TDDC74 – Projektspecifikation

Projektmedlemmar:

Oscar Göransson oscgo580@student.liu.se

Edvin Ljungstrand edvlj182@student.liu.se

Handledare:

Linnea Faxén linfa440@student.liu.se

25 maj 2016

Innehåll

1 Projektplanering

Projektets syfte är att konstruera ett så kallat shoot ’em up-spel i vilket spelaren kontrollerar en rymdfarkost och har som uppgift att förstöra asteroider samt anfallande ufon. Spelet kontrolleras med tangentbordet och stödjer upp till två spelare.

1.1 Kort projektbeskrivning

Spelet är baserat kring en fast skärm till vilken det spelarstyrda rymdskeppet är begränsat. Spelaren kan rotera skeppet i alla riktningar men endast gasa och skjuta i den riktning skeppet är vänt. Skeppet har en viss tröghet vilket innebär att då det försatts i rörelse i någon riktning kommer det att förbli i rörelse tills dess att spelaren ingriper, dvs. gasar i någon annan riktning, eller tills skeppet stannar på grund av friktion. I början av varje omgång kommer ett visst antal stora asteroider flyta omkring på skärmen och om dessa blir beskjutna av spelaren kommer de att splittras i mindre bitar, vilka är snabbare än de stora asteroiderna. Enligt någon regel kommer även ufon att visa sig på skärmen och dessa kan, till skillnad från asteroiderna, anfalla spelaren. Att förstöra asteroider och ufon belönar spelaren med poäng och spelets mål är att samla så många poäng som möjligt. Alla objekt kan röra sig mellan skärmens ändar, så ett objekt som rör sig ut från skärmen i någon ände kommer att röra sig in på skärmen fån motsatt ände med sin fart bevarad.

1.2 Utvecklingsmetodik

Kommunikation kommer att ske regelbundet via flera olika kanaler. Koden kommer versionshanteras samt delas via Git. Arbetet kommer främst ske på schemalagd tid men då ytterligare tid krävs kommer arbete även att ske utanför den schemalagda tiden. Projektet uppskattas omfatta cirka tio timmar per vecka under åtta veckors tid.

1.3 Grov tidplan

Arbetet kommer till att börja med att ske i grupp men delas upp mellan parterna enligt behov. Till halvtidsmötet ska åtminstone ett kontrollerbart spelarstyrt skepp och rörliga asteroider, i någon form, vara implementerade. Vecka 19 ska åtminstone alla obligatoriska krav vara klara och vecka 20 planeras ägnas åt eventuell finputsning och optimering samt åt förberedelse av redovisning.

1.4 Betygsambitioner

Projektet ämnar uppnå kraven för betyg 5.

2 Användarmanual

För att starta spelet öppnas main.rkt i den senaste versionen av DrRacket. Skriv (start-game) och tryck på enter. Välj sedan om du vill spela en eller två spelare i den grafiska menyn.

Kontroller för spelare:

Funktion	Spelare 1	Spelare 2
Gasa	W	I
Rotera vänster	A	J
Rotera höger	D	L
Skjut	Space	Enter
Teleportera	Left Shift	Backspace

Tabell 1 Tangentkommandon för spelarna.

