

Projektbeskrivning

Become the Level!

2032-03-22

Projektmedlemmar:

Jonatan Larsson <jonla400@student.liu.se>

Handledare:

Philip Rettig <phire844@student.liu.se>

Innehåll

1. Introduktion till projektet.....	2
2. Ytterligare bakgrundsinformation.....	2
3. Milstolpar.....	2
4. Övriga implementationsförberedelser.....	4
5. Utveckling och samarbete.....	4
6. Implementationsbeskrivning.....	6
6.1. Milstolpar.....	6
6.2. Dokumentation för programstruktur, med UML-diagram.....	6
7. Användarmanual.....	7

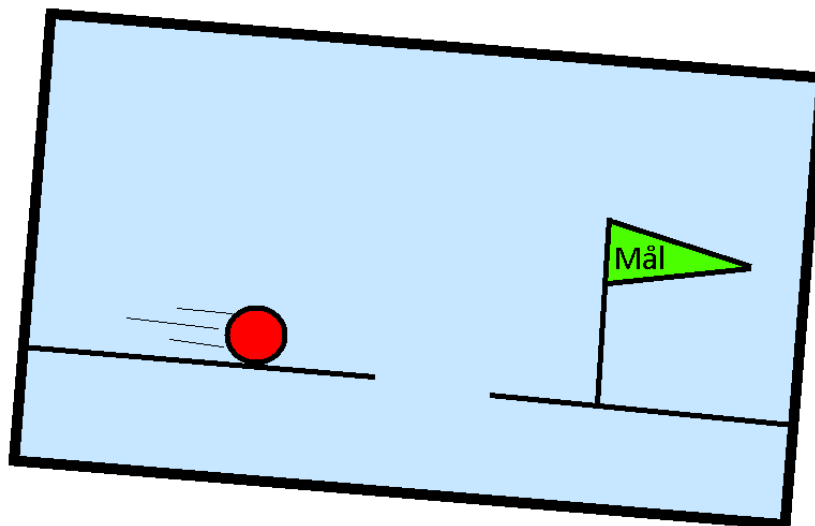
Projektplan

1. Introduktion till projektet

Programmet jag tänker utveckla är ett spel som liknar en 2D platformer men med en twist: Istället för att styra spelaren styr man hela nivån för att få spelaren att ta sig till ett utsatt mål. Man ska kunna styra nivån på två sätt: antingen genom att luta nivån med en godtycklig vinkel eller genom att förflytta nivån i en godtycklig 2D riktning. Spelaren kommer vara cirkulär och agera fysikaliskt trovärdigt utifrån hur en "boll" skulle röra sig i 2D. Om t.ex. spelaren till att börja med står stilla på en horisontell plattform, ska man kunna få spelaren att rulla åt höger genom att luta banan lite åt höger. Man ska också kunna förflytta banan snabbt uppåt för att få plattformen som spelaren står på att skjuta upp spelaren i luften.

Låter detta krångligt? För att få en intuition över detta kan man tänka sig hur spelet skulle se ut om det inte var ett dataspel utan ett "riktigt spel": Tänk dig att du håller i en nivå i dina händer som om du höll i en bilratt. Banan innehåller flera plattformar (golv, tak och väggar) samt en kula på en position långt åt vänster, och ditt mål är att ta kulan till en position som ligger långt åt höger. Du kanske börjar med att luta banan åt höger för att få kulan att börja rulla mot målet. Sedan kanske du stöter på en glipa i plattformen som spelaren rullar på, vilket tvingar dig att förflytta banan hastigt uppåt för att få spelaren att hoppa över glipan. Se figur 1.

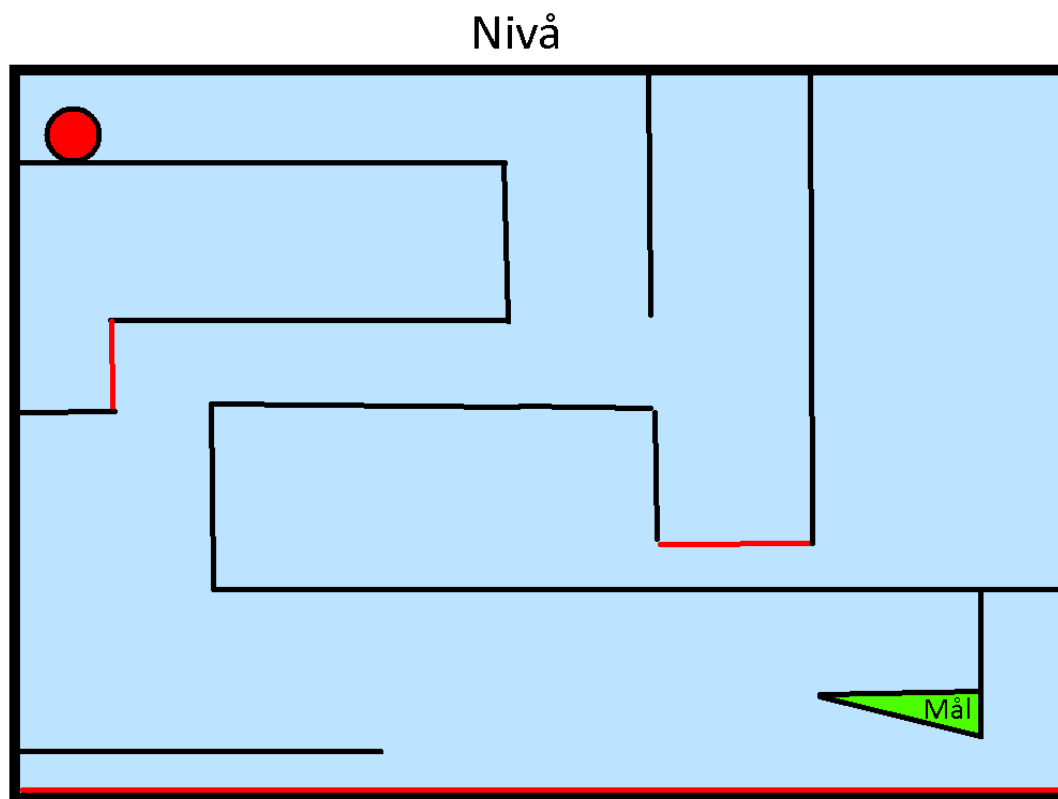
Nivå



Figur 1: En illustration över hur en nivå i spelet skulle kunna se ut. Nivån lutas åt höger för att få spelaren (den röda bollen) att rulla åt höger. För att ta sig över diket skulle man kunna förflytta nivån uppåt med hög hastighet för att få bollen att skjutas uppåt så att den på så sätt hoppar över diket.

I det riktiga dataspellet kommer man givetvis inte kunna hålla i en nivå i sina händer. Istället

kommer man använda musen för att luta och förflytta nivån. Detaljer om detta framgår i nästa avsnitt. Olika sorters plattformar beskrivs också i nästa avsnitt.



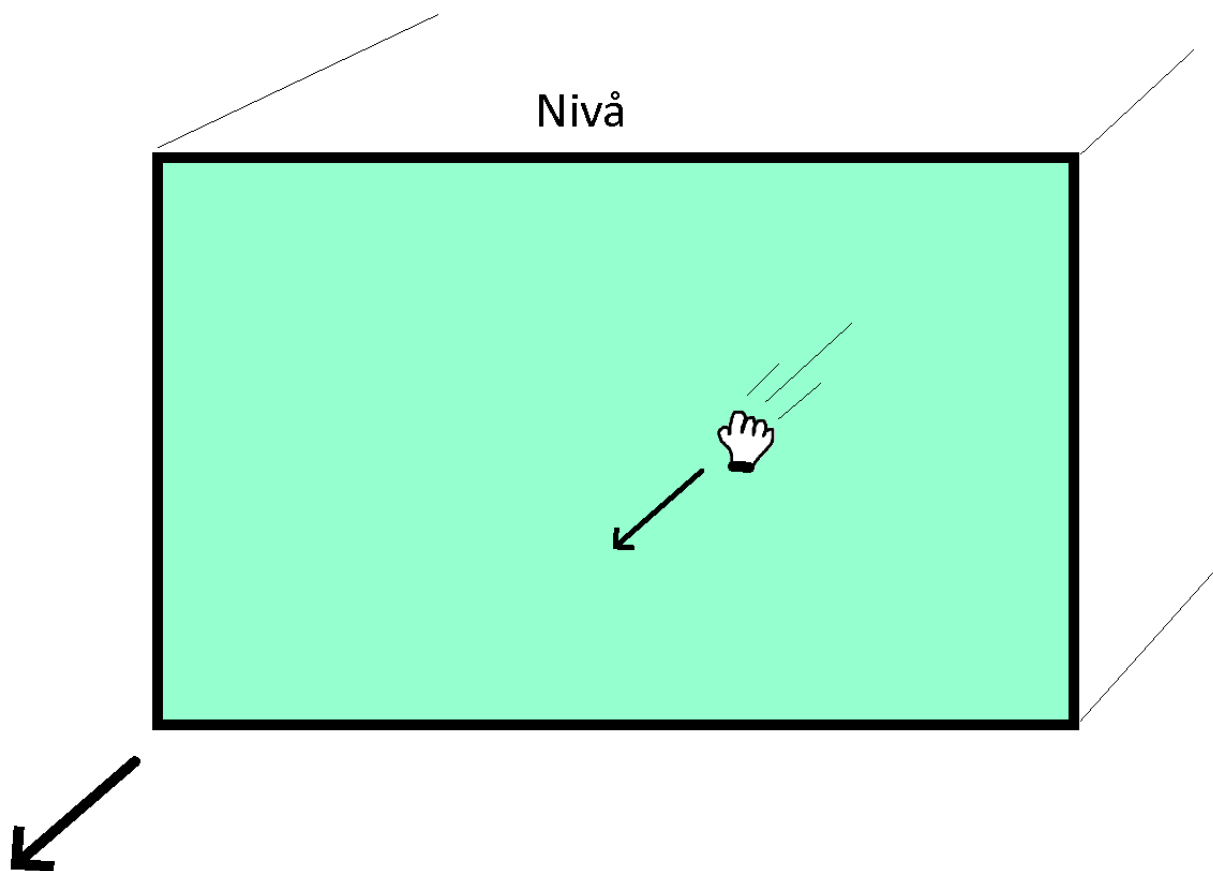
Figur 2: Illustrerar en lite mer komplex nivå. De röda linjerna innebär att man får Game Over och får starta om ifall man kolliderar med dem. Hur skulle du göra för att ta dig till målet? (Tips: sista delen kan kräva att du lutar nivån uppochner)

