en loadingscreen som varar i 4 sekunder.

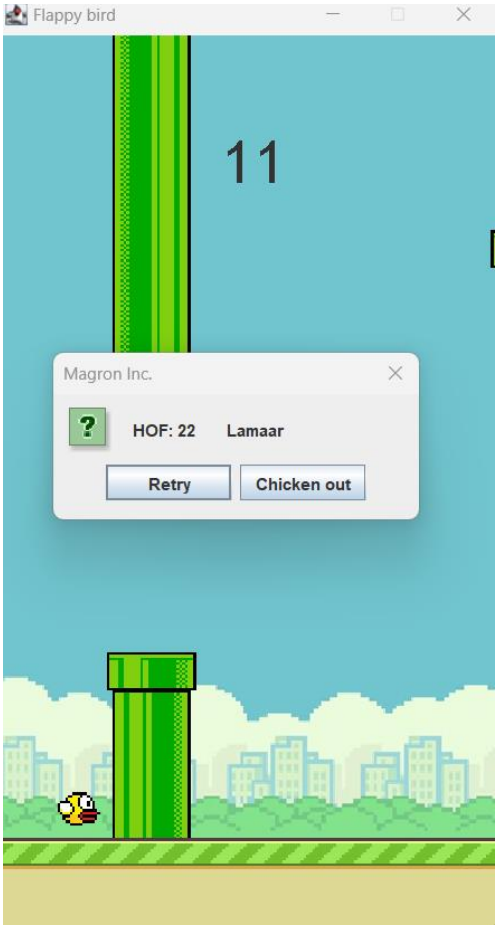
Sedan inväntar programmet på knapptryck från använder. I nuläget finns det endast en rörelse som använder kan göra, vilket är en uppåt rörelse, eller ett fågelkrax. Detta kan göras med hjälp av W, mellanslag eller uppåt pilen, beroende på vad man föredrar.



Sedan gäller det att undvika kontakt med rören, taket och golvet i spelet. Man får poängför varje rör man passerar. Då och då (slumpmässigt) kan man få en power-up. Det finns 2 olika power-ups.

- Gravity. Om fågeln blir röd innebär det att gravitationen blivit dubbelt så stark. Alltså krävs det fler fågel kraxar för att hålla sig uppe. Till sin fördel kan användaren flyga genom den övre delen av de övre rören. Alltså ovanför den utstickande “svampen”.
- Super. När fågeln blir blå kommer den att flyga genom alla rör i hög hastighet och samla poäng per sekund i stället. Man är odödlig så länge man inte faller på golvet.

Båda superkrafterna ger en odödlighetsskydd efter att de försvunnit, för att ge användaren tid att återhämta sig. Superkrafterna varar också under en slumpmässig tidsgång.



Highscores sparas lokalt på datorn. Och som i det originala Flappy Bird visas endast det högsta poänget som en användare har fått. Mitt personliga rekord är 63.