Projektbeskrivning

Java Asteroider

2022-05-20

Projektmedlemmar:

Jacob Alzén <jacal732@student.liu.se>

Jonathan Björkdal <jonbj802@student.liu.se>

Handledare:

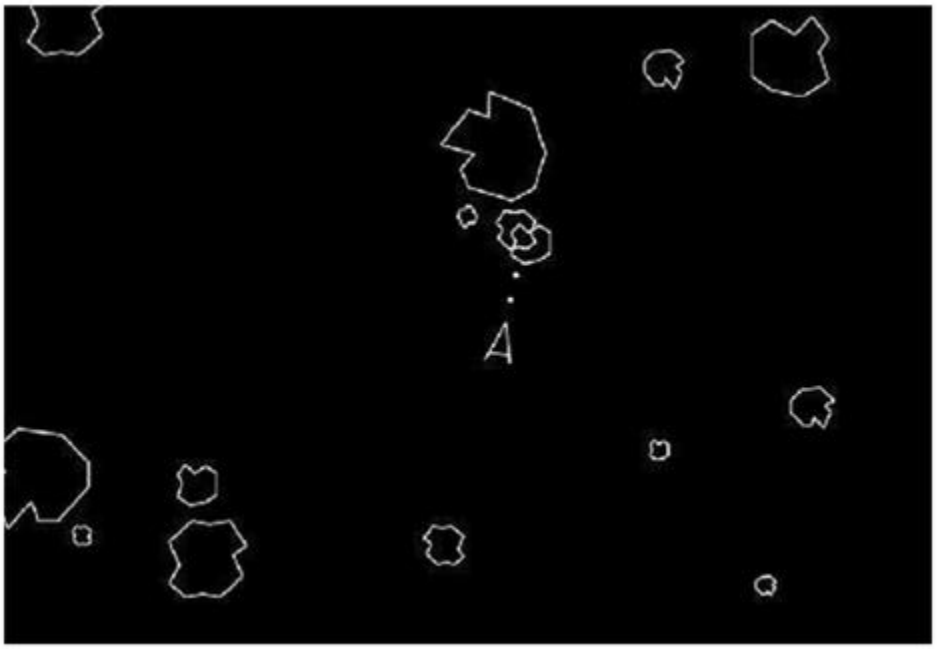
Daniel Pettersson <danpe975@student.liu.se>

Innehåll

Projektplan

1. Introduktion till projektet

Tanken är att implementera en modern klon av det klassiska Atari-spelet ”Asteroids” (se figur #1). Som namnet kanske antyder, går spelet ut på att man ska flyga med ett litet skepp i rymden och skjuta ner samt undvika asteroider. Samtidigt ska spelaren bekämpa diverse fiender som dyker upp utan förvarning. Dessa fiender kan till exempel vara ”flygande tefat” eller andra typer av främmande objekt.



Figur 1: Skärmbild från inspirationsprojekt.

Idén är att utveckla spelet med olika ”powerups” och diverse olika fiender. Om tiden tillåter är planen även att implementera ett lokalt flerspelarläge.

2. Ytterligare bakgrundsinformation

Asteroids släpptes i november 1979 för Ataris arkadkonsoler och blev en väldigt stor hit när det släpptes på hemmakonsolerna 1981. Spelet är implementerat i vektorgrafik och är bara uppbyggt av raka linjer, vilket bidrar till dess charm. Tanken är alltså att det ska gå att skjuta sönder asteroider och eventuella ”flygande tefat” som kan dyka upp. När en asteroid skjuts sönder så bryts den upp i mindre asteroider. Med tiden kommer fler och fler, vilket gör det hela

tiden blir svårare. Att krocka med en asteroid, eller att bli skjuten av ett ”tefat”, gör att spelaren förlorar ett liv.

Mer information om spelet hittas på dess Wikipedia-sida (https://en.wikipedia.org/wiki/Asteroids_(video_game)).

3. Milstolpar

Nedan följer en lista (tabell #1) med milstolpar som bygger upp ett mål med vad projektet ska, eller bör, innehålla när det är färdigt.