2. Ytterligare bakgrundsinformation

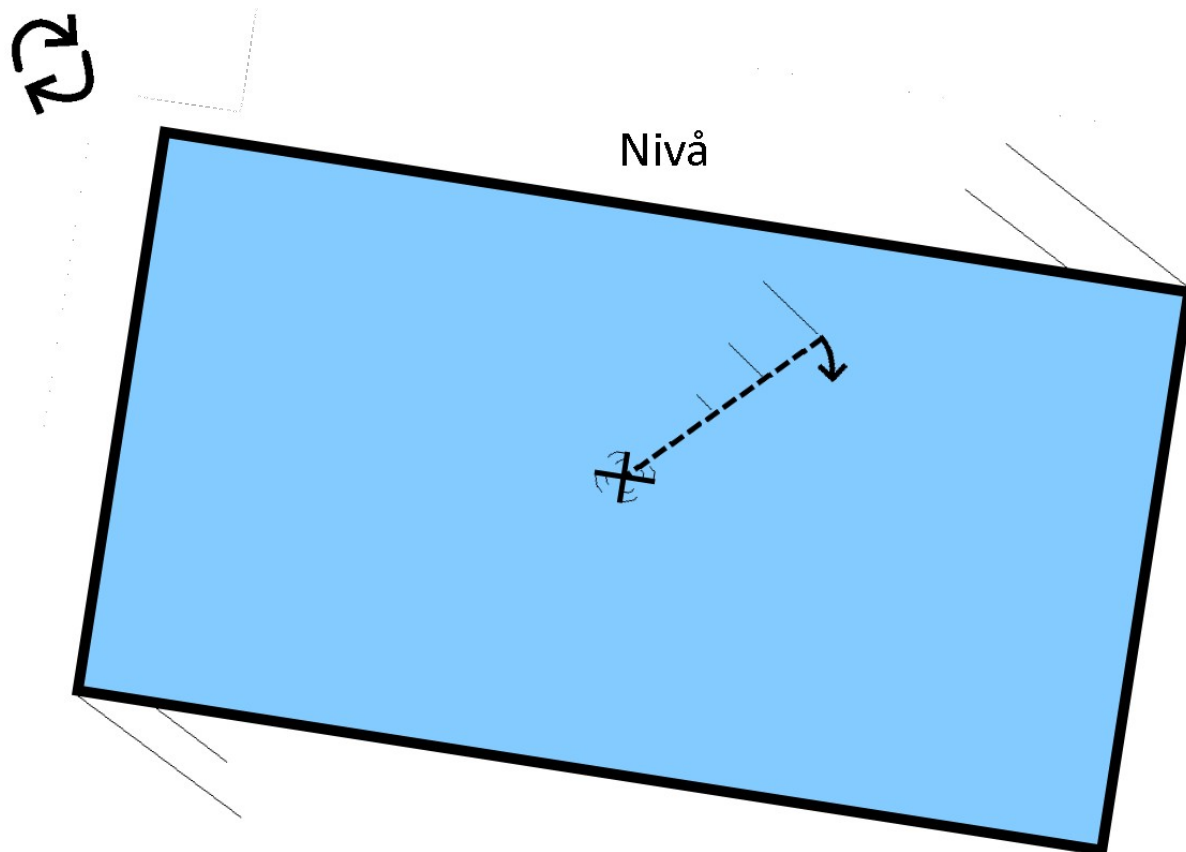
Som nämnt i det förra avsnittet ska man kunna luta och förflytta en spelnivå för att få spelaren att ta sig till nivåns utsatta mål. För både lutning och förflyttning av en nivå ska musen användas. Förflyttning av en nivå ska ske genom att man håller ner höger musknapp och sveper musen i den riktning man vill flytta nivån i, se figur 2. Rotation är lite knepigare eftersom man både måste kontrollera runt vilken punkt som rotationen ska ske (Runt mitten? Runt ett hörn?...) samt hur mycket nivån ska rotera. För att åstadkomma denna kontroll gäller följande förfarande för spelaren (se figur 4):

1. Tryck och håll ned vänster musknapp. Den position som musen har när du inleder trycket kommer bli punkten runt vilken nivån roterar.
2. Förflytta musen bort från rotationspunkten.
3. Så länge du förflyttar musen bort från eller mot rotationspunkten kommer nivån vara stilla. Om du däremot drar musen i en cirkulär rörelse runt rotationspunkten, kommer nivån roteras.

4. Släpp vänster musknapp för att avsluta rotationen.



Figur 3: Illustrerar förflyttning diagonalt ner åt vänster av en nivå. Förflyttningen görs genom att spelaren håller ner höger musknapp samtidigt som musen (handen på bilden) sveps diagonalt ner åt vänster.



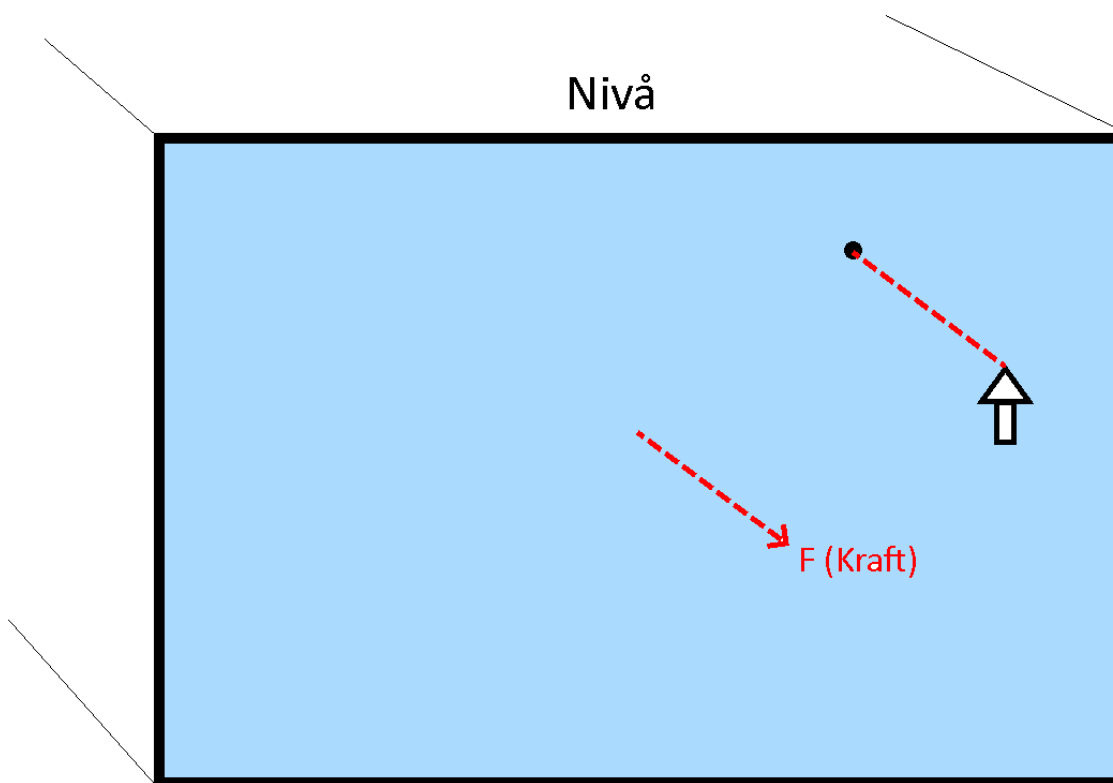
Figur 4: Illustrerar rotation åt höger runt mitten av en nivå. Spelaren börjar med att hålla ner vänster musknapp i mitten av nivån, vilket sätter ut rotationspunkten där (krysset i bilden). Sedan förflyttar spelaren musen till den andra änden av den streckade linjen. Till sist börjar spelaren förflytta musen medurs runt rotationspunkten (krysset), vilket gör att nivån i sig börjar rotera medurs runt rotationspunkten.