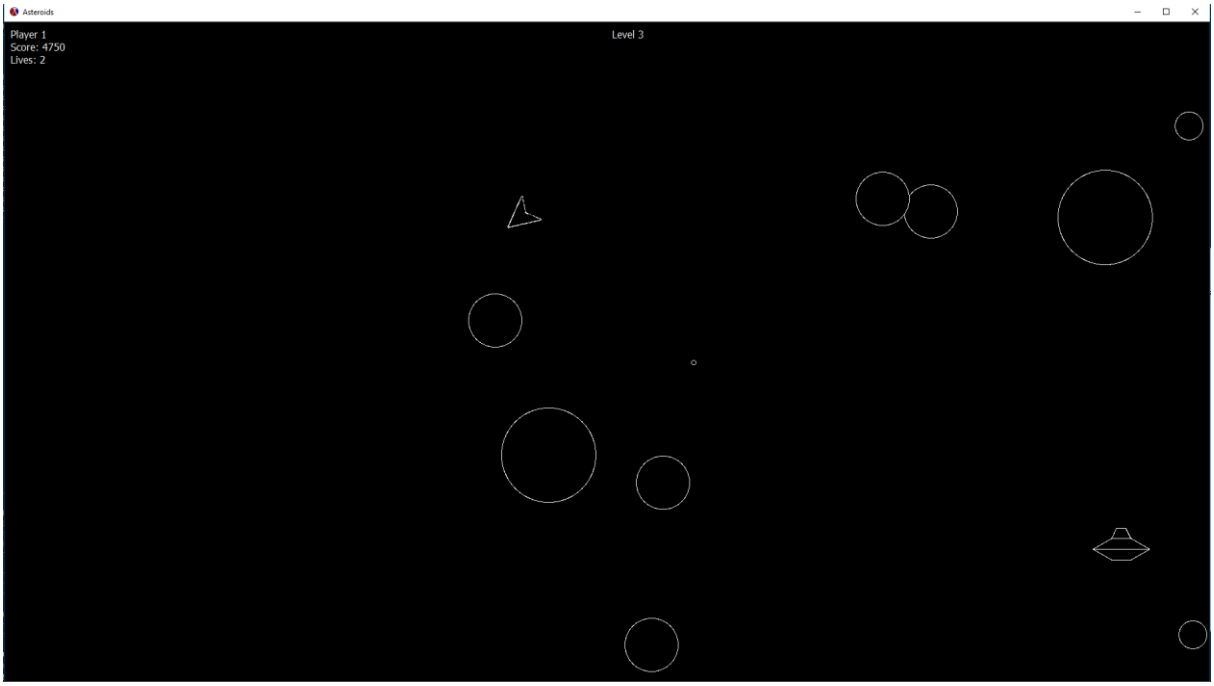
Spelet kan när som helst avslutas med escape.

Spelet går ut på att få så många poäng som möjligt samt att komma till så hög nivå som möjligt. Spelaren får poäng då denne skjuter sönder asteroider eller rymdskepp och man kommer till nästa nivå genom att ha sönder samtliga asteroider på spelplanen. Två spelare kan tävla om vem som kan få flest poäng.

Då en spelare kolliderar med något objekt och inte har några återstående liv dör spelaren. Då samtliga spelare har dött avslutas spelet och spelarnas poäng visas på skärmen.

Objekt	Poäng
Stor asteroid	50
Mellanstor asteroid	75
Liten asteroid	100
UFO	200

Tabell 2 De poäng olika objekt är värda.



Figur 1 Skärmbild från den färdiga produkten.

2.1 Kravlista

#	Beskrivning	Prioritet
1	Spelet ska startas från DrRacket med kommandot (play-game)	1
2	Spelet ska kunna avslutas genom att klicka på Esc	1
3	Spelaren ska kunna förflytta sig med tangentbordskommandon	2
4	Spelaren ska ha ”tröghet”	2
5	Asteroider ska finnas	3
6	Asteroider ska röra sig	4
7	Spelaren skall kunna skjuta med tangentbordskommandon	4
8	Mindre asteroider rör sig snabbare än större asteroider	5
9	Asteroider ska delas upp i mindre asteroider då de beskjuts	5
10	Objekt kan passera genom skärmens kanter och komma ut på motsatt sida med konstant riktning och fart	6
11	Spelaren upplever friktion	6
12	Motståndare, ufon skall finnas	7
13	Ufon skall komma in i spelet periodiskt	7
14	Ufon skall kunna skjuta	8
15	Ufon ska försvinna efter en viss tid	8
16	Spelare ska ha liv	9
17	Då en spelare krockar med annat objekt eller blir beskjuten förlorar spelaren ett liv	9
18	Spelet har stöd för två spelare	9
19	Med ett tangentbordskommando skall spelaren kunna förflytta sig till slumpmässig plats på spelplanen	10
20	Spelet avslutas när alla spelare har slut på liv	11
21	Man ska kunna välja mellan en eller två spelare	12

22	Det ska finnas rundor i spelet då det kommer nya asteroider	13
23	En meny	14
24	Antalet asteroider ökar varje runda	15
25	Ufona blir mer pricksäkra för varje runda	16
26	Poängsystem där man får poäng för varje skjuten asteroid eller ufo	17
27	Power-ups	18

Krav med prioritet 1 till och med 13 skall vara uppfyllda medan krav med högre prioritetsnummer genomförs om möjlighet finns.

3 Implementation

3.1 Abstrakta datatyper eller klasser

Klass	Beskrivning
asteroid%, medium-asteroid% och small-asteroid%	Klasser för asteroidobjekt vilka har egenskaper som position, storlek och hastighet.
ship%	Klass för spelarkontrollerade objekt. Innehåller information om egenskaper som, till exempel, antal liv, position samt riktning. Har metoder för styrning och för att kunna skjuta.
ufo%	Klass för ufo-objekt. Har egenskaper som t.ex. position, fart, riktning och träffsäkerhet. Har metod för att kunna skjuta.
bullet%	Klass för de kulor som skjuts av spelarkontrollerade objekt samt ufon.