Tabell #1: Planering av milstolpar.

#	Beskrivning
1	Grundläggande klasser för, skepp, asteroider, spelare och rendering.
2	Uppritning av skepp och bakgrund.
3	Rörelse av skepp.
4	Ritning av asteroider.
5	Rörelse av asteroider.
6	Skjuta kollisionslösa skott från skeppet.
7	Skjuta sönder asteroider.
8	Antal asteroider ökar med tiden.
9	Ritning av tefat.
10	Rörelse av tefat.
11	Tefat skjuter skott.
12	Tefat skjuter skott mot spelaren.
13	Asteroider bryts ner i mindre delar vid beskjutning.
14	Flera liv och poäng.
15	Poänglista med resultat från tidigare rundor.
16	Olika typer av asteroider eller “powerups” som ändrar funktioner.
17	Stöd för två lokala spelar på en skärm.
18	Spelläge där spelarna tävlar mot varandra.
19	Ljud och bakgrundsmusik.

20	Spelläge med två separata spelplaner.
21	Spara speltillstånd och pausa spelet.
22	Stöd för handkontroller.

4. Övriga implementationsförberedelser

En klass ska hantera uppritning av spelplanen och dess komponenter. Alla objekt som kan röra sig på skärmen implementerar gränssnittet “MoveableObject” och under det ska gränssnittet “EnemyObject” finnas. Spelaren (d.v.s. raketen) tillhör “MoveableObject”. Både asteroider och ”tefat” är fiender som kan skada spelaren och tillhör därför “EnemyObject”. Eventuellt ska en ”renderare” för simpel vektorgrafik användas för att rendera linjer mellan fördefinierade punkter.

5. Utveckling och samarbete

Diskussionen kring utveckling och samarbete fördes muntligt och vi är båda införstådda i vad som gäller.

Projektrapport

6. Implementationsbeskrivning

Kapitel 6 förklarar implementationen av *Java Asteroids*. Detta sker med hjälp av UML-diagram, beskrivning av övergripande programstruktur samt milstolparna för utvecklingsprocessen.

6.1. Milstolpar

Tabell #2 beskriver de milstolpar som sattes upp för projektet. Den innehåller även ifall stegen är avklarade eller inte och i så fall varför.

Tabell #2: Tabell för de olika milstolparna med implementationsstatus.

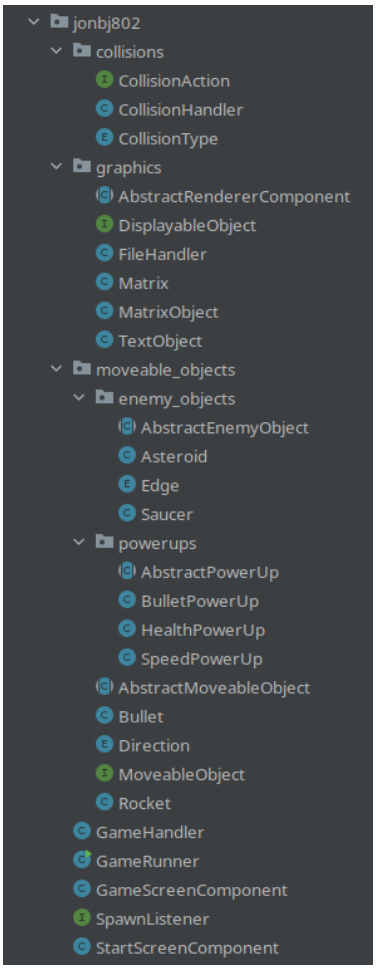
#	Beskrivning
1	**Grundläggande klasser för, skepp, asteroider, spelare och rendering:** Avklarad.
2	**Uppritning av skepp och bakgrund:** Avklarad.
3	**Rörelse av skepp:** Avklarad.
4	**Ritning av asteroider:** Avklarad.
5	**Rörelse av asteroider:** Avklarad.
6	**Skjuta kollisionslösa skott från skeppet:** Avklarad.
7	**Skjuta sönder asteroider:** Avklarad.
8	**Antal asteroider ökar med tiden:**

