Projektbeskrivning

Ett bilspel
2017-05-07

Projektmedlemmar:
Gustav Andersson <gusan092@student.liu.se>

Handledare:
Mikael Nilsson <mikael.a.nilsson@liu.se>

Table of Contents

Planering

1. Introduktion till projektet

Mitt program är ett top-down spel med rullande bakgrund. Det går ut på att samla poäng genom att ta sig så långt som möjligt med en bil som man styr med tangenter under tiden man undviker hinder för att inte mista liv och samlar power ups för att få fördelar. Man kommer ha en begränsad area att styra sin bil på med piltangenterna under tiden objekt faller neråt för att skapa effekten att bilen åker framåt. Så länge man inte krockar med ett hinder kommer hastigheten av alla objekt öka vilket också ökar svårighetsgraden. Ett annat sätt spelet kommer öka i svårighetsgrad är att svårare hinder kommer läggas till ju längre du kommer till exempel hinder som styrs genom simpel form av AI.

Utöver spelet kommer det finnas möjlighet att spela multiplayer lokalt, ändra inställningar som till exempel vilka tangenter som gör vad.

Det kommer finnas möjlighet att spela multiplayer (lokalt). Då är det den som dör sist som vinner.

3. Milstolpar

#	Beskrivning
1	Det ska finnas en spelmotor som sköter grafiken och tick samt ett fönster som visar spelplanen.
2	Det ska finnas en spelare i form av en bil som går att styra med piltangenterna.
3	Det ska finnas simpla hinder som faller neråt på spelplanen. Det ska också finnas en kollisionshantera till spelaren. Spelplanen ska utökas med begränsningar av vart spelaren kan köra.
4	Spelaren ska ha liv och poäng som visas någonstans på spelplanen.
5	Det ska finnas power ups som ändrar förhållanden i spelet.
6	Det ska finnas en startmeny där man kan starta spelet ifrån.
7	Det ska gå att pausa spelet samt förlora med hjälp av en fönstermeny eller snabbkommandon. Det ska också finnas en highscore-lista som man skriver in sig i när man förlorar. Den ska visa namn, datum och poäng sorterade med högst poäng först samt gå att komma åt via startmenyn och när spelet är pausat.
8	Det ska finnas ett spawn objekt som lägger till objekt på spelplanen. Den ska bero på antalet tick spelplanen gjort.
9	Lägg också till några flera simpla hinder med mer rörelse samt mer omgivning.
10	Lägg till ett hinder som styrs av en AI.
11	Det ska gå att välja färg på spelaren innan man startar spelet. Lägg till

	flerspelarläge med nya kollisionshanterare som inte minskar farten.
12	Det ska gå att byta samt spara inställningarna i spelet både under spelets gång och i startmenyn.
13	Det ska finnas hinder som är motsatsen till power ups, som påverkar spelaren negativt när spelaren kollidera med dem.
14	Det ska finnas en smartare AI som styr ett hinder som försöker förstöra för spelaren.
15	Det ska gå att spela fler på likadana spelplaner men inte samma.
16	Det ska spelas upp musik i bakgrunden. Det ska också finnas val att stänga av musiken i inställningarna.
17	Det ska gå att spara ett spel och öppna det någon annan gång.

4. Övriga implementationsförberedelser

Det ska finnas en klass för spelmotorn, detta ska vara ett fönster som innehåller allting som spelet innehåller. Den innehåller en tick som driver spelet och en render som ritar ut grafiken.

Det ska finnas en abstract klass för spelobjekt som innehåller egenskaper hos alla spelobjekt som kommer befinna sig på spelplanen, till exempel tick och render.

Det ska finnas en klass som sparar ner spelobjekt i en lista. Denna har en tick och en render som anropar alla spelobjekts tick ock render.

Det ska finnas kollisionshanterare i form av ett interface som till exempel spelaren använder sig av.

Power ups kommer inte vara ett spelobjekt utan bara ha en funktionaliteten att förändra spelobjektet. Dessa ska ett spelobjekt bära på och aktivera när de kolliderar med ett spelobjekt.

Jag kommer visa programmet i samma Jframe hela tiden och bara byta dess komponenter beroende på vart man är i programmet.

5. Utveckling och samarbete

Jag har ambitioner till en femma och kommer därför fokusera på att skriva snyggkod och inte använda mig av halvdana lösningar.

Slutinlämning

6. Implementationsbeskrivning

Projektet är uppdelat i tre delar, en del för spelet, en för användargränssnittet och en för hantering av properties. Tanken är att alla klasser sköter i stort sätt sig själv alltså Game styr kollar inte efter kollisioner eller App kollar inte om Game är game over.

Användargränssnittet

App är huvudklassen med det menar jag för att starta programmet skapar man en App. App skapar antingen ett spel eller en AComponent. Dess enda syfte är att visa innehållet i programmet. Menu är en JMenuBar till App som aldrig byts ut. Den fungerar istället genom att inaktivera knappar som inte ska kunna användas på vissa ställen och aktivera de som ska kunna användas.