Det finns några viktiga aspekter om förflyttning och lutning som hittills inte tagits upp. För det första kommer varken förflyttning eller rotation ha en direkt påverkan på spelarens (bollens) position. Om bollen t.ex. befinner sig långt ner på skärmen i ett ögonblick, följt av att nivån roteras 180° så att nivån blir uppochner, kommer bollen inte hänga med i rotationen utan istället stanna vid samma låga position på skärmen. Detta exempel förutsätter dock att ingen plattform kolliderar med bollen under rotationen, för i sådant fall skulle bollen knuffas undan av plattformen.

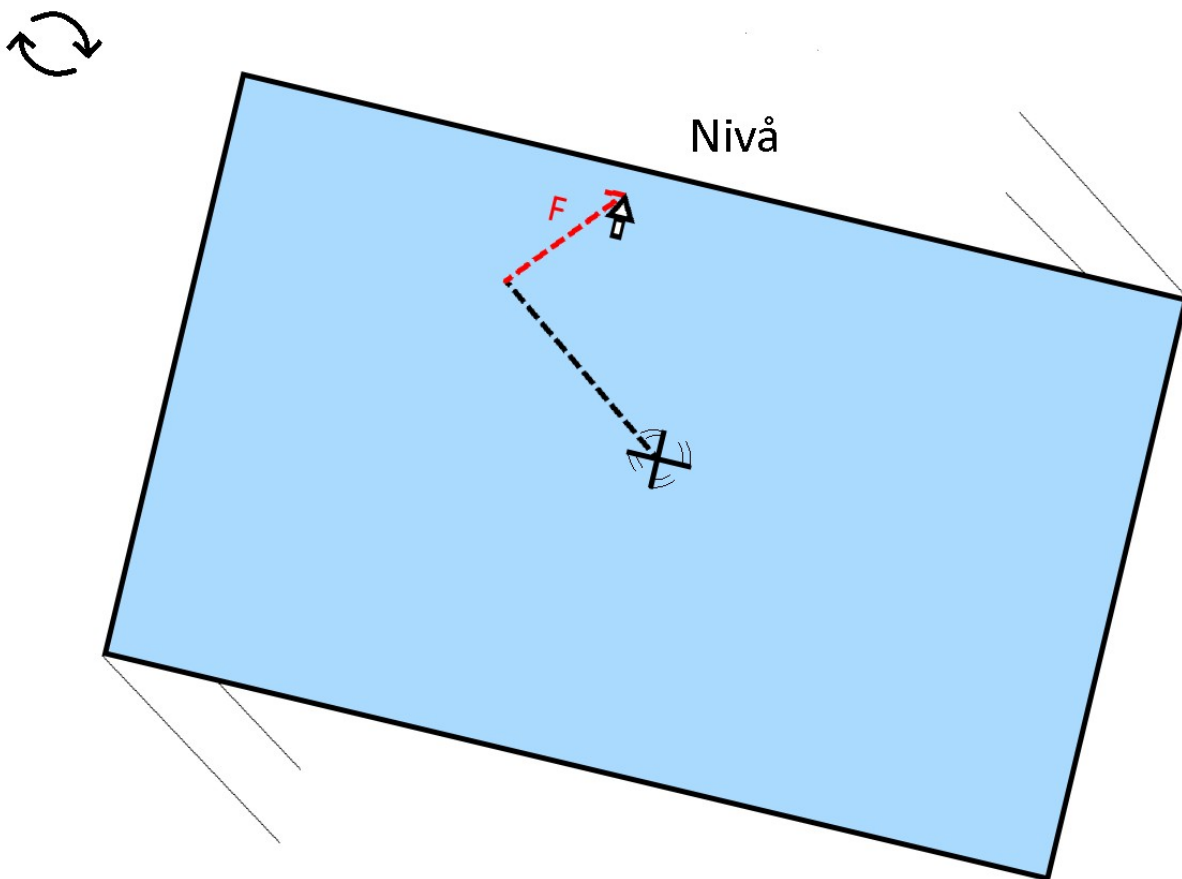
För det andra kommer "hävarmen" (den streckade linjen i figur 4) under rotation kunna ha en godtycklig startvinkel. Om t.ex. en spelare drar ut hävarmen åt höger kommer "höger" vara startvinkeln (dvs. den vinkel där ingen rotation har skett). Om spelaren istället drar ut hävarmen neråt kommer "neråt" vara startvinkeln, osv. Hävarmens riktning kan inte avgöras i samma ögonblick som spelaren sätter ut rotationspunkten (genom att klicka på vänster musknapp), eftersom musen då befinner sig på samma position som rotationspunkten. Detta ska lösas genom att musen först måste förflyttas en liten sträcka från rotationspunkten innan startvinkeln avgörs.

För det tredje bör varken förflyttning eller rotation genomföra det musen instruerar på en gång, för det skulle resultera i att bollen skjuts iväg okontrollerat av plattformar med för höga hastigheter under en förflyttning/rotation. Det är ju mycket enkelt för spelaren att förflytta musen långt och snabbt, men spelet blir mer stabilt om förflyttning och rotation har en viss "tyngd". Därför ska

nivån i sig simuleras fysikaliskt trovärdigt (inte bara bollen). Mer konkret innebär detta att nivån ska ha en egen massa (som kommer vara ett mått på hur mycket motstånd nivån gör mot förflyttning) samt ett [tröghetsmoment](#) (som kommer vara ett mått på hur mycket motstånd nivån gör mot rotation). Vid förflyttning ska en kraft appliceras på masscentrumet av nivån, vilket bygger upp linjär rörelse men ingen rotation eftersom kraften appliceras på masscentrumet (se bild 5). Vid rotation ska en kraft från hävarens yttre punkt till musen appliceras på nivån. Detta kommer dock ge den oönskade effekten att rotationspunkten förflyttar sig. För att förhindra detta kommer jag implementera någonting som i [fysikmotorer](#) brukar benämnas som constraints (begränsningar) för att begränsa rörelsen hos fysikaliskt simulerade kroppar. I mitt fall kommer jag implementera en constraint som spikar fast en viss punkt hos en fysikalisk kropp till "bakgrunden" så att punkten inte kan röra på sig. Denna constraint ska sättas på nivåns rotationspunkt. Efter det kan jag som tidigare beskrivit applicera en kraft från hävarens ände till musen, och sedan låta spikbegränsningen på rotationspunkten applicera en motkraft så att rotationspunkten förblir stilla (se bild 6).



Figur 5: Illustrerar förflyttning av en nivå, denna gång kraftbaserat. Kraften går från punkten som musen klickade på i början av förflyttningen (den svarta punkten) till musens nuvarande position. Kraften appliceras dock på nivåns masscentrum för att kraften inte ska bidra till någon rotation. Hur mycket motstånd nivån gör mot förflyttningen ges av nivåns massa.



Figur 6: Illustrerar rotation av en nivå, denna gång kraftbaserat. Kraften appliceras från hävarens ände till musen. För att inte få rotationspunkten (krysset i bilden) att förflytta sig i samband med kraften, kommer den spikas fast i "bakgrunden". Hur mycket motstånd nivån gör mot rotationen ges av nivåns tröghetsmoment.

Det ska finnas minst två typer av plattformar förutom den vanliga typen. Den ena av dessa typer ska vid kollision med spelaren orsaka Game Over och tvinga spelaren att börja om nivån. Den andra typen av plattform ska fungera som nivåns mål, dvs. en plattform som spelaren ska försöka ta sig till och kollidera med för att klara av nivån. De olika typerna av plattformar kan för enkelhetens skull signaleras till spelaren med olika färger.

Nivåerna i spelet ska sparas med hjälp av JSON i .json filer. En sådan fil ska innehålla hörnen och kanterna i nivån, nivåns startposition, nivåns massa, spelarens startposition, spelarens massa, spelarens storlek samt storleken på rektangeln genom vilken nivån syns på skärmen. Man kommer alltså kunna skapa nivåer genom att skriva denna information i .json filer, men för att underlätta skapandet av nivåer kommer java klasser och objekt användas för att först definiera en nivå för att sedan lagra nivån i en .json fil.

3. Milstolpar

Beskrivning

- 1 Det finns en generell struktur för att stega framåt i tid med diskreta tidssteg.

Det är också möjligt att lägga in avbrottsrutiner som sker under ett tidssteg. En testklass finns som förflyttar en punkt fram och tillbaka mellan två gränser under ett antal tidssteg, och skriver ut punktens status både efter varje tidssteg och under kollision med gränserna.

- 2 Det finns en datatyp för 2D vektorer samt en testklass som testar olika vektoroperationer. Datatypen har minst följande vektoroperationer:
- Addera två vektorer
 - Subtrahera två vektorer
 - Multiplicera en vektor med en skalär
 - Räkna ut skalärprodukten mellan två vektorer
 - Räkna ut kryssprodukten mellan två vektorer (där vektorerna implicit har 0 som z-komponenter och där resultatet motsvarar z-komponenten hos 3D vektorn som hade bildats om det var kryssprodukt mellan två 3D vektorer)
 - Rotation med godtycklig vinkel

En 2D vektor kan skapas med både polära- och kartesiska koordinater, och dessa koordinater kan även hämtas från en vektor.

