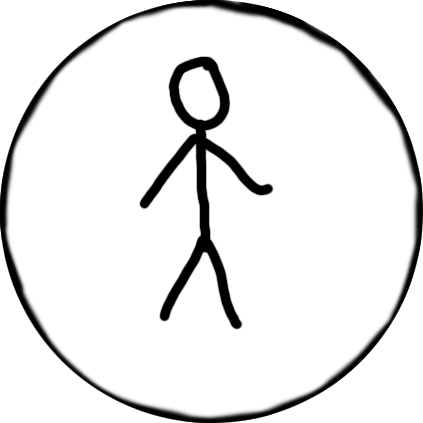
Creating a multiplayer game



# Inledning

# Syfte

Projektet går ut på att skapa ett datorspel, vilket ska vara två-dimensionellt samt över internet. Vi programmerar det i Java, ett objektorienterat programmeringsspråk och vi gör spelmotorn själva, vilket innebär att vi får göra vår egen grafik- och fysikmotor, samt nätverksprotokoll.

Spelet ska vara ett sorts skjutspel, där vi har minst 2 spelare som kan skjuta på varandra.

# Resultat

**Grund**Själva grunden till spelet bygger på klassen Stage. Koden i stage är den som körs först när spelet startas, och det är stage som startar spelets andra delar.

Det är stage som skapar fönstret man spelar i, har hand om man spelar i fullskärm eller inte, och stänger av spelet när man trycker på escape. Det är stage som skapar och startar både grafik- och fysikmotorn, samt nätverksdelen.

Stage håller även koll på om man kör som klient eller server, samt vem man är uppkopplad till om man kör som klient. Detta är information som andra delar kan fråga efter när de behöver den.

Stage fungerar även till viss del som ett nav, där vissa delar kan komma åt andra genom att fråga stage om dem.

## Grafikmotorn

Grafikmotorn bygger på javas grafiska standardbibliotek, vilket innebär att man har en panel där man målar polygoner (mm) med hjälp av en metod i panelen. Den använder en lista med *VisibleObjects* (objekt som kan målas) som den går igenom och målar objekten från. Objekten målas inte direkt i panelen, utan *VisibleObject* har en metod som räknar ut vart objektet ska målas beroende på var den befinner sig i sitt lokala koordinatsystem, samt ser till att de under den i hierarkin målas. Varje objekt som kan målas använder sig också av den metoden för att faktiskt målas, exempelvis polygoner som räknar ut var på skärmen dess punkter är, sedan målar sig där.

Det finns flera typer av objekt som kan målas, vanligast är polygoner och bilder, och det går att kombinera dem till nya objekt. Det finns också några objekt som målas utan att vara grafiska, såsom *Camera*, vilken kan beskrivas som just en kamera. Allting målas i relation till den, så att om den flyttas så flyttas även objekten på skärmen. Det finns också *VisibleStage*, vilken är gjord för objekt som inte ska målas i relation till kameran. Istället så kommer objekten i dess koordinatsystem alltid att målas på samma position (Tänk dig en hälsomätare i andra spel).  
Det finns också *CameraContainer*, vilken förenklar förflyttning av kameran genom att låta den följa en av dem, med möjlighet att växla vilken den följer.

**Inputsystem**

Spelet lyssnar på input på två olika ställen. Dels direkt i stage, och dels i en klass som heter just Input. Input bygger på ett mer avancerat system där den bara lyssnar efter specifika knapptryckningar och sen kör kod kopplad till knapptryckningen. Stage lyssnar däremot efter alla knapptryckningar, och kollar sen med en vanlig if-sats om den ska göra något vid just den här knapptryckningen.

När en knapp har tryckts ner så kan det hanteras på två olika sätt, beroende på vilken knapp som tryckts ner. Den kan antingen verka direkt, som escape som stänger av spelet, eller f11 som växlar mellan fullskärm och "windowed mode". Den kan också verka indirekt genom så kallade "flaggor", variabler som senare läses av annan del av koden för att veta vad spelaren vill göra. Flaggorna används bland annat för att veta vilket håll spelaren vill röra sig åt, eller om den vill skjuta.

**Fysikmotorn**

Fysikmotorn är den tyngre delen av spelet. Det är fysikmotorn som simulerar allt som händer i spelet, position, hastighet, acceleration, kollisioner och så vidare. Det är helt enkelt en fysiksimulator.

Motorn har en lista på alla objekt som finns på spelplanen. Med hjälp av en timer simulerar den sen steg för steg vad som händer. Det går inte att simulera kontinuerlig rörelse som i verkligheten, men genom att ta tillräckligt korta steg kan man komma nära nog.