	Avklarad men togs bort då det inte ansågs lämpligt.
9	**Ritning av tefat:** Avklarad.
10	**Rörelse av tefat:** Avklarad.
11	**Tefat skjuter skott:** Avklarad.
12	**Tefat skjuter skott mot spelaren:** Avklarad.
13	**Asteroider bryts ner i mindre delar vid beskjutning:** Avklarad.
14	**Flera liv och poäng:** Avklarad.
15	**Poänglista med resultat från tidigare rundor:** Inte påbörjad.
16	**Olika typer av asteroider eller “power-ups” som ändrar funktioner:** Avklarad.
17	**Stöd för två lokala spelar på en skärm:** Inte påbörjad.
18	**Spelläge där spelarna tävlar mot varandra:** Inte påbörjad.
19	**Ljud och bakgrundsmusik:** Inte påbörjad.
20	**Spelläge med två separata spelplaner:** Inte påbörjad.
21	**Spara speltillstånd och pausa spelet:** Avklarad.
22	**Stöd för handkontroller:**

Inte påbörjad.

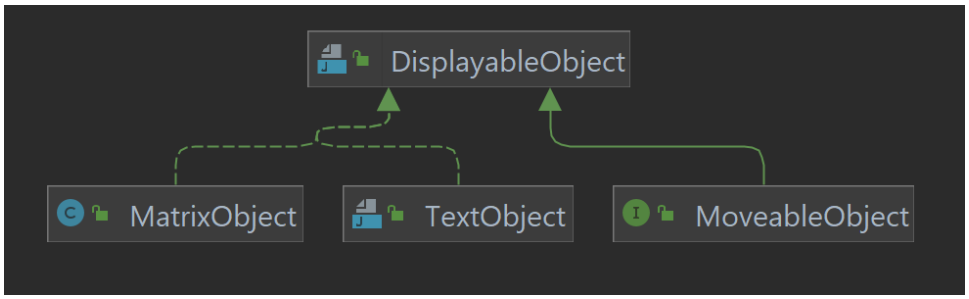
6.2. Dokumentation för programstruktur, med UML-diagram

Programkoden är uppdelad främst i tre paket. Det första är *moveable_objects* som innehåller alla klasser och underpaket som berör rörliga objekt på skärmen (alltså alla spelpjäser). Det andra paketet är g*raphics* och innehåller allt som berör grafik. Det tredje paketet är *collisions* och innehåller klasser för hantering av kollision mellan objekt. Paketet *moveable_objects* innehåller klasserna för de olika pjäserna, det vill säga, asteroiden, raketen, skotten och det flygande tefatet. Detta paket är även indelat i *enemy_objects* som innehåller fienderna och paketet *powerups* som innehåller alla olika "powerups". I grafikpaketet finns till exempel logiken för vektorgrafiken men också koden för att ladda in filerna som representerar de olika pjäserna. Dessa representationsfiler ligger i ett eget paket som heter *matrices* (paketet finns i mappen *resources/images* och inte bland de andra klasserna). Filerna är av typen *JSON* och innehåller listor med koordinater som det ska ritas linjer emellan. Dessa läses in av klassen *FileHandler*. I figur #2 går det att se den övergripande strukturen på paketen och deras tillhörande klasser. Det finns även fem filer som ligger i huvudpaketet. Funktionen av dessa klasser är till exempel att ta hand om hur spelet körs och då även hur fönstret för spelet hanteras. Det är från filen *GameRunner* som det går att starta själva spelet.