Spelet

Klassen Game är spelet och innehåller därför allt kopplat till ett spel. Game är en tråd som kör en loop. Spawner och Handler är klasser som hjälper till med att hantera GameObject's. GameObject's är det som påverkar spelet visuellt samt gör spelet till ett spel. Hud är bara ett visuellt tillägg. PowerUp är en abstract klass för tillägg till Player men ger upphov till väldigt skilda subklasser huvudsakligen begränsade av Player. CollisionHandler är interfacet för kollisionshantering för Player.

Properties

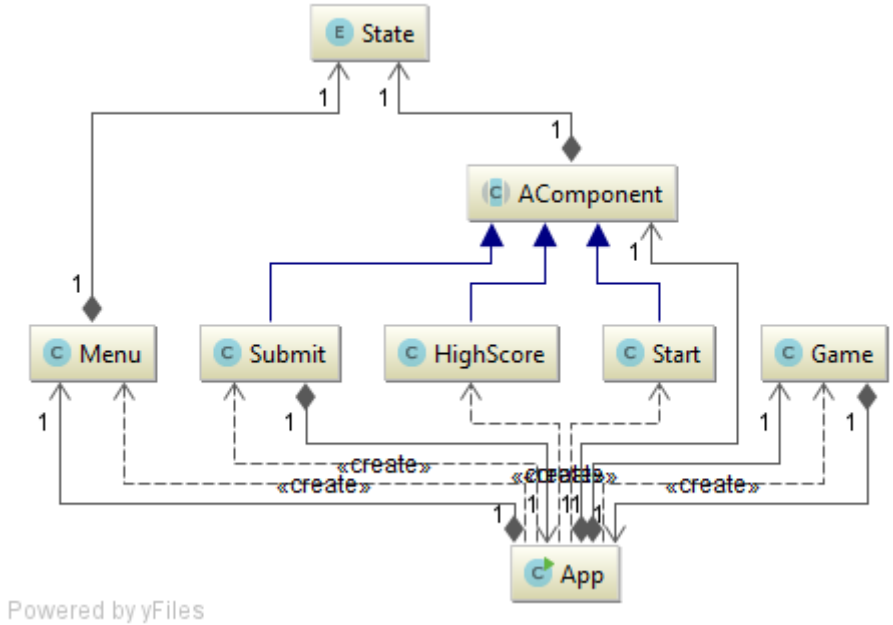
AppProperties är en abstract klass för hanteringen av olika properties filer. Dock fungerar den mer som en hjälpklass för att skriva och läsa data. Subklasser kan därför skilja sig mycket åt beroende på hur den ska hantera properties filer.

6.1. Milstolpar

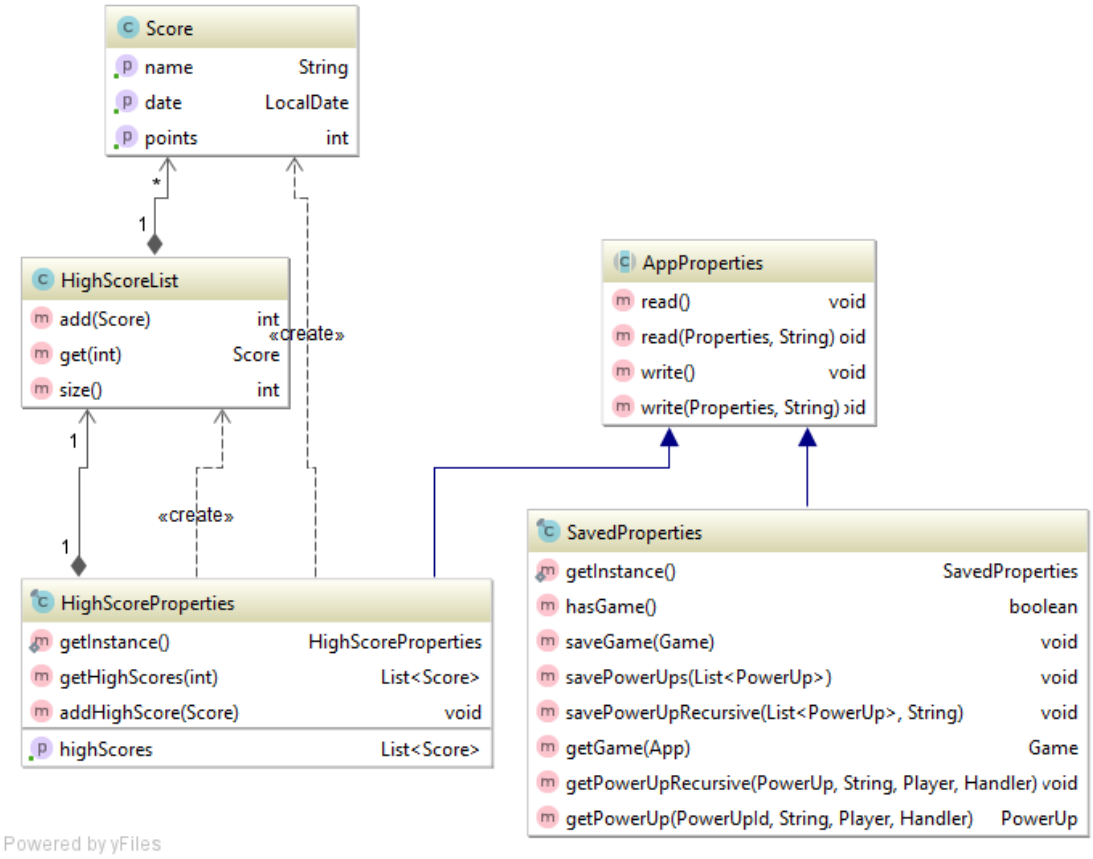
1. Helt
2. Helt
3. Helt
4. Helt
5. Helt
6. Helt
7. Helt
8. Helt
9. Helt
10. Inte alls
11. Inte alls
12. Inte alls
13. Inte alls
14. Inte alls
15. Inte alls