3.1.1 asteroid%, medium-asteroid% och small-asteroid%

Det finns tre olika klasser för asteroider: asteroid%, medium-asteroid% och small-asteroid%, där medium- och small-asteroid% ärver från asteroid%. De har olika värden på vissa fält men samma eller snarlika metoder. Där metoderna skiljer sig åt har det beskrivits hur.

Metod	Beskrivning
get-image	Returnerar värdet på fältet image, en bitmap på vilken asteroiden kan ritas ut.
ceate-asteroid-image bitmap-target	Ritar ut en asteroid på bitmap-target, en bitmap.
get-mid-xpos	Returnerar fältet mid-xpos, x-koordinaten för asteroidens mittpunkt.
get-mid-ypos	Returnerar fältet mid-ypos, y-koordinaten för asteroidens mittpunkt.
destroy! name	För asteroid% och medium-asteroid% skapas nya instanser av medium- respektive small-asteroid% och sedan tas asteroidobjektet bort från asteroids-hash, vilket innebär att objektet inte längre kommer att uppdateras. För small-asteroid% tas objektet endast bort från asteroids-hash.
obj-has-collided-with obj	Bestämmer vad som händer när asteroiden har kolliderat med något annat objekt, obj. Vid kollision anropas destroy! med fältet name som argument.
set-mid-x! new-mid-x	Beräknar och sätter ett nytt värde på x-pos då fältet mid-xpos uppdaterats till new-mid-x.
set-mid-y! new-mid-y	Beräknar och sätter ett nytt värde på y-pos då fältet mid-ypos uppdaterats till new-mid-y.
update! dc	Beräknar asteroidens nya position samt ritar ut asteroiden på dc, en drawing context.

3.1.2 ship%

Metod	Beskrivning
get-image	Returnerar värdet på fältet image, en bitmap på vilken skeppet ritas ut.
get-score	Returnerar fältet score i vilket skeppets poäng lagras.
update-score amnt	Uppdaterar fältet score med amnt poäng.
obj-has-collided-with obj	Gör skeppet odödligt i 0,5 s samt sätter ny position för skeppet. Om skeppet har slut på liv anropas destroy!-metoden.
destroy! name	Tar bort skeppet med nyckeln name från den hashtabell, ship-hash, i vilken alla skepp som ska uppdateras finns. Lägger sedan till skeppet med nyckeln name i den hashtabell där döda skepp lagras.
create-ship-image bitmap-target	Ritar ut ett skepp på bitmap-target, en bitmap.
get-mid-xpos	Returnerar fältet mid-xpos, x-koordinaten för skeppets mittpunkt.
get-mid-ypos	Returnerar fältet mid-ypos, y-koordinaten för skeppets mittpunkt.
set-mid-x! new-mid-x	Beräknar och sätter ett nytt värde på x-pos då fältet mid-xpos uppdaterats till new-mid-x.
set-mid-y! new-mid-y	Beräknar och sätter ett nytt värde på y-pos då fältet mid-ypos uppdaterats till new-mid-y.
accelerate	Ökar hastigheten hos skeppet om det inte redan rör sig i sin maximala hastighet.
turn-left och turn-right	Roterar skeppet mot- respektive medurs.
fire	Skapar en instans av bullet% vid skeppets topp.
hyperdrive	Teleporterar spelaren till en slumpmässig plats på skärmen.
steer	Kontrollerar inkommandon och styr därefter skeppet genom att köra andra funktioner.
update! dc	Beräknar skeppets nya position, lyssnar efter tangentkommandon samt ritar ut skeppet på dc, en drawing context.

3.1.3 ufo%

Metod	Beskrivning
get-mid-xpos	Returnerar fältet mid-xpos.
get-mid-ypos	Returnerar fältet mid-ypos
set-mid-x! new-mid-x	Beräknar och sätter ett nytt värde på x-pos då fältet mid-xpos uppdaterats till new-mid-x.
set-mid-y! new-mid-y	Beräknar och sätter ett nytt värde på y-pos då fältet mid-ypos uppdaterats till new-mid-y.
fire	Skapar en instans av bullet%. Dessa kommer att färdas mot något skepp och blir mer pricksäkra för varje ny nivå. Anropas periodiskt av en timer som startas då ufot skapas.

obj-has-collided-with obj	Bestämmer vad som händer när ufot har kolliderat med något annat objekt, obj. Anropar då destroy! med sitt fält name som argument.
destroy! name	Tar bort ufot med nyckeln name från den hashtabell, ufo-hash, i vilken alla ufon som ska uppdateras finns.
create-ufo-image bitmap-target	Ritar ut ett ufo på bitmap-target, en bitmap.
update! dc	Beräknar ufots nya position, minskar ufots liv, samt ritar ut ufot på dc, en drawing context.