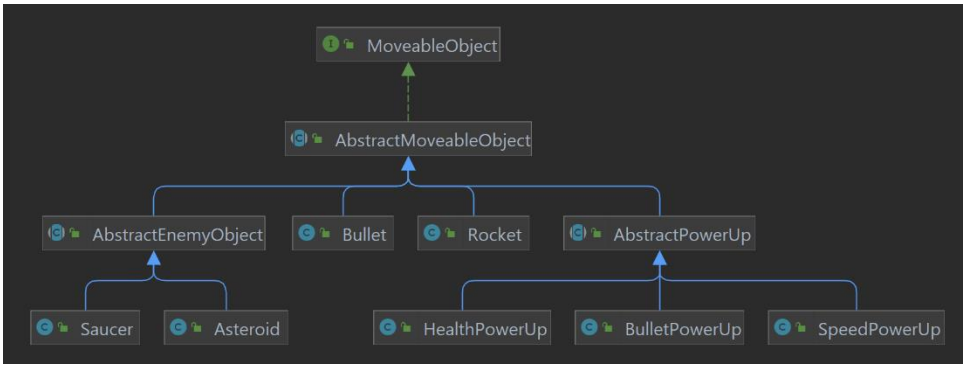
Figur 2: En skärmbild som visar uppdelningen mellan paket och klasser i projektet.

En av de viktigaste delarna i objektorientering är uppdelning av klasser och att låta dem ärva information och beteende från varandra. Denna principiella metod kallas för subtypspolymorfism. Fördelarna är att koden blir strukturerad på ett lättförståeligt sätt samtidigt som mängden duplicerad kod minskar. Med hjälp av substypspolymorfism kan exempelvis en superklass ha funktionalitet som sedan underklasser bygger vidare på. Ett exempel i projektets kodbas är gränssnitten *DisplayableObject* och *MoveableObject*. *DisplayableObject* definierar det absolut minsta som krävs för att renderaren ska kunna rita upp ett objekt, på en specifik position, på skärmen. Gränssnittet *MoveableObject* förlänger *DisplayableObject* med stöd för bland annat kollisioner och förflyttning. Uppdelningen av dessa gränssnitt förtydligas i figur #3.



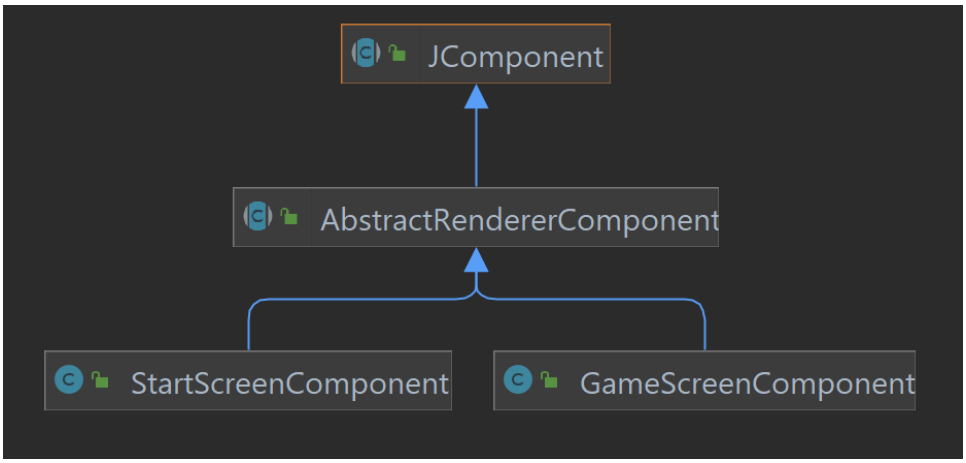
Figur 3: UML-Diagram för relationerna mellan DisplayableObject, MovableObject, MatrixObject och TextObject.

När det finns många klasser som implementerar samma gränssnitt kan det bli mycket duplicerad kod. Detta hanteras genom abstrakta klasser som innehåller gemensam kod som sedan underklasser baseras på. I detta projekt finns till exempel *AbstractMoveableObject* för vanliga rörliga objekt. Det finns också en underklass *AbstractEnemyObject* som förlänger *AbstractMoveableObject* med mer specifik funktion relaterad till fienden. Funktionen med de abstrakta klasserna är att de ska utnyttjas i objekt som kan dra nytta av den gemensamma koden. Dessa är raketen, skotten, asteroiden och tefatet. Med andra ord är alltså alla pjäser i spelet av typen *MoveableObject*. Tefatet och asteroiden använder sig av *AbstractEnemyObject* eftersom de är fiender till spelaren. Skotten och raketen använder *AbstractMoveableObject* direkt medan fienden bara använder det genom *AbstractEnemyObject*. Dessa relationer visualiseras i figur #4. I kombination med gränssnitt gör detta att: om det förutsätts att objektet som refereras till är av typen *MoveableObject så* går det att kalla på dessa metoder, utan att veta vilken specifik klass objektet tillhör. Ett exempel på detta är att alla olika objekt som ligger i en lista uppdateras genom att kalla på metoden *update,* som finns i gränssnittet.