16. Inte alls
17. Helt

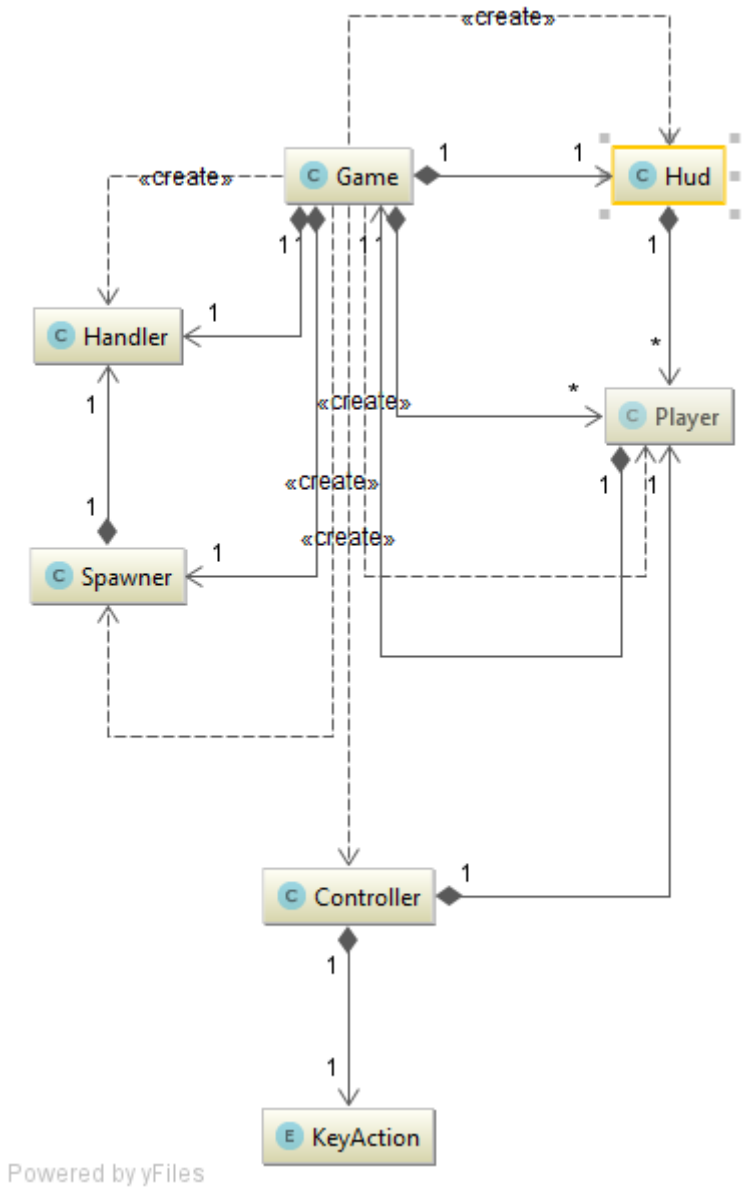
6.2. Dokumentation för programkod, inklusive UML-diagram



App är en subklass till JFrame. I den finns en main-metod som skapar en App. I konstruktorn läggs en Menu och en Start till. Menu är en subklass till JMenuBar. Menu's JMenuItem's använder ActionListeners med metoder definierade i App. Den använder också EnumMap's med State och listor för att göra olika JMenuItem tillgängliga eller otillgängliga för vissa State's. Varje State representerar varsin AComponent samt Game detta för att Menu ska med metoden setState() veta vilken AComponent eller Game App visar. App visar alltid högst en komponent, antingen en AComponent som är en abstrakt subklass till JComponent eller ett Game som är en subklass till Canvas, så när man navigerar mellan komponenter tas den gamla bort och en ny läggs till förutom från Game där den istället sparas undan i en variabel för att sedanvisas när man navigerar tillbaka till Game. Start som är en subklass till AComponent läggs till från början och innehåller några JButtons för att navigera till andra AComponents eller ett Game. HighScore är en subklass till AComponent som visar de olika poängen som satts av spelare. Submit är en annan subklass till AComponent som skapas av App genom Game med metoden gameOver().