Genom att beräkna hastighet, krafter och acceleration, kan man sen med hjälp av ett objekts position under början av steget räkna ut dess position och hastighet under nästa steg. Detta görs för alla objekt under varje steg. För att kunna beräkna mer exakt så mäter vi den faktiska tiden som passerar mellan två steg och använder den i våra beräkningar.

**Kollision**

En stor del av fysiksimulationen består av collission detection och collision response. Collision detection innebär upptäckandet av objekt som kolliderar med varandra. Collision response innebär vad man gör med objekten som kolliderar.

Fysikmotorn använder sig av en metod där den räknar ut om saker kommer kollidera innan kollisionen faktiskt har inträffat. Först delar den in listan med objekt i mindre grupper utefter var de befinner sig på spelplanen, för att slippa kolla vartenda objekt mot alla andra, utan bara behöver kolla om objekten som rör sig i samma område kolliderar. Därefter kollar den om två objekts banor korsar varandra. Om de gör det jämför den objektens positioner, hastigheter och accelerationer för att se om det är möjligt för dem att kollidera. Om det är det räknar den ut en ungefärlig tidpunkt strax innan kollisionen.

När samtliga kollisioner har hittats letar collision response upp kollisionen som inträffar först, och räknar ut vilka hastigheter objekten som kolliderade kommer att ha efter kollisionen. Sen uppdaterar den hastigheterna, tar bort alla objektens senare kollisioner, och gör om collision detection för de enskilda objekten med dess nya hastigheter. Detta upprepas tills inga kollisioner återstår.

**Nätverk**

Grunden till nätverksdelen är klassen UDPSocket, som kan skicka och ta emot data över ett nätverk. Denna klass använder protokollet UDP för att skicka data. UDP är ett snabbt protokoll som skickar data som enskilda paket (s.k. datagram), men inte garanterar att det som skickas kommer fram eller hamnar i rätt ordning. UDPSocket har bara hand om grundläggande uppgifter som att skicka och ta emot binär data till och från olika addresser.

Klassen SocketListener använder sig av metoden för att ta emot data i UDPSocket för att göra något användbart med informationen den får. SocketListener ser till att informationen tolkas så att den sen kan användas på rätt sätt.

För att konvertera mellan binär data och användbar data används en klass kallad Communication. Communication skriver om meddelandetyp, och värdet på alla vaiabler som ska skickas med som ettor och nollor. När ett meddelande sen tolkas så talar meddelandetypen om hur meddelandet ska läsas och vad som ska göras med det.

Det finns ett antal olika meddelandetyper. Det finns kontrollmeddelanden som till exempel kan tala om att en klient vill ansluta sig till servern. Det finns meddelanden för att skapa objekt, som innehåller all information om objektets storlek, massa, utseende, hastighet, position och så vidare som krävs för att kunna skapa det. Det finns meddelanden för att kunna uppdatera hastighet och position, meddelanden för att tala om att ett objekt ska tas bort och meddelanden för att be om information om objekt.

De flesta meddelanden är tätt knutna till fysikmotorn. I varje steg fysikmotorn tar ingår att applicera alla uppdateringar som kommit in sen senaste steget, och att i slutet av steget själv skicka uppdateringer. Silka uppdateringar som skickas beror på om spelet körs som klient eller server. En server skickar uppdateringar om alla objekt på spelplanen som genomgått någon större förändring, plus några fler för att synkronisera med klienterna. En klient skickar bara information om sin egen karaktär.

När en klient vill ansluta sig till servern skickar den ett kontrollmeddelande och talar om att den vill ansluta sig. Servern lägger då till klienten i en lista på spelare som den ska skicka uppdaterningar till, skapar ett objekt för att representera spelaren, och skickar ett meddelande med information om objektet till klienten, som skapar ett likadant objekt under klientens kontroll. Servern skickar därefter information om alla objekt på spelplanen, följt av en till uppdatering av spelarens position. Därefter börjar information skickas som vanligt.

**Process**

Programmering är långt ifrån en linjär process, och i stort sett alla delar av ett program kan förändras allteftersom man kommer på bättre lösningar. En hel del förändringar är också bara tillfälliga, gjorda i testsyfte för att senare att tas bort helt. Det är därför inte särskilt lätt att planera eller förutse arbetsgången, utan man får i stället försöka bestämma ungefär vilka delar man ska försöka göra i ungefär vilken ordning, och sen revidera allt eftersom.

Något av det första vi gjorde var att fixa ett verktyg för hur vi skulle kunna dela och uppdatera koden på ett enkelt och effektivt sätt. Vi använde oss av SVN, vilket i princip innebär att olika versioner av programmet sparas på en server, och att man laddar ner kopior på sin dator som man arbetar med, för att sedan uppdatera servern med förändringarna så att andra kan ta del av dem. Verktyget ser till att man inte kan uppdatera om man inte har den senaste versionen av filerna, vilket ser till att det inte blir konflikter i koden om man arbetar på samma fil.