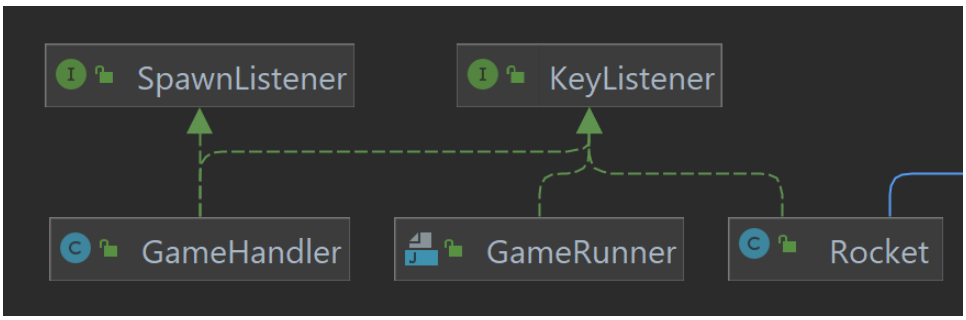
Figur 4: UML-Diagram för hur alla MoveableObjects relaterar till varandra.

Projektet utnyttjar även importerade paket från till exempel *Swing* och de används bland annat för fönsterhantering och uppritning. Ett exempel är *JComponent* som används för de komponenter som visas upp på skärmen. Hur denna abstrakta klass påbyggs, för att skapa fönstren och innehållet i dem, går att se i figur #5.



Figur 5: UML-Diagram som illustrerar hur klasserna från detta projekt relaterar till Swings JComponent.

En vanlig typ av klass inom programmering är lyssnare och de används på ett antal platser i projektet. Dessa lyssnare inväntar någon typ av indata från koden, eller användaren, och hanterar den. Här används tekniken för att ha en *SpawnListener* som väntar på att någon klass ska kalla på dess *spawn*-funktion, som lägger till objektet på skärmen. Den andra typen är en *KeyListener*. Klasser som implementerar detta gränssnitt lyssnar efter knapptryck på tangentbordet. Vad som sker vid detta knapptryck hanterar lyssnaren. Se figur #6 för illustration av vilka klasser som implementerar vilka lyssnare.



Figur 6: UML-Diagram för lyssnare.

All kod i spelet utgår från filen *GameRunner*. När denna fil körs börjar programmet skapa fönstret som spelet visas i. Därefter initialiseras startskärmen via klassen *StartComponent* och fönstret börjar lyssna efter knapptryckningar från tangentbordet. Efter det steget finns det en startskärm med text på och spelaren kan då starta spelet. När det startas läggs *GameComponent* till i fönstret. Det är då som alla objekt läggs till och timern som kör spelet skapas via metoden *setUpTimer*. Timern startas sedan från metoden *start*. Denna timer har hand om att uppdatera spellogiken 50 gånger per sekund. Den kallar först på *spawnObjects* som lägger till nya asteroider eller tefat ifall de behövs. Sedan körs metoden *updateObjects,* som tar hand om att uppdatera objekten. Uppdateringens funktion är huvudsakligen att se till att objektens position är korrekt samt att ta bort de objekt som har hamnat utanför skärmen. Borttagning av objekt utanför skärmen görs för att underlätta för processorn, då den får färre objekt att hålla reda på.