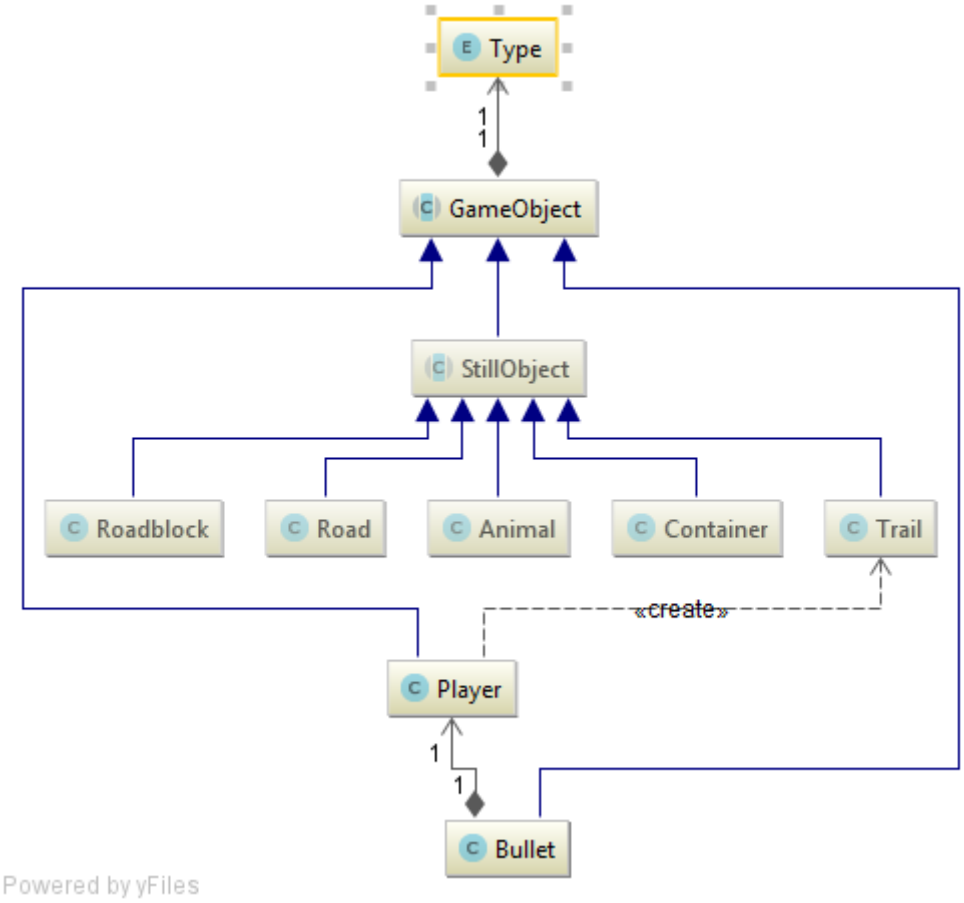


Både HighScore och Submit använder sig av HighScoreProperties för att hämta och registrera spelresultat. HighScoreProperties är en Singelton samt en subklass till AppProperties. I AppProperties finns ett filnamn på .properties-filen den hanterar samt en Property som i konstruktorn laddas med data från den filen. Det finns också metoden write() för att skriva över data i filen från Property och read() för att hämta data från filen till Property. Alla subklasser är Singeltons så att inte programmet kan hantera samma fil genom flera objekt helt omedveten om objektet har den senaste datan och används då genom getInstance(). HighScoreProperties sparar ner information i en HighScoreList i form av Score's. HighScoreList sköter sedan sorteringen av dessa. Påverkan på en HighScoreList sker genom HighScoreProperties. SavedProperties är en annan subklass till AppProperties och hanterar sparade Game. Genom denna kan man spara ner ett Game samt återskapa ett Game. Metoden saveGame() förlitar sig på att Game, Spawner, GameObjects och PowerUps returnerar all väsentlig information för att kunna återskapas genom metoden getSaveValues(). Till exempel använder Player i sin getSaveValues() metoden savePowerUps() i SavedProperties för att spara ner dess PowerUps. getGame() förlitar sig på att Game, Spawner, GameObjects och PowerUps har konstruktorer och metoder för att återskapa värdena de returnerat med getSaveValues(). Det kan vara viktigt att notera att alla metoder i SavedProperties endast är konstruerade för att bara ha ett spel sparat samt en spelare.