- 3 Det finns en datatyp för en fysikalisk kropp samt en datatyp för en fysikrymd som innehåller flera kroppar och kan utföra tidssteg (utan kollisioner). Det finns också en testklass som skapar en kropp med starthastigheter (vanlig hastighet och vinkelhastighet), och genomför ett antal tidssteg med utskrifter av kroppens status. En annan testklass ska också finnas som applicerar impulser på olika ställen av en fysikalisk kropp och skriver ut hastighetsändringarna.

- 4 Det finns en grafisk komponent som målar ut kropparna i en fysikrymd. Det finns dessutom en testklass som skapar en fysikrymd, ett fönster och en grafisk komponent i fönstret som visar fysikrymden. Testklassen utför sedan tidssteg på fysikrymden med hjälp av en `java.swing.Timer`. Animationen syns i den grafiska komponenten.

- 5 En generell struktur för hastighetsbegränsningar finns. En global lösning till flera hastighetsbegränsningar kan approximeras iterativt. En konkret hastighetsbegränsning som spikar fast en punkt av en fysikalisk kropp är implementerad. Testklassen från förra steget spikar fast en fysikalisk kropp och testar spiken genom att applicera impulser på kroppen. Testklassen kan dessutom applicera en kraft nedåt före varje tidssteg (för att simulera gravitation) utan att den fastspikade punkten rör sig. Spiken begränsar hastigheten före varje tidssteg men efter att krafter/impulser applicerats.

- 6 Två nya hastighetsbegränsningar som skapar "top-down" friktion är implementerade. Den ena av dessa begränsningar strävar efter att nollställa den vanliga hastigheten av en fysikalisk kropp, men är begränsad utav en maximal friktionskraft. På samma sätt strävar den andra av dessa begränsningar att nollställa vinkelhastigheten av en fysikalisk kropp. Samma testklass som tidigare skapar en ny kropp med en starthastighet och startvinkelhastighet, och applicerar friktionsbegränsningarna för att

successivt få kroppen att stanna.

- 7 Det finns en ny hastighetsbegränsning som begränsar hastigheten på en kropp som kolliderar med någonting orörligt. Kollisionsinformationen som denna begränsning tar del av är:
- kontaktpunkten på kroppen (punkten som kolliderar)
 - kollisionsnormalen (åt vilken riktning en impuls ska appliceras)
 - studscoefficient (värde mellan 0 till 1. Anger hur mycket av farten längs kollisionsnormalen som ska bevaras)
 - friktionscoefficient
 - Separation (hur långt bort var kontaktpunkten från det orörliga föremålet, kan vara negativ. Bör vara nära noll)

Det finns ett ramverk för kontinuerlig kollisionsdetektion som tar som input hur långt det är kvar på ett tidssteg och ger tillbaka en kollision som inträffar inom tidssteget och exakt när kollisionen sker. Det finns även en "dummy" konkretisering av detta ramverk som rapporterar en kollision när en kropp nått en viss y-koordinat. Dessutom finns ett generellt ramverk för kollisionslyssnande samt en tidsstegs-avbrytare som avbryter ett tidssteg vid kollision och hanterar kollisionen. Det finns en konkretiserad kollisionslyssnare som lägger in en hastighetsbegränsning för kollisionen bland andra hastighetsbegränsningar, löser samtliga begränsningar, för att till sist ta bort kollisionsbegränsningen.

Genom dessa implementationer testar den sistnämnda testklassen kollision genom att sätta upp en kollisionsplan långt ner på skärmen, skapa kroppar högt upp på skärmen, applicera krafter neråt före varje tidssteg för att simulera gravitation, och se till att ett avbrott från tidssteget sker vid kollision med planen.

- 8 Den konkreta kollisionsdetektionen är generaliserad till att detektera kollision mellan en kropp och en godtycklig plan. Testklassen testar detta.
- 9 En datatyp för en "root-finder" är implementerad som kan finna en approximation till en rot av en godtycklig kontinuerlig funktion givet ett intervall. Den ska sträva efter att hitta den första roten inom intervallet. En ny testklass testar att leta efter rötter för olika funktioner.
- 10 Den konkreta kollisionsdetektionen är generaliserad till att detektera kollision mellan en kropp och en godtycklig plan som sitter på en annan kropp. Kollisionsdetektionen använder sig utav "root-findern" för att finna när separationen mellan kroppen och planen kommit nära noll. Testklassen som visualiserar kropparna testar detta.
- 11 Kollisionsinformationen är utökad till att inkludera den andra kroppen (som tidigare benämnts som orörligt föremål) och dess kontaktpunkt. Kollisionsbegränsningen tar detta i beaktande för att även begränsa hastigheten på den andra kroppen. Testklassen testar detta.
- 12 Den konkreta kollisionsdetektionen är generaliserad till att detektera kollision mellan en cirkulär kropp med en radie och en godtycklig linjesegment som sitter på en annan kropp. Testklassen testar detta.
- 13 Den konkreta kollisionsdetektionen är generaliserad till att detektera

kollision mellan en cirkulär kropp med en radie och ett antal linjesegment som sitter på en annan kropp. Testklassen testar detta.

14 Det finns en datatyp som representerar fysiken i en hel nivå, dvs. bollen och den fysiska banan. Tidssteg kan göras av ett objekt av denna typ. Detta är den huvudsakliga API:n till fysiken i spelet.

15 Det finns ett generellt ramverk för cirkelmålare samt linjesegmentmålare, samt default-konkretiseringar av dessa.

Den grafiska testklassen har utökats till att skapa en fysikalisk nivå och måla ut den med en cirkelmålare respektive linjesegmentmålare.

16 Det går att förflytta en nivå med musen.

17 Det går att rotera en nivå med musen.

18 Det finns (minst) en datatyp för att definiera grafer med hörn och kanter. Det går att genomföra olika sorters transformationer på grafer, såsom rotation och förstoring.

19 Det finns möjlighet att hämta och lagra grafer med JSON i textfiler.

20 Det finns en datatyp för en nivådefinition som innehåller en graf, nivåkropp (position och massa), spelare-kropp (position och massa), spelare-radie samt en vy-rektangel genom vilken spelet kommer synas.

21 Det finns möjlighet att lagra och hämta nivådefinitioner från filer

22 Det finns ett komplett spel där nivådefinitioner hämtas från filer i början, och där spelaren sedan spelar varje nivå ett i taget.

23 Det finns en grafisk nivåskapare (om jag hinner)

4. Övriga implementationsförberedelser

Det ska finnas en generell struktur för tidssteg, där avbrott kan ske under ett tidssteg som ändrar hur resten av tidssteget kommer se ut. Denna struktur kommer användas för att först få fysikaliska kroppar (bollen och nivån) att stega fram i tid, bli avbrutna av en kollisionshanterare som ändrar på kropparnas hastigheter, följt av att kropparna fortsätter sina tidssteg nu med ändrade hastigheter.

Många tvådimensionella vektoroperationer kommer göras och därför är det lämpligt att skriva en tvådimensionell vektor-klass.

5. Utveckling och samarbete

Min ambitionsnivå är att göra ett spel som utmanar mig och som är "feature-complete" eftersom många av mina tidigare spel är inkompleta. Jag siktar också på att nå högsta betyg.

Jag tänker till största delen arbeta för mig själv och googla när jag stöter på problem eftersom jag brukar vara som mest produktiv när jag tvingas läsa på saker själv.

Projektrapport

6. Implementationsbeskrivning

6.1. Milstolpar

Observera att spelet har genomgått en stor strukturell förändring från planeringen. T.ex. styrs nu nivån med tangenter och masscentrumet av nivån kan förflyttas med musen. Dessutom kan man förflytta och zooma in och ut kameran i spelet. Därför är endast vissa milstolpar lämpliga att gå igenom. Resten av milstolparna är ointressanta och ej genomförda.

Milstolpe 2 som handlar om en datatyp för 2D vektorer har genomförts via klassen "Vector2D" i "math" paketet.

Milstolpe 3 som bl.a. handlar om en datatyp för fysikaliska kroppar har genomförts via klassen "Body" i "physics" paketet.

Milstolpe 5 som bl.a. handlar om en generell struktur för hastighetsbegränsningar har huvudsakligen genomförts via klasserna "VelocityConstrainer", "ActiveVelocityConstraint" och "IterativeVelocityConstrainer". Dessa klasser och andra relaterade klasser finns i paketet "physics.constraint". En "spik constraint" har inte implementerats eftersom denna visade sig inte behövas.

Milstolpe 6 handlar om två typer av hastighetsbegränsningar som skapar "top-down" friktion för hastigheten respektive vinkelhastigheten av en fysikalisk kropp. Dessa begränsningar är implementerade via klasserna "AngularVelocitySeeker" och "VelocitySeeker" i paketet "physics.constraint.implementation". Som namnen antyder är dessa constraints inte bara ämnade att sträva efter hastigheten 0, utan vilken hastighet som helst.