3.1.4 bullet%

Metod	Beskrivning
get-image	Returnerar värdet på fältet image, en bitmap på vilken ufot ritas ut.
get-mid-xpos	Returnerar fältet mid-xpos, x-koordinaten för kulans mittpunkt.
get-mid-ypos	Returnerar fältet mid-ypos, y-koordinaten för kulans mittpunkt.
set-mid-x! new-mid-x	Beräknar och sätter ett nytt värde på x-pos då fältet mid-xpos uppdaterats till new-mid-x.
set-mid-y! new-mid-y	Beräknar och sätter ett nytt värde på y-pos då fältet mid-ypos uppdaterats till new-mid-y.
obj-has-collided-with obj	Bestämmer vad som händer när kulan kolliderar med något objekt, obj. Om kulan förstör obj anropas update-score hos det skepp som skapat kulan med poängen för obj som argument. Anropar sedan destroy! med sitt fält name som argument.
destroy! name	Tar bort kulan med nyckeln name från den hashtabell, bullets-hash, i vilken alla kulor som ska uppdateras finns.
update! dc	Beräknar kulans nya position samt ritar ut kulan på dc, en drawing context.

3.2 Testning

Alla funktioner kontrolleras så att de uppfyller kraven i kravspecifikationen då de implementerats. Åtminstone fyra större tester kommer att genomföras då kompatibilitet mellan alla implementerade funktioner undersöks. Dessa har för avsikt att kontrollera att produkten uppfyller projektambitionen, så som den beskrivits under rubrik 1.1, i den mån produkten kan förväntas göra det i sitt dåvarande utvecklingsstadium. Större tester kommer att ske

1. Inför halvtidsmötet.
2. Då de obligatoriska kraven är uppfyllda.
3. Då eventuella ytterligare krav är uppfyllda.
4. Inför slutleverans.

Ett större test tar formen av en längre tids koncentrerat spelande där eventuella ytterlighetsscenarion, såsom oväntade tangentbordskommandon, grundligt kontrolleras. Ett

sådant test torde ge en rättvisande bild av hur väl produkten kan förväntas fungera vid normalt såväl som onormalt användande.

3.3 Beskrivning av implementationen

Då spelet startas skapas så många spelarobjekt som valts i spelmenyn och sedan anropas en initialiseringsprocedur, *init-level*, i main.rkt som skapar asteroidobjekt och startar en timer som periodiskt skapar ufo-objekt. Därefter startas en timer som periodiskt uppdaterar skärmen. Då skärmen uppdatera anropas procedurerna *render* och *on-level-over*.

Proceduren *render* ansvarar för att rita ut objekt på skärmen. Huruvida ett objekt ska ritas ut avgörs av om de ingår i särskilda hashtabeller som lagras i utilities.rkt. Då ett objekt skapas lägger det till sig självt i en hashtbabell innehållande alla objekt av samma klass. Då *render* anropas adderas varje värde i alla objekthashtabeller till en lista. För varje element i listan anropas metoden *update!* som beräknar objektens nya position och ritar ur dem, vilket får det att se ut som att de rör sig. Då objekt inte längre ska ritas ut, t.ex. då ett ufo dör, tas det bort från sin hashtabell vilket innebär att *update!* inte kommer att anropas för objektet.

Kulors, ufons och asteroiders rörelse beskrivs av två parametrar: förflyttning i x-led och förflyttning i y-led. Då ett objekt rör sig adderar *update!* förflyttning i x- och y-led till de nuvarande x- och y-koordinaterna och dessa uppdateras. Objektet ritas sedan ut på de nya koordinaterna och detta ger intrycket av att objekten rör sig kontinuerlig. Om ett objekt rör sig utanför skärmen kommer det att flyttas tillbaka in på skärmen, men på motsatt sida. Denna funktionalitet tillhandahålls av proceduren *screen-wrap*, i utilities.rkt, som anropas av *render* för alla objekt och som kontrollerar om objektens x- eller y-koordinater är större än vad skärmen är stor i x- respektive y-led. Då skärmen spelaren rör sig på är begränsad av att den är statisk är detta nödvändigt för spelbarheten.