Nästa steg är att kalla på metoden *findCollisions* som går igenom alla objekt och testar om de kolliderar. Kollisionshanteringen använder sig av klassen *Rect* från *Swing* och dess metod *Intersects*. Varje *MoveableObject* har en kollisionsruta på samma plats på skärmen och om två kollisionsrutor överlappar så räknas det som en kollision. Om en kollision upptäcks kallar den på metoden *collided* på de två objekten som kolliderade. Den kör även den funktion som är definierad i *CollisionHandler* för kollision mellan just de här två typerna av objekt. Slutligen ser timern till att uppdatera skärmen så att objektens förändringar syns i fönstret.

Objektorienteringen används även på andra sätt i detta projekt; principer så som konstruktorer, överladdning, överskrivning och inkapsling. Konstruktorer är ett sätt att skapa en ny instans av en klass och samtidigt initialisera den. Detta möjliggör skapelsen av flera versioner av samma objekt där alla objekt automatiskt blir förberedda att direkt användas. Objekten kan skilja sig åt eller vara exakt likadana. Ett sätt att variera dem är att ändra parametrarna som konstruktorn tar in, för att få olika resultat. Inom objektorientering går det att skapa flera konstruktorer för en och samma klass. De varierade kontruktorerna kan då ändra vilka parametrar de ska ta in och hur klassen skapas. Detta används i projektet genom att ha två olika konstruktorer för till exempel asteroiderna. Asteroiderna har en konstruktor för skapelsen av den första asteroiden och en för de nedbrutna och mindre asteroiderna, som är avkommor från den första. Att bygga konstruktorer (och metoder) på detta sätt är en typ av överladdning; vilket även utnyttjas i andra delar i projektet.

Överskrivning tillåter att skriva över metoder från superklasser och används i detta projekt för att definiera metoder som specificeras från gränssnitt som exempelvis *MoveableObject*. Gränssnittet *MoveableObject* definierar att dess underklasser ska ha metoderna *update, getPos, setPos* med flera, vilka sedan alla objekt av den typen måste överskriva och implementera.

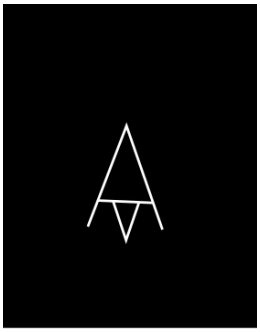
Inkapsling är även en metod som brukas i projektet och som genom olika tillgångsnivåer tillåter klasser att interagera på ett specificerat sätt. Som exempel på detta kan alla klasser nå *MoveableObjects update*-metod eftersom den är angiven att vara publik. Då kan metoden *update* hantera det privata innehållet i klassen. Om klasserna inte hade varit uppbyggda på detta sätt hade metoden *update* endast varit tillgänglig från objektets klass, vilket hade gjort det svårt för GameHandler att komma åt den. Annars hade koden implicit behövt kontrollera vilken klass ett objekt tillhör och då följs inte principerna från objektorientering på ett lämpligt sätt. Då kan inkapsling och gränssnitt användas så att de metoder som används internt i klasserna är privata. De metoder som andra klasser använder sig av är istället publika och tillhör ett lämpligt gränssnitt.

7. Användarmanual

Det absolut första som syns när spelet startas är startskärmen (se figur #7). Det är i denna meny som spelaren väljer att starta en omgång i spelet. Det är också här spelaren får börja om efter att ha förlorat en omgång. När användaren känner sig redo är det bara att klicka på knappen “enter” (även känd som “return”) för att sätta igång spelet.

Figur 7: Bild på startskärmen.