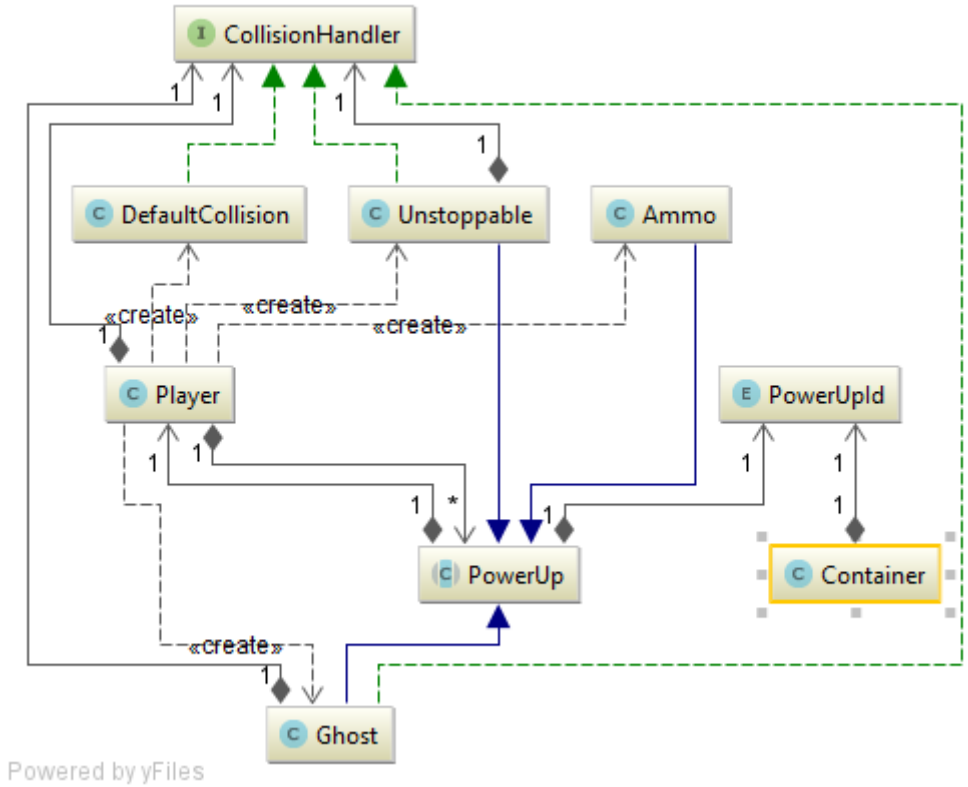
Game är en subklass till Canvas samt implementeras av Runnable så den kan köras som en tråd. Ett Game skapas huvudsakligen genom App med metoden newGame(). Metoden run() i Game som anropas när tråden skapas och startas med metoden start() innehåller en while-loop som kan stoppas av ett metodanrop till stop() som också stoppar tråden. I while-loopen anropas tick() beroende på variablen amountOfTicks och render() 60 fps. tick() är metoden som driver spelet framåt och från den bör metoder hos objekt i Game med samma uppgift anropas. render() är metoden som ritar upp allt i Game som vill ritas upp genom att anropa deras metoder med den uppgiften. Game innehåller en Spawner, två Handler och en Hud. Spawner's tick() är den första metoden att anropas från tick() då det är denna som lägger till GameObject's i de båda Handler. Sedan anropas Handler's tick() som i sin tur anropar alla deras GameObject's tick(). Det finns en Handler för GameObjekt's i bakgrunden kallad environment och en för GameObject's som påverkar spelet kallad handler. I render() anropas därför environment's render() först för att dess GameObject's ska ritas upp under handler's. I render() anropas också Hud's render(). Hud innehåller alla Player's i spelet och visar vilka PowerUps en Player har, liv kvar genom en bar och poäng. En annan betydelsefull funktion i while-loopen är att amountOfTicks ökar varje sekund med konstanten INCREASE vilket gör att tick() anropas mer och mer frekvent. Game skapar en Player i sin konstruktor, till denna skapas också en Controller som är en subklass till KeyAdapter som gör det möjligt för spelaren att styra Player då den läggs till i Game med addKeyListener(). Controller använder sig av en EnumMap med KeyAction och KeyCode för att veta vilken tangent som ska göra

vilken handling då KeyAction representerar varsin handling hos en Player. Controller har också en EnumMap med KeyAction och booleska variabler för att inte anropa samma metod flera gånger om en tangent är nertryckt.