Då spelare ska kunna kontrollera skepp krävs det att *update!*-metoden i ship.rkt inte bara uppdaterar skeppens position enligt någon regel utan aktivt lyssnar efter tangenttryck. Det åstadkoms genom att utnyttja canvas%-objekts förmåga att reagera på tangentbordshändelser: Om en tangent trycks ned anropar *asteroids-game-canvas* i main.rkt proceduren *key-handler* i utilities.rkt som sätter värdet för den nedtryckta tangenten till att var sant i hashtabllen key-hash (och om tangenten släpps sätts det till att vara falskt). Metoden *update!* i ship.rkt kontrollerar sedan om de tangenter som angetts kontrollera skeppet är nedtryckta och om de är det anropas de metoder som hanterar rörelse. Att lagra tangenttryck som sanningsvärden i en tabell visade sig vara extremt viktigt för hur precisa och användarvänliga kontrollerna upplevdes vara; då denna metod inte tillämpades kändes det otympligt att kontrollera skeppet då det inte reagerade på tangenttryck särskilt snabbt samt rörde sig ”hackigt”. Det var då dessutom inte möjligt att registrera simultana tangenttryck vilket innebar att det inte gick att t.ex. gasa och skjuta samtidigt, något som är av yttersta vikt i ett spel av detta slag.

När vissa tangenter är nedtryckta kommer, som sagt, *update!* i ship.rkt att anropa de metoder som hanterar rörelse. Då rotationsmetoderna anropas ökar, alternativt minskar, den vinkel, angle, skeppet anses ha gentemot någon viloposition. Skepp kan då ritas ut genom att roteras runt sina centrum med angle radianer. Om metoden som kontrollerar gasen anropas ökar variabeln speed och förflyttningsvariablerna beräknas genom ett trigonometriskt uttryck. De

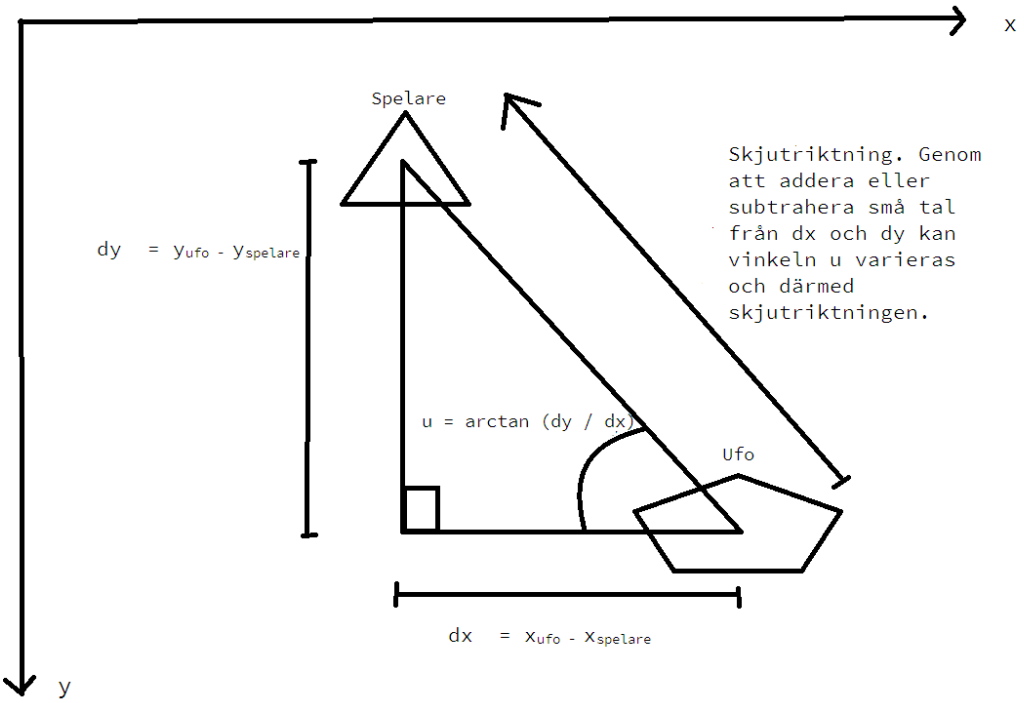
adderas sedan till skeppets x- och y-koordinat, precis som för andra objekt. Skepp har även förmågan att skjuta, vilket här innebär att då ett skepps skjutmetod anropas så skapas ett kulobjekt, vilket av skeppet förses med tillräcklig information för att kulobjektet ska kunna rita upp sig framför skeppet, som sedan färdas rakt fram tills det kolliderar med annat objekt eller försvinner; kulobjekt kan försvinna av sig själva genom att efter en viss tid anropa metoden *destroy!* i bullet.rkt. Att kulobjekten försvinner efter en viss tid är förstås nödvändigt för att spelet ska vara utmanade liksom för att det, om kulorna blir alltför många, inte ska krascha.