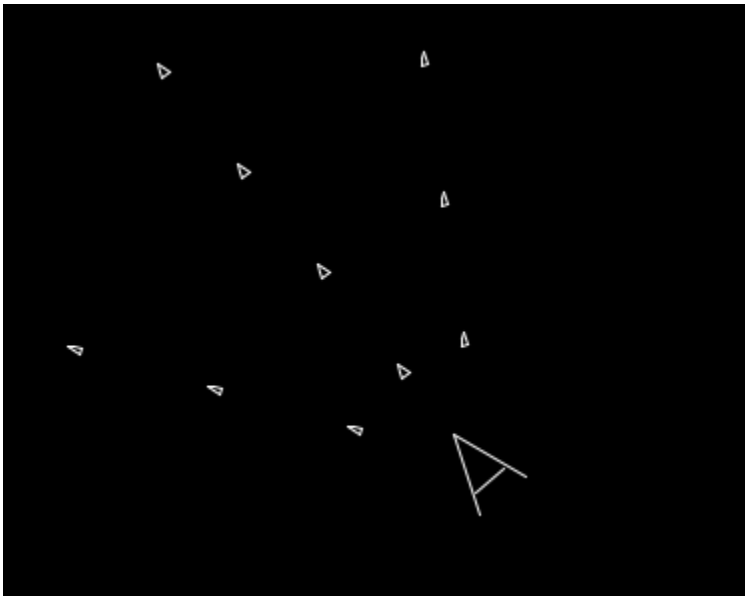
Inne i en spelomgång är raketen det första som syns. Det är denna raket som spelaren manövrerar för att beskjuta asteroiderna och även de flygande tefaten som kommer lite senare. Rörelserna styrs antingen med piltangenterna eller knapparna “w”, “a” och “d”. Uppåtpil eller “w” knappen används för att raketen ska röra sig framåt. När det sker visas det eld från raketmotorn för att indikera att raketen rör sig framåt. Detta kan ses i figur #8. Om raketen åker utanför någon av kanterna på spelplanen kommer den ut på motstående sida, i höjd med där den åkte ut. Tröghet finns implementerat i spelet och det betyder att raketen fortsätter att röra sig i samma riktning en kort stund, även om spelaren slutar att trycka ner knappen för att röra sig framåt.



Figur 8: Raket med eld från motorn.

För att rotera kan antingen höger- och vänsterpil eller “a”- och “d”-knapparna användas. Med dessa roterar raketen runt sin egen axel men den ändrar bara rörelseriktning om den samtidigt rör sig framåt med hjälp av motorn. Effekten av detta är att det snabbaste sättet att göra en U-sväng är att släppa framåtknappen, rotera 180 grader och sedan trycka på framåtknappen igen. Då har rörelsen i den tidigare riktningen motverkats och svängradien blivit minimal.

Raketen skjuter när spelaren trycker på knappen “mellanslag”. Skotten skjuts ut från nosen och fortsätter i samma vinkel som raketen hade när de avfyrades. När skotten åker utanför spelplanen kommer de, precis som raketen, ut från motstående sida men de försvinner också efter två sekunder. Om raketen har plockat upp en “powerup” som ändrar skotten skjuts det ut skott i tre riktningar, istället för bara rak framåt. Exempel på hur skotten skjuts från raketen kan ses i figur #9.