GameObject är en abstrakt klass för objekt i spelet. Dessa hanteras i Game av en Handler. GameObject innehåller variabler och metoder för position, hastighet och utseende. Den innehåller också en Type som representerar varsin GameObject subklass och används i kollisionshantering samt återskapning. Ett GameObject innehåller också den Handler den befinner sig i eftersom den ska anropa sin Handler för att bli borttagen eller kollisionsdetektion. Metoderna collisionWithGameObject() och collisionWithPlayer() är speciella metoder som bara bör överskrivas om GameObjektet ska göra något som beror på det GameObjekt den kolliderat med eller tvärtom. Just collisionWithPlayer() används av Player när den kolliderar med ett GameObject som behöver använda Player's metoder som till exempel då en Player kolliderar med en Container och Container behöver kunna använda Player's metod newPowerUp(). StillObject är en abstrakt subklass till GameObject och har en tick() som gör att objektet på spelplanen beter sig som ett objekt som inte rör sig uppåt eller neråt dock eftersom spelet har en neråt rullande bakgrund med en hastighet på en pixel vartannat tick så följer ett StillObject den hastigheten. StillObjects behöver i vanliga fall ingen kollisionshanterare eftersom de är väldigt simpla och har bara en riktning. Animal som är en subklass till StillObject är ett mer avancerat hinder som innehåller en Timer som flyttar den i sidled i en jämn hastighet samtidigt som StillObjects tick() flyttar den neråt. Därför behöver den detektera och hantera kollisioner varje gång den flyttar sig i sidled. Bullet är en subklass till GameObject och flyttar sig i en riktning framåt med maximalhastighet och hanterar därför kollisioner. En Bullet fungerar som en förlängning av Player och innehåller därför den Player som skapade den så att den kan anropa collisionWithPlayer() hos det GameObject den kolliderade med om det är en fördel för Player. Trail är en subklass av StillObject och skapas av Player varje gång tick() anropas samt Player inte står still (velY != 1). Trail läggs oftast in i environment eftersom den inte ska påverka spelet utan bara skapa effekten av ett spår efter

Player.



Player är en subklass till GameObject, detta objekt kan styras av spelaren genom att Controller ändrar Player's hastigheter. Player sköter sin egen kollisionsdetektion, den kan men lite felmarginal lista ut vilken sida kollisionen skedde vilket representeras av en Side. Kollisionshantering sker dock genom en CollisionHandler. En CollisionHandler har en metod collision() denna anropas när det skett en kollision och tar det GameObject som Player kolliderat med som parameter. Eftersom en kollision kan påverka hastigheten på spelet alltså amountOfTicks så tar collision() också emot det Game som körs. En Player har en lista för PowerUp. En PowerUp skapas och läggs till av Player själv när den kolliderar med en Container som anropar Player's metod newPowerUp() och skickar med den PowerUpId som den bär på. Ett PowerUpId representerar varsin PowerUp och innehåller en färg, samt vilka andra PowerUps 's den inte kan kombineras med. När en ny PowerUp ska läggas till i en Player finns det tre scenarion, det första är att det bara läggs till och används tillsammans med de redan tillagda PowerUp's, det andra är att den har samma PowerUpId, alltså är likadana, då anropas hasSamePowerUp() hos den redan tillagda PowerUp eller så kan inte den nya PowerUp kombineras med en redan tillagd PowerUp då anropas metoden interrupt() hos den tillagda och tas bort från Player samt läggs till i en lista med avbrutna PowerUp's kallad interrupted i den nya PowerUp som läggs till hos Player. När en PowerUp tar slut anropas reset(), då läggs de PowerUp som blivit avbrutna till hos Player om de kan kombineras med de redan tillagda eller förblir avbrutna hos det PowerUp som de inte kan kombineras med genom metoden addInterruptedPowerUp. I reset() kallas också interrupt() för att återställa Player. Eftersom en PowerUp antingen ska påverka Player direkt eller genom spelarens kommando så finns det olika metoder som PowerUp's kan använda till det. För att kunna skapa PowerUp's som inte börjar förbrukas när de skapas av SavedProperties används resume() istället för att aktivera PowerUp's, resume() anropas därför varje gång en PowerUp läggs till i Player. För PowerUp's som vill användas på spelarens kommando finns use() och stop() som anropas indirekt genom Player av Controller. Unstoppable som är en subklass till PowerUp samt implementeras av CollisionHandler använder resume() för att sätta Player's CollisionHandler till sig själv. Medans Ghost som är en subklass till PowerUp samt implementeras av CollisionHandler använder use() för att sätta Player's CollisionHandler till sig själv.

6.3. Användning av fritt material

Ingen användning av fritt material har gjorts.