Milstolparna 7-13 som handlar om kontinuerlig kollisionsdetektion har ej implementerats eftersom det var onödigt komplicerat för mitt spel. Istället har jag implementerat en generell struktur för icke-kontinuerlig kollisionsdetektion och kollisionshanterande i paketet "physics.collision". En konkret implementation av kollisionsdetektionen som ämnar att finna kollisioner mellan en cirkel "CircleCollider" och en linjesegment-figur "CustomCollider" finns i paketet "physics.collision.implementation".

Milstolpe 20 som handlar om implementation av en nivådefinition har åstadkommit via klassen "LevelDefinition" i "leveldefinition" paketet. Milstolpe 21 har också implementerats som handlar om att hämta och lagra nivådefinitioner från och till filer. Detta har gjorts via klassen "LevelIO", och både "GameRunner" och "CreatorRunner" anropar metoder från denna klass.

Milstolpe 22 som handlar om ett komplett spel där spelaren tar sig från en nivå till nästa har lagts in, huvudsakligen via klasserna "GameRunner", "GameWorld" och "LevelWorld" i "main.game" paketet.

Milstolpe 23 som handlar om en grafisk nivåredigerare har implementerats. Samtliga klasser i paketet "main.levelcreation" är här intressanta, men de mest centrala klasserna är "CreatorRunner", "CreateAndTestWorld" och "LevelCreator".

6.2. Dokumentation för programstruktur, med UML-diagram

6.2.1 Övergripande programstruktur

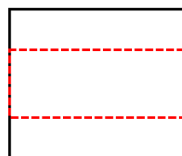
Klassen för att starta programmet är "Runner" i "main" paketet. Denna klass poppar upp en fråga om man vill spela spelet eller skapa en nivå. Beroende på valet man gör sker följande:

1. Spela spelet

"GameRunner"s "run" metod anropas som till att börja med hämtar samtliga nivåer i spelet från programmets resurser via "LevelIO". Dessa nivåer är i form av "LevelDefinition" objekt som definierar hur en nivå ser ut när den startas (efter nivån startats kommer givetvis saker och ting ändras, bollen kommer till exempel röra på sig).

Efter att "GameRunner" hämtat samtliga nivåer i spelet skapar den en "GameWorld" som representerar den aktuella spelvärlden. Nivådefinitionerna skickas med till GameWorld. En GameWorld kan lyssna på användarinput, måla ut sig själv på ett "Graphics2D" objekt givet en "RectangularRegion" där den ska målas ut, och stega fram i tiden givet en storlek på tidssteget. En GameWorld har dessutom en kamera i form av en RectangularRegion. Hur används då kameran? Jo efter att GameRunner skapat spelvärlden skapar den en "WorldGUI" som representerar det grafiska användargränssnittet. I detta fall kopplas detta GUI till GameWorld och GUIt gör följande:

- Målar GameWorld utifrån dess rektangulära kamera genom att transformera grafikobjektet. Eftersom bredd-till-höjd förhållandet för GUIt och kameran kan vara olika ser GUIt till att åtminstone visa det som täcks av kameran, men tillåter att mer av GameWorld målas ut, se figur 7. Att GUIt anpassar sig till GameWorlds kamera gör att GameWorld och resten av programmet inte behöver bry sig om pixlar. Denna "decoupling" är viktig, eftersom det tillåter resten av programmet (däribland fysiken) att tänka i meter istället för i pixlar.
- Stegar fram tiden i GameWorld regelbundet via en swing Timer. Storlekarna på tidstegen som skickas till GameWorld har sekunder som enhet.
- Talar om för GameWorld när användaren trycker/släpper en tangent eller musknapp, samt när musen scrollas. Talar också regelbundet om var musen är i spelvärlden (dvs. i spelvärldens logiska koordinatsystem, inte var musen är på skärmen pixel-mässigt).



Figur 7: Den röda rektangeln visar kameran av t.ex. en "GameWorld". Den svarta rektangeln visar vad en "WorldGUI" väljer att måla ut av spelvärlden. Här ser vi att GUIt väljer att visa mer av spelvärlden på höjden än vad spelvärldens kamera anger. Detta beror på att GUIt har ett annat bredd-till-höjd förhållande än kameran. GUIt väljer därför att visa den **minsta omslutande regionen till kameran som uppfyller GUIts bredd-till-höjd förhållande**.

GameWorld har en aktuell "LevelWorld" (egentligen en WorldWithMovableCamera<LevelWorld>, mer om detta senare) och delegerar de flesta metoderna till den. GameWorld lyssnar på intressanta händelser i sin aktuella LevelWorld genom att vara en "LevelListener". Här används alltså "observer" mönstret för att LevelWorld ska kunna prata till GameWorld utan att vara direkt kopplad till den. När GameWorld får reda på att spelaren förlorat i den aktuella nivån, slänger

GameWorld bort nivån och återskapar den. När GameWorld får reda på att spelaren vunnit nivån instansierar den nästa nivå och gör den till den nya aktuella nivån.

Som namnet "WorldGUI" antyder används GUI inte bara för att upprätthålla en GameWorld, utan den kan upprätthålla vilket objekt som helst som implementerar ramverket "FilmedWorld".

Paketet "main.world" anger de klasser som har med grafiska gränssnitt, världar och kameror att göra. Jag går igenom mer om detta senare, men innan dess ska vi titta på hur nivåskaparen startas upp.

2. Skapa nivå

"CreatorRunner"s "run" metod anropas som till att börja med frågar användaren om vilken .json fil som innehåller nivån som ska redigeras. En "LevelDefinition" laddas från filen via "LevelIO". Om en fil anges som inte finns i det lokala filsystemet, skapas en tom LevelDefinition. När en LevelDefinition fått fram, antingen genom att få tag på den från filen eller genom att skapa en ny tom sådan, skapas en "CreateAndTestWorld". En "WorldGUI" skapas för att upprätthålla denna värld via ett grafiskt gränssnitt.

CreateAndTestWorld börjar i create-läge med den givna nivådefinitionen som startpunkt. I create-läge delegerar denna värld till en "LevelCreator" (egentligen en "WorldWithMovableCamera<LevelCreator>", mer om detta senare). En LevelCreator är bl.a. en komposition av följande:

- "DrawableLevelBlueprint", som egentligen är en "LevelBlueprint" med utritningsfunktioner. En LevelBlueprint innehåller all data som är associerad till hur nivån under konstruktion ser ut just nu. Den innehåller en lista av hörn i form av "Vector2D" objekt, bollens start position, nivåns startkamera osv. En LevelBlueprint liknar en LevelDefinition på så sätt att den beskriver start läget av en nivå, men de lagrar data på lite olika sätt och har olika funktionalitet eftersom en LevelBlueprint kan ändras medan en LevelDefinition är oföränderlig efter dess konstruktion.
- "CommandTimeLine" som möjliggör "undo" och "redo" när man skapar nivån (via CTRL+Z respektive CTRL+Y). Här används "Command"-designmönstret, mer om detta senare.
- En aktuell "Mode" som är det läge som nivåskaparen befinner sig i. Ett exempel på ett Mode är "AddVertexMode" som placerar ut hörn till nivån när användaren klickar på musen. Ett annat exempel är "RemoveLineSegmentMode" som tar bort linjesegment som användaren klickar på. Här används alltså "State"-designmönstret för att minska kopplingar mellan modes och se till att allting som har med ett visst Mode att göra inkapslas till en egen enhet.

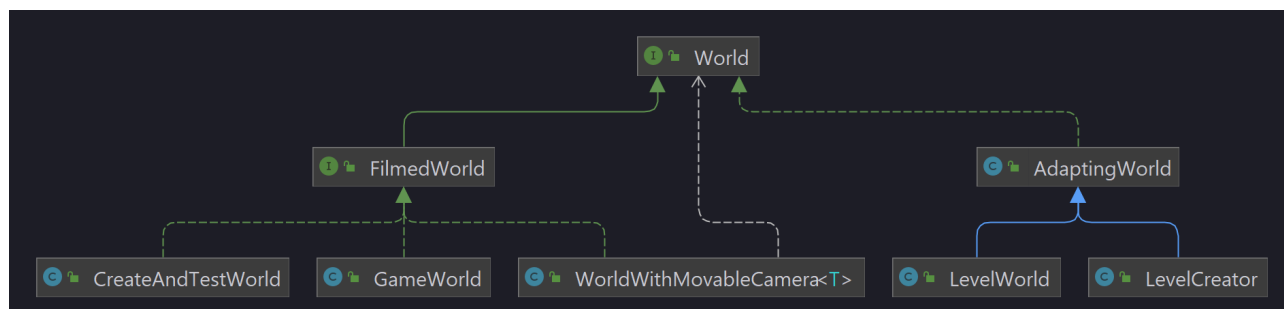
När användaren vill testa sin nivå trycker hen på ENTER knappen, vilket fångas upp av WorldGUI. GUI skickar vidare detta till sin värld, som i detta fallet är CreateAndTestWorld. I vanliga fall skulle CreateAndTestWorld endast delegera knapptrycket till sin LevelCreator, men eftersom just ENTER var tryckt byter den sin värld till en "LevelWorld" (egentligen en WorldWithMovableCamera<LevelWorld>) genom att:

1. Plocka ut LevelCreatorns LevelBlueprint
2. Konvertera blueprinten till en LevelDefinition
3. Instansiera en LevelWorld från nivådefinitionen
4. Peka ut den skapade nivåvärlden som den aktuella världen som framöver delegeras till

När ENTER trycks ner igen pekar CreateAndTestWorld tillbaka till sin LevelCreator och återgår till att delegera till den. Från utsidan verkar alltså CreateAndTestWorld "byta klass", vilket har uppnåtts genom tillämpning av "State"-designmönstret.

