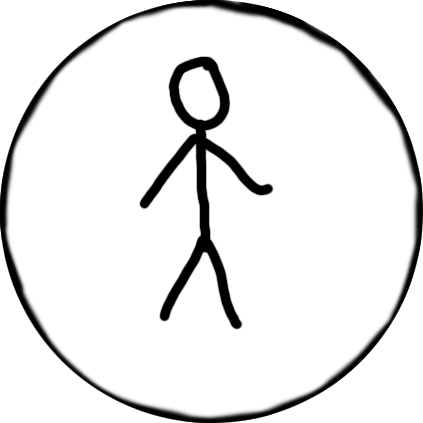
Creating a multiplayer game



# 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

Protokollet som används kallas för UDP, det är ett ”*connectionless*” protokoll, vilket innebär att till skillnad från TCP så etableras inte en ordentlig anslutning, utan istället så skickas data bara en gång och ingen konfirmation ifall data kommer fram skickas tillbaka, vilket innebär att den kan förloras på vägen. En annan skillnad är att UDP inte skickar lika mycket data fram och tillbaka.

När en klient vill ansluta sig till spelet så börjar den med att skicka ett paket till servern som för servern att lägga dess ip-adress i listan med klienter samt att skapa ett spelarobjekt för den. Spelarobjektet skickar i sin tur ett paket till klienten med ett id som talar om för klienten att den ska skapa ett lokalt spelarobjekt (härmed kallad spelare), och vilket id dess spelare på servern har. Klienten kommer då konstant att skicka data till servern för att synka spelaren på servern med sin lokala spelare, vilket inkluderar data om vilka knappar som klienten använder, så att när klienten försöker skjuta så skapas kulor på servern.

Men det räcker inte med att klienten synkar sin spelare med servern, utan klienten behöver också få data om de andra spelarna samt olika objekt på servern. Detta får den genom att när klienten ansluter till servern så skickar servern data om alla objekt så att klienten skapar fysiska representationer av dessa. Sedan så går servern vid behov genom listan med klienter och skickar data om objekten så att klienten kan uppdatera sina representationer av dem.

## Övrigt