Tack vare SVN så har vi även loggar över våra förändringar, och kan därför beskriva förloppet någorlunda väl.

Det började med att vi skapade den grundläggande klassen Stage, för att kunna starta de andra delarna. Felix började sen arbeta på grafiken, för att kunna rita upp saker på skärmen. Han skapade klassen Painter, som är en panel där sakerna målas, och klassen VisibleObject, som representerar saker som ska målas ut. Niclas började jobba på ett inputsystem, för att kunna känna av när vissa knappar trycktes ner. Hur man ritar ut saker på skärmen och lyssnade efter knapptryckningar fick vi leta upp på internet. Felix gjorde även ett test för Hooks, alltså händelser som en metod kan lyssna efter.

Niclas skapade sen första versionen av PhysicsEngine, början på fysikmotorn. Felix arbetade vidare på VisibleObjects genom att bland annat skapa en hierarki för hur de målades ut och ge dem lokala koordinatsystem som påverkar objekten längre ner i hierarkin. Han och skapade även PhysicsObjects (då kallade BasePhysics), vilka representerade objekt i spelet, och kopplade ihop dem med VisibleObjects så att ett PhysicsObject representerades av ett VisibleObject på skärmen.

Niclas skapade ett flaggsystem för att kunna markera när inputsystemet kände av att knappar var nedtryckta. Felix kopplade ihop PhysicsObjects med PhysicsEngine så att varje PhysicsObject kunde uppdatera sin position och hastighet för varje gång PhysicsEngine körde igenom ett nytt steg.

Vi skapade sen en klass för koordinater, för att förenkla representation av och beräkningar med x- och y-koordinater och vektorer. Därefter arbetade vi båda vidare på att förbättra fysikmotorn och VisibleObjects.

Niclas började sen arbeta med collision detection, ett moment som han inte riktigt visste hur han skulle lösa, och därför krävde en hel del efterforskning, tänkande och beräkningar. Mycket av det arbete han faktiskt gjorde blev överflödigt när han bestämde sig för ett annat tillvägagångssätt.

Under tiden skapade Felix en typ av VisibleObject som kunde ladda in bilder från filer och rita upp dem på skärmen. Han gjorde det även möjligt att byta skala och rotera dem.

Niclas bytte tillvägagångssätt för collision detection, och började med att bara använda sig av cirklar för att förenkla det hela. Han la till ett system för att se om två objekt skulle kollidera, och att uppskatta tiden för kollisionen. Han skapade även en klass för att spara information om kollisionen.

Niclas la sen till kod för collision response, där han använde sig av formler han letat reda på på internet för att räkna ut och uppdatera objekts hastigheter efter kollisioner med hjälp av deras hastigheter, positioner och massa i förhållande till varandra.

Därefter började Niclas med ett system för att kunna uppdatera och skapa objekt. Han gav objekt id-nummer, och skapade klasser innehållande information om objekt, samt skapade ett system med mallar för olika typer av objekt. Sen skapade han en klass för att konvertera den här datan till binär data som skulle kunna skickas över internet och sen tillbaka till användbar data igen. Han skapade UDPSocket och SocketListener för att ta hand om nätverkskommunikation, och integrerade dem med PhysicsEngine.

Felix gjorde mindre ändringar i grafikmotorn, gjorde så att kameran fungerade så som den skulle och skapade CameraContainer för att ge möjlighet till enkla cinematiska scener och liknande.

Därefter kom slutrycket. Vi arbetade med att få det som återstod av nätverkskoden att fungera. Vi la till kod för att hantera spelare, uppkopplingar, klienter och servrar. Vi skapade system för och hur data skulle skickas, och hur den skulle hanteras beroende på om man var klient eller server. Vi la till kod för hur skapandet av nya spelare skulle gå till. Vi la till ett nytt inputsystem, något enklare och annorlunda än det gamla systemet, la till möjligheten att skjuta, samt la in nya bilder att använda i spelet.

Till slut nådde vi en punkt där alla de grundläggande delarna vi ville ha med i spelet faktiskt fungerade någorlunda, och valde att stanna där för tillfället. Spelet är fortfarande instabilt och inte på långa vägar färdigt, men all funktionalitet vi ville uppnå finns där, och är redo för demonstration. Mycket både kan och skulle fortfarande behöva göras, och har vi tid och möjlighet kommer vi troligtvis att göra det, men vi har i alla fall något som tål att visas upp.

Vi hade förstås både planerat och hoppats komma längre än så här, men det har varit ett väldigt stort och omfattande projekt, och nu så här i efterhand så har vi faktiskt fått gjort en hel del. Att ha haft mer tid hade dock definitivt inte skadat.