När CTRL+S trycks ner under skapandet (eller testandet) av en nivå sparas nivån på den fil som angavs när nivåskaparen startades. Eftersom det var "CreatorRunner" som till en början frågade om vilken nivåfil som skulle redigeras, så är det också den klassen som sköter sparandet av nivån (med hjälp av "LevelIO").

6.2.2 "main.world" paketet



Paketet "main.world" innehåller klasser och interfaces som används både när nivåer skapas och när själva spelet körs. "World" är ett interface som beskriver en två dimensionell värld som kan:

- Svara på användarinput, t.ex. när en tangent trycks ner eller när musen förflyttas.
- Stega fram i tid givet en storlek på tidssteget, "deltaTime". I programmet har deltaTime enheten sekunder.
- Måla sig själv på ett Graphics2D objekt vid en viss rektangulär region.

En World vet ingenting om pixlar, och när den får ett anrop att måla sig själv vid en viss region kommer den ta sitt logiska innehåll i den regionen (som tolkas i meter) och måla ut innehållet i grafikens koordinat system på exakt samma plats.

En World känner heller inte till kameror. T.ex. bryr sig "LevelWorld" inte om det finns en kamera som kollar ner på världen. Den bryr sig bara om vad den ska göra när den stegar fram i tid (applicera gravitation på bollen och låta sin "PhysicsEngine" sköta tidssteget, kollisioner mm.), vad den ska göra när den målar ut sig själv vid en given rektangulär region (måla ut bollen, nivån...) samt vad den ska göra vid användarinput (flyta nivåns masscentrum vid musklick och röra nivån vid tangenttryck). På samma sätt bryr sig inte "LevelCreator" om någon kamera.

En "FilmedWorld" är ett interface som ärver från "World" och lägger på en metod som returnerar en kamera i form av en "RectangularRegion". Varje "FilmedWorld" lovar liksom varje "World" att kunna måla ut sig själv vid en given rektangulär region, kunna stega fram i tid och kunna agera på användarinput, men dessutom att kunna ge ut en rektangulär region genom vilken den anses observeras (dvs. dess kamera). En "WorldGUI" upprätthåller en godtycklig FilmedWorld och varje gång GUI:t ska måla ut sin FilmedWorld hålls följande informella dialog:

GUI: "Hej världen, genom vilken rektangulär region anses du observeras?"

FilmedWorld: "Mellan dessa två x-koordinater och dessa två y-koordinater"

GUI: "Då ska jag transformera grafiken så att åtminstone den regionen syns på skärmen, men jag

kanske ber dig måla ut lite mer längs antingen x- eller y-axeln ifall jag har ett annat bredd-till-höjd förhållande jämfört med regionen du gav mig”.

Som tidigare nämnt har varken en ”LevelWorld” eller en ”LevelCreator” någon vetskap om en rektangulär kamera som tittar ner på dem. Men när man kör programmet kan man ju ändå flytta och förstora/förminska kameran både när man skapar en nivå och när man spelar en nivå. Detta förklaras via klassen ”WorldWithMovableCamera” som omsluter en godtycklig World och kombinerar den med en kamera (RectangularRegion). Eftersom en ”WorldWithMovableCamera” innehåller en World kan den göra allt en World kan göra via delegering (måla, stega fram i tid och hantera användarinput) men också ge ut en kamera. Därav implementerar

WorldWithMovableCamera gränssnittet FilmedWorld. WorldWithMovableCamera är en ”generic type” så att man kan få ut exakt vilken konkret typ av World som den omsluter. Förutom att delegera den användarinput som rapporteras till en WorldWithMovableCamera till sin omslutna World, så ser den till att ändra på sin kamera vid följande tillfällen:

- När användaren trycker och håller ner höger musknapp, förflyttas kameran i motsatt riktning till musens förflyttning.
- När användaren scrollar musen, förstoras/förminskas kameran.

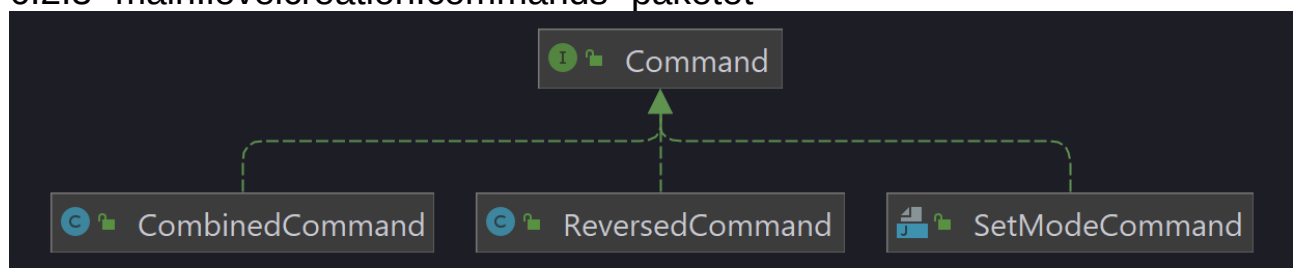
Sammanfattningsvis möjliggör WorldWithMovableCamera att vilken World som helst kombineras med en kamera som kan förflyttas med musen. Dessutom slipper man som utvecklare tänka på kameran när man implementerar en ny World.

En ”GameWorld” är en ”FilmedWorld”. För varje metod som GameWorld ärver från FilmedWorld delegerar GameWorld till sin aktuella WorldWithMovableCamera<LevelWorld>.

WorldWithMovableCamera<LevelWorld> är alltså en LevelWorld kombinerat med en kamera som kan ändras med musen. På liknande sätt är ”CreateAndTestWorld” en FilmedWorld som antingen delegerar till en WorldWithMovableCamera<LevelCreator> eller en WorldWithMovableCamera<LevelWorld> beroende på om den är i create- eller i test-läge.

Den sista saken jag vill nämna om ”main.world” paketet är att ”AdaptingWorld” är en klass som implementerar World ramverket och förser samtliga nedärvda metoder med tomma implementationer. Detta undviker redundanta tomma implementationer för de som ärver från AdaptingWorld.

6.2.3 ”main.levelcreation.commands” paketet



Paketet ”main.levelcreation.commands” handlar om kommandon som görs på en ”LevelCreator”. En central klass i detta paketet är ”CommandTimeLine” som håller reda på en tidslinje av genomförda kommandon och var nivåskaparen befinner sig på den tidslinjen. En CommandTimeLine har en lista av ”Command” objekt. Command är ett interface, och varje Command lovar att:

- Kunna exekveras (execute) på en given LevelCreator för att ändra dess tillstånd.
- Kunna ångras (undo) på en given LevelCreator för att ta tillbaka tillståndet av LevelCreatorn till hur det var innan kommandot exekverades. Detta löfte bygger på att LevelCreatorn befinner sig i samma tillstånd som inträffade direkt efter att kommandot exekverades.

När ett objekt, t.ex. ett "Mode", ber en LevelCreator exekvera ett kommando, ångra ett kommando (undo) eller göra om ett kommando (redo) delegerar LevelCreatorn till sin CommandTimeLine som gör följande:

När ett kommando exekveras:

1. Alla ångrade kommandon efter det "nuvarande" kommandot glöms bort
2. Det nya kommandot läggs i slutet av listan
3. Pekaren på det "nuvarande" kommandot hoppar fram ett steg så det pekar på det nya kommandot.
4. Det nya kommandot exekveras

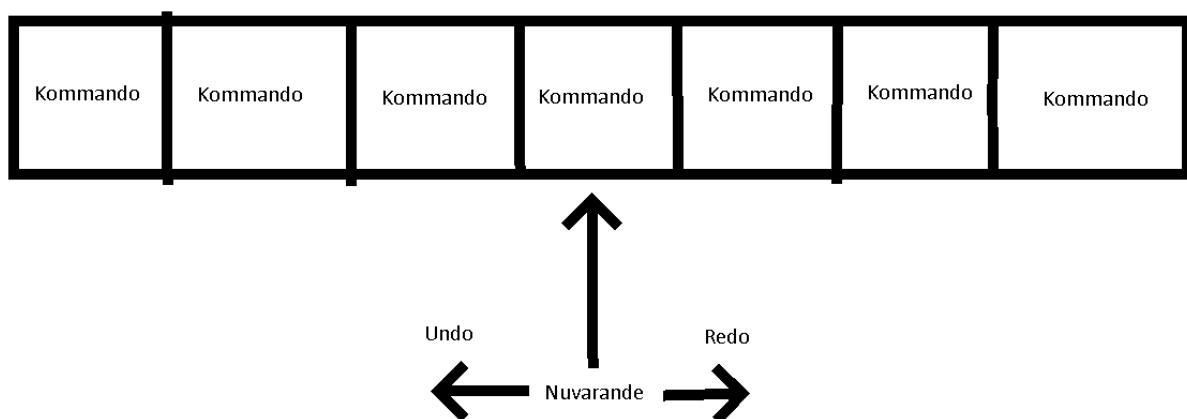
När ett kommando ångras (undo):

1. Det "nuvarande" kommandot ångras
2. Pekaren på det "nuvarande" kommandot hoppar bak ett steg

När ett kommando görs om (redo):

1. Pekaren på det "nuvarande" kommandot hoppar fram ett steg
2. Det "nuvarande" kommandot exekveras

Det "nuvarande" kommandot är alltså det kommando som LevelCreatorn exekverat sist om vi tänker oss att ångrade kommandon aldrig har exekverats.