Det är även i *render* som kollisioner hanteras. Varje gång *render* anropas kontrolleras om något objekt kolliderat med något annat objekt med proceduren *collision?* i utilities.rkt. Vid kollision anropas metoden *obj-has-collided-with* för båda objekten och varje objekt avgör själv vad som ska hända med det. Då ett kulobjekt kolliderar med ett annat objekt undersöker *obj-has-collided-with* i bullet.rkt vem som skapat kulan. Om det visar sig vara en spelare anropas metoden *update-score*, i ship.rkt, för spelaren och det andra objektets points-fält sätts som argument.

Rörande kollisionsdetekteringen var vår tanke att för varje objekt beskriva dess gränsyta med något uttryck och sedan lagra alla de punkter som uppfyllde uttrycket i en lista, alternativt en vektor. Vi tänkte sedan söka igenom och jämföra alla objekts listor efter gemensamma punkter, där en gemensam punkt (x, y) skulle innebära att objekten kolliderat. Detta visade sig svårt, främst då gränsytorna för skepp och ufon inte var särskilt enkla att beskriva. Vid konsultation med handledare föreslogs det att vi istället kunde skapa en, så kallad, hitbox av enkel karaktär, t.ex. en cirkel, och sedan för varje objekt kontrollera om avstånden mellan objektens centra var mindre eller lika med avståndet mellan hitboxarnas radier. Nackdelen med denna metod är förstås att kollisionsdetektionen blir mindre exakt, men vid testning visade sig felet vid all kollision vara mer eller mindre försumbart och då det var mycket mer tidseffektivt valde vi att beskriva kollision med cirkulära hitboxar.

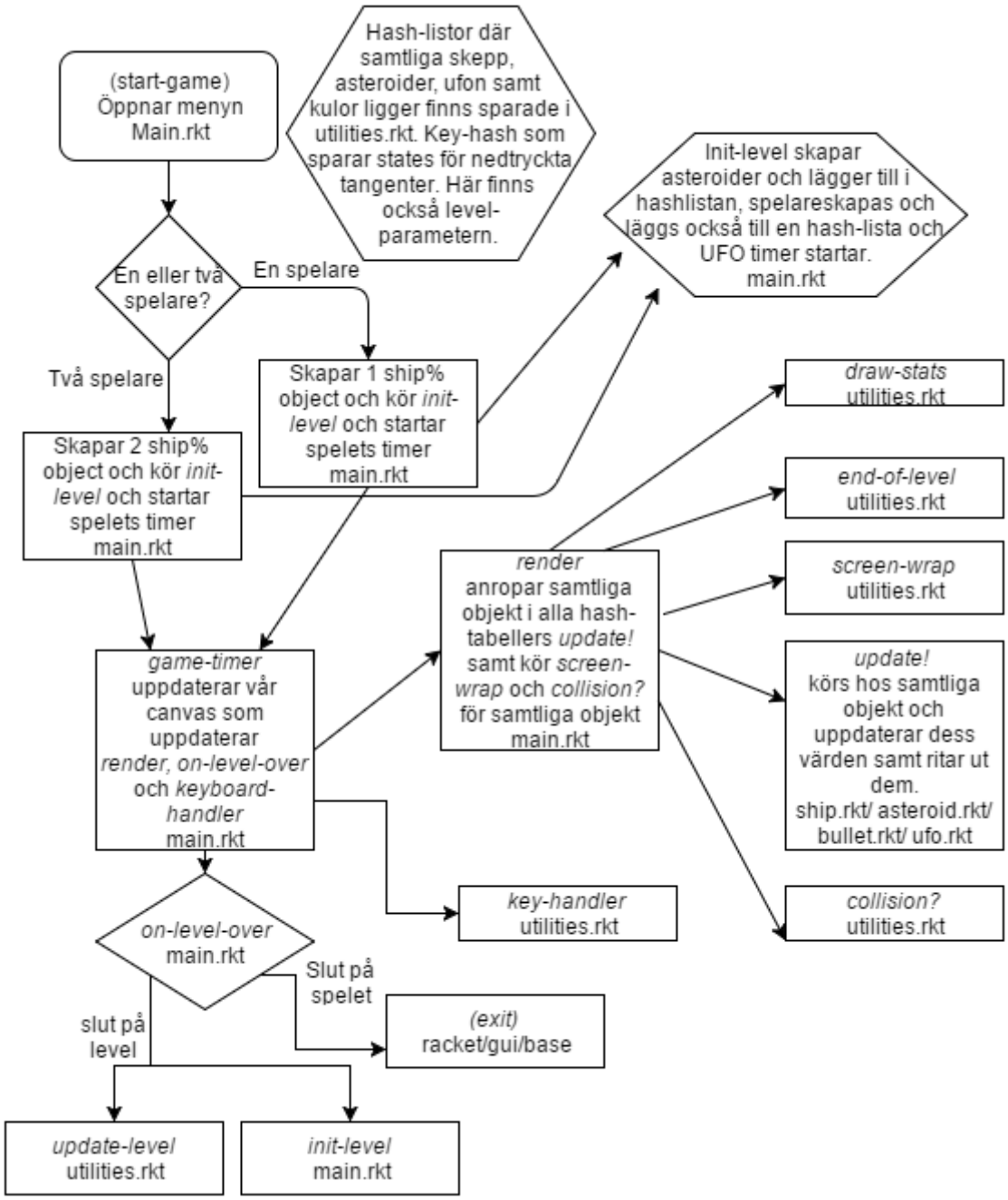
Proceduren *on-level-over* är ansvarig för att detektera när spelet eller den nuvarande nivån är över, vilket den gör genom att testa om hashtabellerna som innehåller spelarobjekt respektive asteroidobjekt är tomma. Om spelet bedöms vara över så anropas *exit*, som förses av racket/gui/base, eller om nivån är över så uppdateras variabeln level i utilities.rkt och sedan anropas *init-level* med den uppdaterade level som argument.