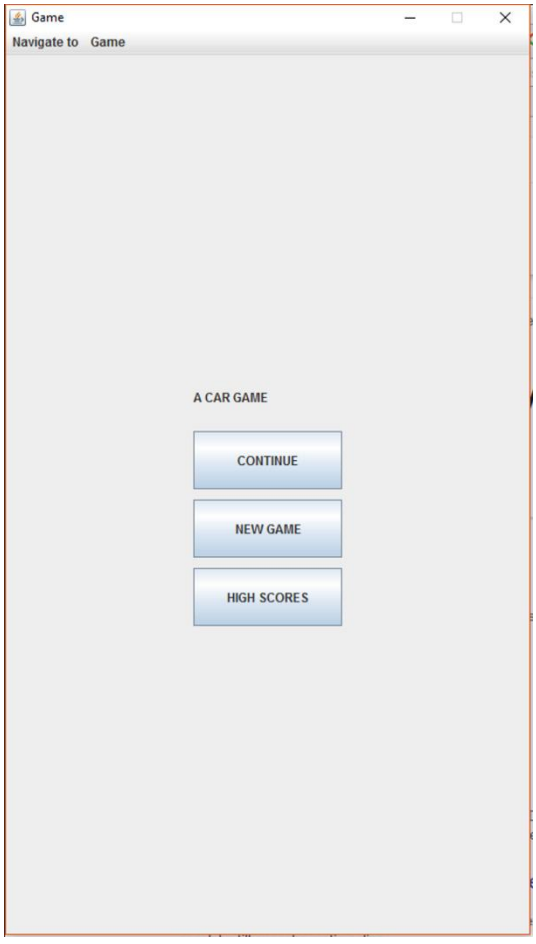
6.4. Användning av objektorientering

1. Abstrakta klasser, AComponent, jag har blivit av med upprepande kod då alla skulle överskriva getPrefferedSize(), jag har tydligare parat ihop en AComponent med en State. Jag skulle kunna ha uppnått samma sak med att direkt ärva från JComponent men då hade jag behövt hålla reda på vilken State som hör till vilken JComponent i App.
2. Gränssnitt, CollisionHandler, hjälper mig byta ut hur jag vill hantera kollisioner hos Player. Jag skulle kunnat skapa alla olika CollsionHandler som metoder i Player och använt en switch för att bestäma vilken Player ska använda. Detta skulle dock gjort Player onödigt stor och det skulle bli krångligt att skapa fler metoder för att hantera kollision.

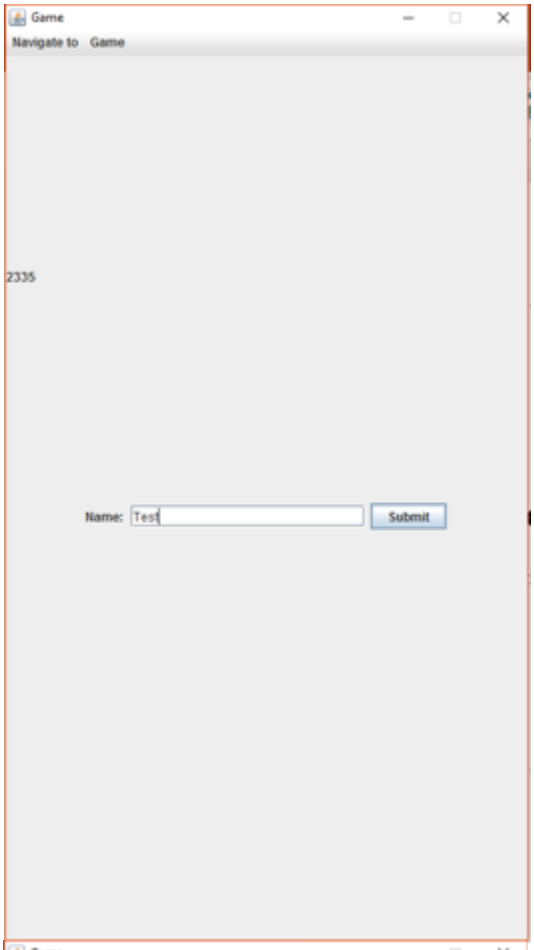
6.5. Motiverade designbeslut med alternativ

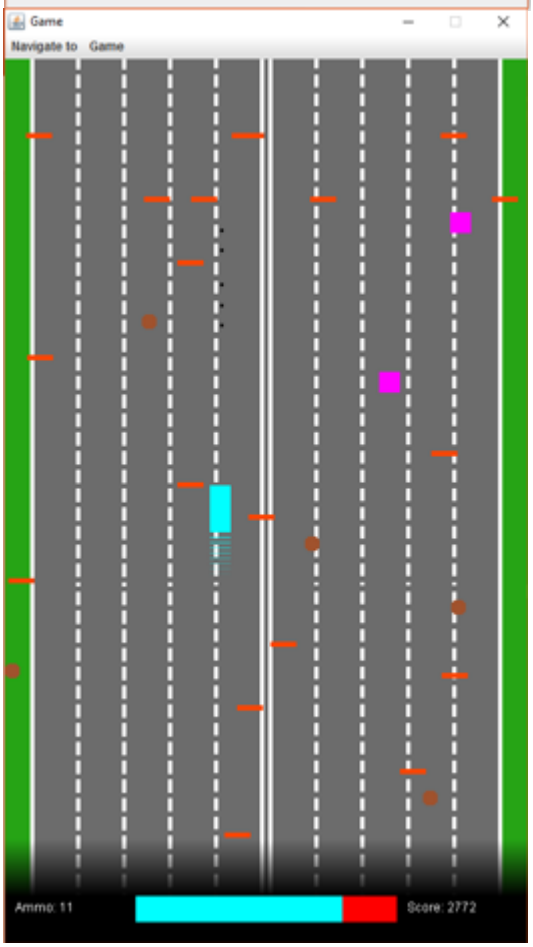
1. Att göra Game till en subklass av Canvas istället för AComponent var ett beslut som togs för exempel jag hittade på spelmotorer använde det samt att det ska vara en bättre rendering genom BufferStrategy dock i efterhand tror jag det skulle varit enklare att gjort den till en subklass till AComponent då jag är osäker om BufferStrategy hade behövts eftersom jag kör render i 60 fps, samt att det hade förenklat mycket kod i App och Game som rör bytet mellan komponenter.
2. Att använda HighScoreList för att hantera Scores i HighScoreProperties var ett bra designbeslut för lättläslighet och sortering dock hade samma sak kunna uppnåtts med en lista för scores, en Comperator för att sortera samt få tag i indexet för det nya elementet med metoden indexOf. Det hade kanske sett bättre ut.