"SetModeCommand" är en konkretiserad Command som vid dess skapelse lagrar LevelCreatorns nuvarande "Mode" och ett nytt Mode som den ska byta till under exekvering. När kommandot ångras ser den till att LevelCreatorn byter tillbaka till det Mode som LevelCreatorn hade under kommandots skapelse.

Ett "ReversedCommand" är ett Command som gör motsatsen till ett ursprungs Command. När det omvända kommandot exekveras, ångras ursprungskommandot. När det omvända kommandot

ångras, exekveras ursprungskommandot. ReversedCommand klassen undviker redundant kod där motsatsen till ett kommando ska exekveras. Jag visar exempel på detta snart. Ett annat Command är "Combined Command" som exekverar/gör om ett godtyckligt antal kommandon åt gången.

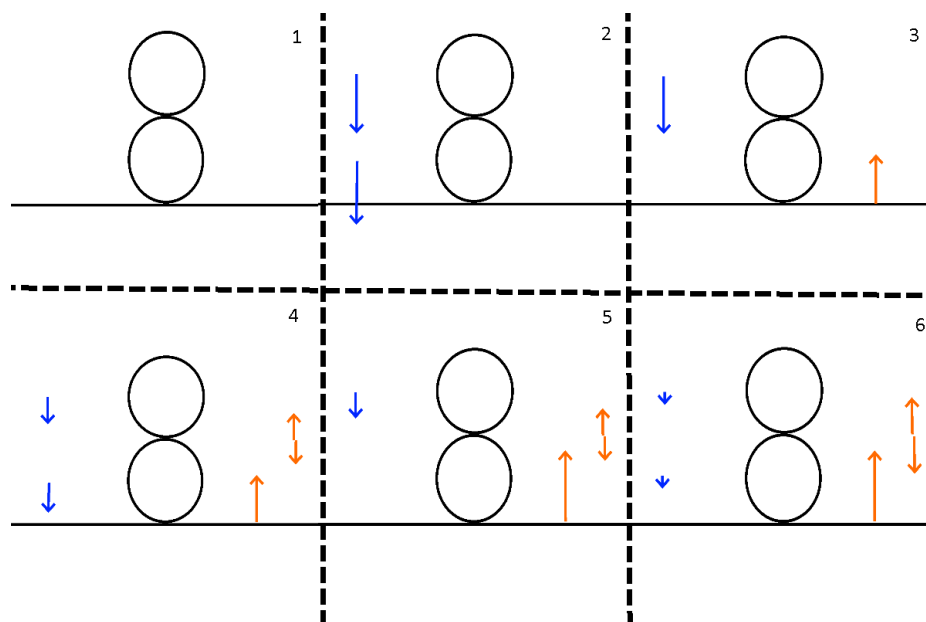
Varje "Mode" i koden har i dagsläget minst en nästlad klass som implementerar Command (dessa syns inte i UML diagrammet eftersom de är privata). T.ex. har "AddVertexMode" en "AddVertexCommand" klass. När denna inre klass instansieras tar den emot ett hörn i form av en "Vector2D". När kommandot exekveras läggs hörnet till på slutet i LevelCreatorn (som delegerar till sin "DrawableLevelBlueprint" som i sig delegerar till sin "LevelBlueprint"). När kommandot ångras tas det sista hörnet i LevelCreatorn bort, dvs. hörnet som lades till givet att tillståndet inte ändrats från exekveringen. AddVertexMode har en funktion som gör att om man har påbörjat en ny linjesegment (starthörn utan sluthörn) och trycker på ESCAPE så tas starthörnet på den inkompleta linjesegmenten bort. Om man sedan ångrar genom att trycka CTRL+Z läggs starthörnet in i LevelCreatorn igen. Detta borttagandekommando är alltså motsatsen till "AddVertexCommand": När man exekverar kommandot tas hörnet bort, och när man ångrar kommandot läggs hörnet till. Här utnyttjas "ReversedCommand".

6.2.4 "physics" paketet

Paketet "physics" innehåller de fysikbaserade aspekterna av programmet. Tanken är att detta paket ska vara nästintill helt fristående från resten av programmet så att det kan användas i andra syften och sammanhang. En central klass är "Body" som representerar en tvådimensionell fysikalisk kropp med en massa, position, hastighet, vinkel, vinkelhastighet och "vinkelmassa" (egentligen tröghetsmoment). En Body har själv ingen kollisionsfigur, och kan därför även användas i sammanhang där kollisioner inte är intressanta. En Body kan utföra ett diskret tidssteg genom att ändra på dess position och vinkel utifrån dess hastighet respektive vinkelhastighet.

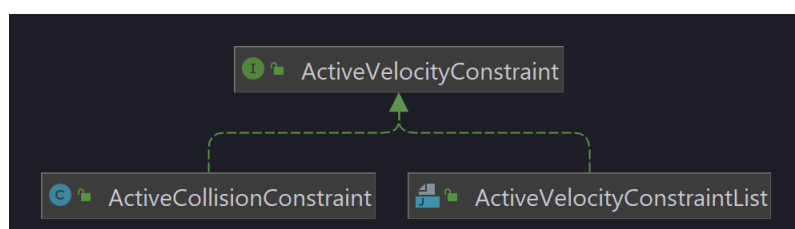
En "PhysicsEngine" innehåller ett godtyckligt antal fysikaliska kroppar. Fysikmotorns huvuduppgift är att genomföra diskreta tidssteg. I början av varje tidssteg ser fysikmotorn till att kropparnas hastigheter är begränsade så att t.ex. kroppar som kolliderar får hastigheter i motsatta riktningar. Efter att hastigheterna blivit begränsade ändras positionerna på samtliga kroppar genom att delegera till kropparnas egna tidsstegsmetoder.

Hur begränsas då kropparnas hastigheter i början av varje tidssteg? Jo fysikmotorn ber sin "IterativeVelocityConstrainer" att försöka finna en global lösning till samtliga hastighetsbegränsningar. När man löser en hastighetsbegränsning genom att ändra på de involverade kropparnas hastigheter finns det risk att man förstör för en annan hastighetsbegränsning eftersom de två begränsningarna kan verka över gemensamma kroppar. IterativeVelocityConstrainer försöker approximera en global lösning för samtliga hastighetsbegränsningar genom att flera gånger gå igenom varje begränsning och lösa dem var för sig. Här följer ett exempel på hur detta iterativa försök till en global lösning kan se ut:



Figur 8

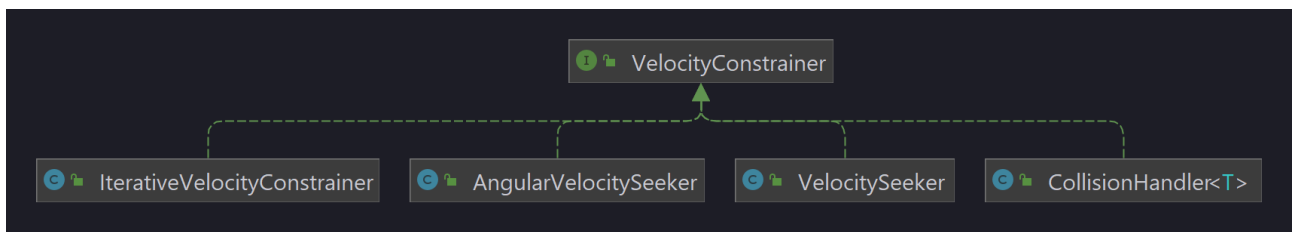
Figur 8 visar hur en global lösning till två hastighetsbegränsningar approximeras iterativt. De blåa pilarna indikerar kropparnas hastigheter och de orangea pilarna visar impulserna som hastighetsbegränsningarna applicerar för att ändra på hastigheterna. Steg 1: Två cirkelar med samma massa ligger staplade på ett fast plan. Det finns två hastighetsbegränsningar. Den ena är mellan planen och den undre cirkeln och försöker göra den undre cirkeln hastighet till noll. Den andra är mellan cirkelarna och försöker göra den relativa hastigheten mellan cirkelarna till noll. Steg 2: Gravitation appliceras på cirkelarna, vilket ger dem samma hastighet neråt. Den undre hastighetsbegränsningen är nu bruten. Steg 3: den undre hastighetsbegränsningen tillfredsställs. Den övre hastighetsbegränsningen är nu bruten. Steg 4: Den övre hastighetsbegränsningen tillfredsställs, den undre hastighetsbegränsningen är nu bruten. Steg 5: Den undre hastighetsbegränsningen tillfredsställs. Den övre hastighetsbegränsningen är nu bruten. Steg 6: Den övre hastighetsbegränsningen tillfredsställs, den undre hastighetsbegränsningen är nu bruten. Efter steg 6 kan vi se att den undre hastighetsbegränsningen är bruten, men betydligt mindre än vad den var i steg 2. När fysikmotorn sedan förflyttar kropparna längsmed deras hastigheter kommer dem sjunka ner i marken lite, men inte mycket eftersom tidssteget är kort ($1 / 90$ sekunder).