För att spelet ska bli svårare allteftersom det fortgår skapar *init-level* fler asteroider varje nivå och ufons träffsäkerhet är proportionell mot nivån. Att ufons träffsäkerhet ökar innebär att de kulobjekt som de skapar (ufon skapar kulobjekt på samma sätt som spelarobjekt gör) färdas längs en linje som rör sig närmare något spelarobjekts mittpunkt. Om den linjen ses som hypotenusan i en rätvinklig triangel kan linjen dras godtyckligt nära något spelarobjekts mittpunkt genom att variera katetrarna i triangeln, dvs. genom att addera eller subtrahera allt mindre tal från differenserna mellan ufots och spelarobjektets x- respektive y-koordinater. Ett annat sätt att göra ufon mer utmanande hade t.ex. varit att låta deras kulor färdas snabbare för varje nivå, men vi valde att inte göra detta då vi bedömde att då träffsäkerheten blir god efter relativt få nivåer så hade det varit alltför svårt. Hur ofta ufon skjuter regleras av en timer, *ufo-fire-timer*, i ufo.rkt, som startas då ufo-objektet skapas. Även denna timer hade kunnat modifieras för att vara proportionerlig mot nivån men idén förkastades av samma skäl som snabbare kulor gjorde.



Figur 2 Skiss av hur ett ufo skjuter.

Antalet asteroider för en nivå beräknas genom (nivå + 2) vilket förefaller vara en skapligt långsam ökning men då varje stor asteroid, dvs. asteroid%-objekt, skapar två medium-asteroid%-objekt och varje medium-asteroid%-objekt ger upphov till två small-asteroid%-objekt är det maximala antalet small-asteroid%-objekt för en nivå (4 (nuvarande nivå + 2)) = (4*nivå + 8) vilket vi bedömde vara utmanande för åtminstone de högre nivåerna.