7. Användarmanual



Programmet startas via main-metoden i App. Det första som kommer upp är startmenyn samt en meny längst upp i fönstret. Genom knappen CONTINUE kan du fortsätta spelet du spelar, återskapa ditt senast sparade spel eller om du inte har något spelar något spel eller har något sparat kan du skapa ett nytt spel. Denna knapp motsvarar Continue under menyn Game samt kommandot Ctrl-C. Genom knappen NEW GAME kan du skapa ett nytt spel. Denna knapp motsvarar Newunder menyn Game samt kommandot Ctrl-N. Genom knappen HIGH SCORES kan du se alla sparade resultat från tidigare spel. Denna knapp motsvarar High Scores under menyn Navigate to samt kommandot Alt-H. Du kan navigera tillbaka till startmenyn med knappen Start under menyn Navigate to eller kommandot Alt-S. I spelet kan du pausa och fortsätta spelet med knappen Paus/Resume under menyn Game eller kommandot Ctrl-P. I spelet kan du spara spelet med knappen Save under menyn Game eller kommandot Ctrl-S. Varje gång du går till ifrån spelet till start eller high score listan pausas spelet.





Om du dör i spelet kommer du till en registreringssida där du ska skriva in ditt namn och trycka på knappen Submit för att lägga till resultatet i high score listan. Continue kommer då inte ta dig till ett nytt spel om du inte registrerat dig dock kommer new göra det.

Spelet går ut på att köra en bil så långt som möjligt genom att undvika hinder och använda power ups. Bilen kommer vara en turkos rektangel som du styr med piltangenterna, bilen kommer åka framåt av sig själv, bakåtpilen betyder att du stannar och då kommer du inte kunna styra, framåtpilen gör att du åker dubbelt så fort fram och med sidopilarna svänger du. Det finns två sorters hinder, vägspärr som är en orange rektangel och rådjur som är en brun cirkel. När du kör in i ett hinder förlorar du liv och fart. Om du frontalkrockar kommer du ta mer skada än när du krockar med ett hinder från sidan.

Power ups är färgglada kvadrater. Du plockar upp en power up genom att krocka med den. Rosa power up kallas Ghost den gör att du kan åka genom objekt utan att ta skada så länge du håller ner space. Du får 5 sekunder av denna power up. Gul power up kallas Unstoppable och gör att du kan åka på objekt utan att ta skada samt få extra poäng för det den går dock inte kombinera med Ghost. Denna power up börjar gälla direkt och börjar räkna ner från 5 i sekunder direkt när du plockar upp den. Röd power up kallas Ammo och gör att du kan skjuta bort hinder och plocka upp power ups med hjälp av kulor du skjuter framåt med space. Den kan inte kombineras med Ghost. Du får 10 skott. Om du har en power up och plockar upp en power up som är likadan får du dess innehåll. Om du har en power up och plockar upp en power up som inte kan kombineras så läggs den gamla åt sidan och ersätts tills den nya tagit slut och den inte hindras av att den inte kan kombineras med de power ups som finns.

8. Slutgiltiga betygsambitioner

Jag har ambitioner för betyg 3.

9. Utvärdering och erfarenheter

Detta avsnitt är en väldigt viktig del av projektspecifikationen. Här ska ni tänka tillbaka och utvärdera projektet (något som man alltid bör göra efter ett projekt). Som en hjälp på vägen kan ni utgå från följande frågeställningar (som ni gärna får lämna kvar i texten så att vi lättare kan sammanställa information om specifika frågor):

- *Vad gick bra? Mindre bra?*
    - Det har gått väldigt bra att skriva kod
    - Det har gått bra att lösa problem
    - Det har gått bra att hitta information och kunskap
    - Det har gått mindre bra med slutinlämningen
    - Det har gått mindre bra med dokumentationen
- *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har "hjälpt" bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!*
    - *Google*
    - *Föreläsningar*
- *Har ni lagt ned för mycket/lite tid?*
    - I perioder har det blivit mycket annars inget så lagom tror jag
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
    - Ja
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Milstolpar
    - Övriga implementationsförberedelser
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
    - Det har fungerat bra
- *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*
    - Att förstå uppgifterna i slutuppgiften
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
    - Börja tidigare med slutuppgiften så du kan fråga handledaren om råd
- *Har ni saknat något i kursen som hade underlättat projektet?*
    - Exempel på projekt man skulle kunna ta sig ann.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
    - Nej