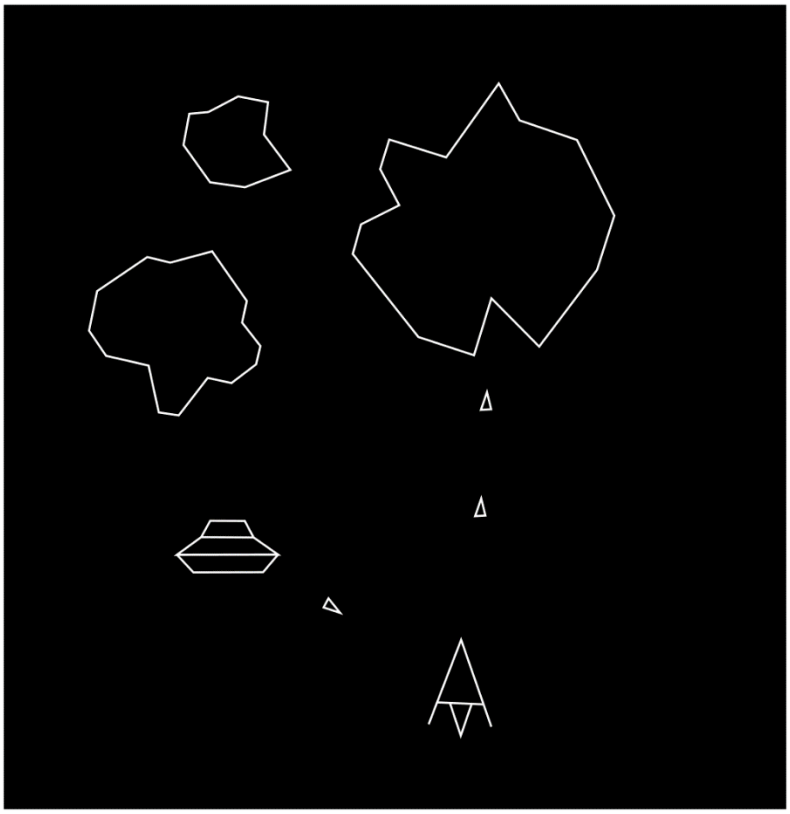


Figur 9: Skott med "powerup".

Om raketen blir träffad förlorar den ett liv. Eftersom spelaren börjar med tre liv betyder detta att det går att blir träffad max tre gånger innan spelomgången är slut. När raketen blir träffad startar den om i mitten av spelplanen och dess vinkel blir återställd så att den pekar uppåt. Samtidigt stängs också kollisionerna av i två sekunder för att undvika att raketen blir träffad direkt när den startar om. När alla liv är borta kan spelaren trycka på “enter” för att komma tillbaka till huvudmenyn.

Det går även att pausa spelet genom att trycka på knappen “escape” högst uppe i vänstra hörnet på tangentbordet. När spelet väl är pausat kan det startas igen genom att trycka en gång till på samma knapp.

Huvudmålet är, som tidigare nämnts, att skjuta sönder föremål och tjäna poäng. Självklart måste det finnas något att skjuta sönder och detta är fiender i form av asteroider och flygande tefat. Alla dessa fiender dyker upp vid kanten på skärmen och rör sig in mot spelplanen. Olika fiender ger olika mycket poäng när de skjuts sönder. Spelaren har tre liv från början och bör undvika att bli träffad av fienderna. Ett exempel på hur detta kan se ut kan ses i figur #10.



Figur 10: Exempelbild från en spelomgång.

Den första fienden som dyker upp är asteroiden. Asteroiderna förekommer i ett antal olika storlekar och former. Storleken på en asteroid påverkar också dess hastighet. Mindre asteroider rör sig betydligt snabbare än sina större motsvarigheter. När en asteroid blir träffad delar den upp sig i två mindre asteroider som rör sig i ungefär samma riktning som den asteroid som blev sönderskjuten. Tillräckligt små asteroider kommer att försvinna helt när de blir träffade.

Den andra fienden är det flygande tefatet. De dyker upp med jämna mellanrum och syns ganska sällan. Tefaten är betydligt farligare än asteroiderna och därmed ges det även fler poäng när spelaren skjuter ner dem. Tefatet skjuter skott, med jämna mellanrum, mot den plats där spelaren befinner sig. Visserligen går det då att undvika skotten med lite skicklighet, men det blir betydligt svårare om det samtidigt finns andra fiender runt omkring som måste undvikas. Skotten från tefatet och skotten från spelaren ser likadana ut. Se till att hålla koll på vilka skott som tillhör spelaren och vilka som tillhör tefatet.

Det finns tre olika “powerups” i spelet och ikonerna för dessa kan ses i figur # 11. De förekommer med längre intervall än flygande tefat och försvinner efter sju sekunder. När raketen åker igenom en av dem får den en effekt i tio sekunder. Vilken effekt det blir beror på vilken typ det är som den kolliderar med. Den första typen ökar raketens maximala hastighet. Den andra typen påverkar skotten och gör så att det skjuts ut år tre håll, med 30 graders vinkel emellan, istället för att bara skjuta rak framåt. Den tredje, och sista typen, ger ett extra liv till raketen.

Figur 11: Olika typer av "powerups".