(start-game)
Öppnar menyn
Main.rkt
Hash-listor där samtliga skepp, asteroider, ufon samt kulor ligger finns sparade i utilities.rkt. Key-hash som sparar states för nedtryckta tangenter. Här finns också level-parametern.
Init-level skapar asteroider och lägger till i hashlistan, spelareskapas och läggs också till en hash-lista och UFO timer startar.
main.rkt
En eller två spelare?
En spelare
Två spelare
Skapar 1 ship% object och kör init-level och startar spelets timer
main.rkt
Skapar 2 ship% object och kör init-level och startar spelets timer
main.rkt
draw-stats
utilities.rkt
end-of-level
utilities.rkt
screen-wrap
utilities.rkt
render
anropar samtliga objekt i alla hash-tabellers update! samt kör screen-wrap och collision? för samtliga objekt
main.rkt
update!
körs hos samtliga objekt och uppdaterar dess värden samt ritar ut dem.
ship.rkt/ asteroid.rkt/ bullet.rkt/ ufo.rkt
collision?
utilities.rkt
game-timer
uppdaterar vår canvas som uppdaterar render, on-level-over och keyboard-handler
main.rkt
key-handler
utilities.rkt
on-level-over
main.rkt
Slut på spelet
slut på level
(exit)
racket/gui/base
update-level
utilities.rkt
init-level
main.rkt

4 Utvärderingar och erfarenheter

Projektet har varit väldigt utvecklande men också utmanande. Det projekt vi gjort har involverat att använda Rackets grafiska bibliotek vilket har varit nytt för oss och som bitvis har varit ganska så klurig. Att få en inmatningsmetod som fungerar bra har också haft vissa utmaningar. Att ha en projektspecifikation att luta sig tillbaka på där vi vet vad vi skall uppnå har varit bra. Att vi här också skrivit ned vissa procedurer som vi tror vi behöver implementera har också det varit bra, även om vi inte använt speciellt många av de vi skrev ned från början hade vi ändå något att börja på och jobba utifrån vilket gjort att vi kommit framåt och utvecklat våra idéer allt eftersom.

Vi har lagt en väl avvägd mängd tid känns det som i dagsläget, vi jobbade på ganska så mycket hela tiden för att ha överblick och marginal vilket gjort att vi kunnat planera vår tid mot slutet då vi kunnat uppskatta hur mycket arbete det varit kvar. Vi har under projektet haft en bra dialog mellan oss och samarbetet har fungerat väl. Github har använts vilket också har varit något nytt och lite klurigt men som också visat sig vara ett mycket bra verktyg för att samarbeta under projektet. Att arbeta både enskilt och tillsammans har fungerat bra för att diskutera fram idéer medan vi varit tillsammans och gjort vissa av dessa för att sedan gå hem och jobba på var för sig och sedan följa upp och diskutera igen.

Under projektet har vi fått en större insikt i vad det innebär att arbeta samtidigt på samma projekt, hur viktigt det är att ha en bra dialog och att ha bra kanaler att dela kod på samt en insikt i att det är en stor fördel att kunna sitta tillsammans och diskutera fram lösningar. Också hur man bygger upp ett större projekt som inte är uppstyrt från början har vi lärt oss mycket om. Det var inte självklart från en början utan något som vi varit tvungna att utveckla under projektets gång, något som krävt många omarbetningar av kod och filstruktur för att få till det bra.

5 Tidrapportering

5.1 Edvin

Vecka	Arbetsuppgift	Tid (h)
14		
15	Kravspecifikation	3
16	ship%, bullet%, game mm.	11
17	Screenwrap, mer på klasserna, skjuta bland annat.	12
18	Uppdatering ändrad, forts arbete på klasser för att få bra parametrar, hitbox, ny keyhandler början.	14
19	Mer på keyhandler, flyttat runt funktioner för bättre abstraktion, mycket mindre förbättringar, meny.	15
20	meny, scoreboard fix, ufots precision mm.	15
21	småfix med filer, och små funktioner förbättras, kommentarer kollas över, projektdokumentation skrivs.	6

5.2 Oscar

Vecka	Arbetsuppgift	Tid (h)
14	Kravspecifikation	3
15	Kravspecifikation. Prototyp av spelarkontrollerat skepp på en skärm.	6
16	Fungerande skepp med stöd för att rotera och att gasa. Kommentering av kod inför halvtidsmöte. Arbete med generell kodstruktur, dvs. skapande av de olika modulerna och fortsatt funderande på implementationen.	16
17	Asteroider. Metod för att skjuta. Kommentering av kod inför halvtidsmöte forts.	8
18	Hashtabell för tangenter. Kollisionsdetektering. Implementation av ufon.	14
19	Omstrukturering av moduler och vissa procedurer för tydligare abstraktion. Hashtabell för tangenter forts. Spelinitialisering och nivåsystem.	12
20	Kommentering av kod. Spelinitialisering och nivåsystem forts. Presentation av spelarpoäng och liv.	8
21	Arbete med projektdokumentation. Ytterligare kommentering av kod.	6