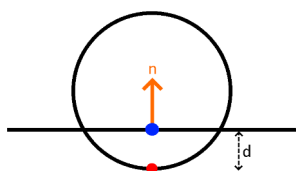
(Det finns dessutom några inre anonyma klasser som implementerar ActiveVelocityConstraint)

I programmet implementerar varje sorts hastighetsbegränsning "ActiveVelocityConstraint". En ActiveVelocityConstraint håller koll på dess impuls som den i ögonblicket applicerar på dess kroppar, och man kan anropa en "updateImpulse" metod för att få begränsningen att uppdatera sin impuls utifrån de involverade kropparnas nuvarande hastigheter så att begränsningen tillfredsställs.

Det finns ett annat liknande interface med namnet "VelocityConstrainer". En VelocityConstrainer lovar att kunna **generera** en ny ActiveVelocityConstraint som ska verka innan kropparna genomgår ett tidssteg av en given storlek. Exempel på en sådan är "VelocitySeeker" som kan generera en ActiveVelocityConstraint som strävar att ge en Body en viss hastighet, men är begränsad av en maximal kraft. Denna används i kombination med den snarlika "AngularVelocitySeeker" för att kunna styra en nivås kropp i spelet (se "VelocityController"). Även IterativeVelocityConstrainer, som vi har pratat om, är en VelocityConstrainer och den delegerar till en lista av andra VelocityConstrainers vars genererade ActiveVelocityConstraints itereras över.

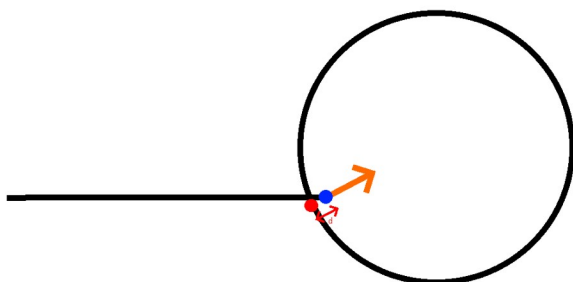


En annan typ av VelocityConstrainer är "CollisionHandler". En CollisionHandler syftar till att begränsa hastigheterna hos kroppar som kolliderar. För att ta reda på vilka kroppar som kolliderar, samt annan information om kollisionerna (t.ex. exakt vilka punkter av kropparna som kolliderar) frågar kollisionshanteraren sin godtyckliga "CollisionDetector" som är ett interface. En konkret typ av CollisionDetector är "CircleVsCustomCollisionDetector" som kan detektera kollisioner mellan en kropp med en bunden cirkel ("CircleCollider") och en kropp med en bunden figur bestående av flera linjesegment ("CustomCollider"). Denna kollisionsdetekterare går igenom varje linjesegment och kollar om cirkeln överlappar med segmentet. Om så är fallet tar den reda på kollisionsnormalen (riktningen på kollisionen), kollisionspunkten på cirkelkroppen samt på linjesegmentkroppen och dessutom hur mycket cirkeln överlappar linjesegmentet längs normalen enligt följande bild.



Den orangea pilen är kollisionsnormalen. Distansen d är hur mycket överlapp som sker längs normalen. Den blåa punkten är kontaktpunkten på linjesegmentkroppen och den röda punkten är kontaktpunkten på cirkeln.

Kollisionen kan också se ut så här:

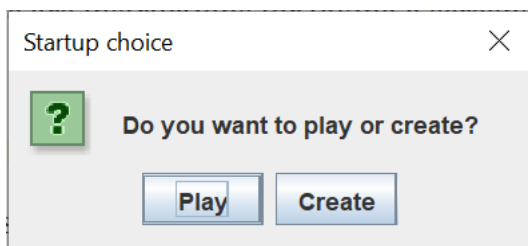


En CollisionHandler tar del av denna kollisionsdata via "CollisionData" och begränsar hastigheterna på kropparna både genom att applicera en impuls längs normalen för att få

kropparna att separeras, men också genom en impuls som är rätvinklig till normalen för att skapa friktion. Utan friktion skulle t.ex. bollen i spelet inte rulla (detta kan testas genom att sätta friktionskoefficienten i CollisionData till 0). En viktig aspekt av kollisionshanteringen är att den faktiska hanteringen av kollisioner är "decoupled" från kollisiondetektionen. Detta gör det möjligt att implementera nya kollisiondetektionsalgoritmer (t.ex. cirkel mot cirkel) utan att CollisionHandler behöver ändras. CollisionHandler behöver bara peka ut den nya konkreta implementationen av en CollisionDetector. En annan aspekt med en CollisionHandler är att den har ett antal lyssnare som den förmedlar CollisionData till. CollisionData kan dessutom ha data som är specifik för en viss typ av kollisiondetektorer, och spelet använder detta för att veta när bollen i spelet kolliderar med olika typer av linjesegment (svart, röd eller grön).

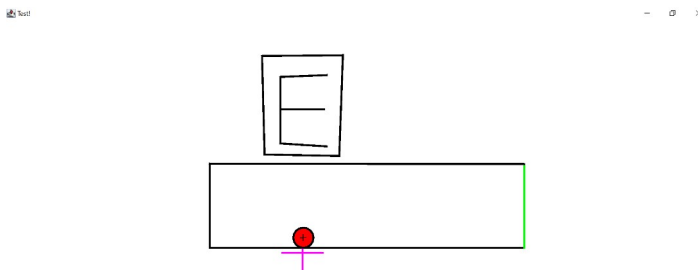
7. Användarmanual

Kör "Runner" som finns i paketet "main". Då bör följande ruta dyka upp:



7.1 Spela spelet

Tryck på "Play" för att starta det faktiska spelet. Du hamnar då i den första banan, vilket ser ut så här:



Du kontrollerar det rosa/lila krysset på följande sätt.

Tangent A → förflytta krysset vänster

Tangent D → förflytta krysset höger

Tangent W → förflytta krysset uppåt

Tangent S → förflytta krysset neråt

Tangent Q → rotera krysset åt vänster (moturs)

Tangent E → rotera krysset åt höger (medurs)

Du kan också omplacera krysset genom att med musen klicka där du vill placera den. Som du märker följer nivån med krysset när du förflyttar och roterar den.

Viktigt: För att starta om en nivå, tryck ENTER

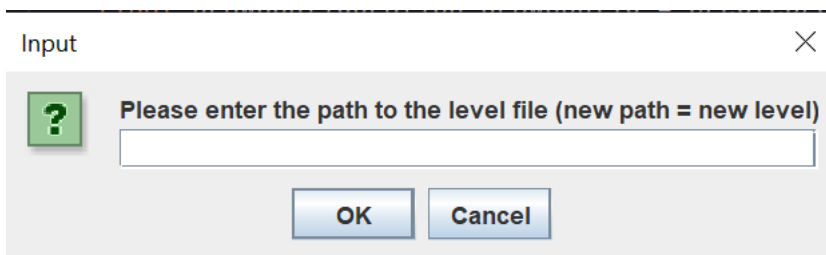
Du kan förflytta kameran genom att hålla ner höger musknapp och svepa i den riktning du vill förflytta världen åt. Du kan också zooma in och ut genom att scrolla mushjulet.

Ditt mål är att få bollen att nudda nivåns gröna plattform för att komma till nästa nivå, och du ska undvika att falla ner eller nudda röda plattformar. Ha så kul!

När du spelar spelet, tänk på att krysset du styr är masscentrumet av nivån och att när nivån applicerar en kraft på bollen så applicerar bollen en lika stor fast motriktad kraft på nivån. Nivån har en mycket större massa än bollen, men i vissa lägen kommer du märka att bollen knuffar undan nivån (speciellt när masscentrumet är långt ifrån bollen!).

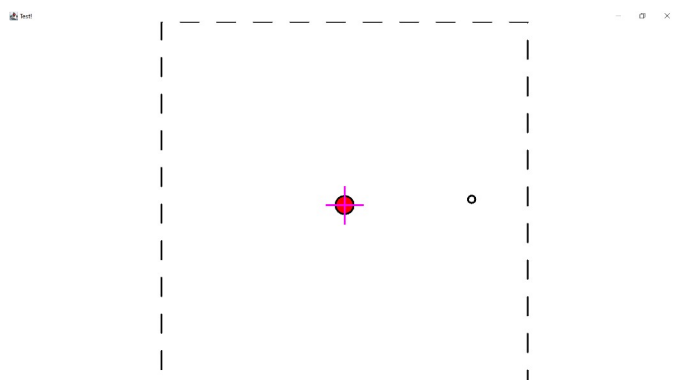
7.2 Skapa en nivå

Tryck på "Create" i den första rutan för att skapa en nivå. Då presenteras följande ruta:



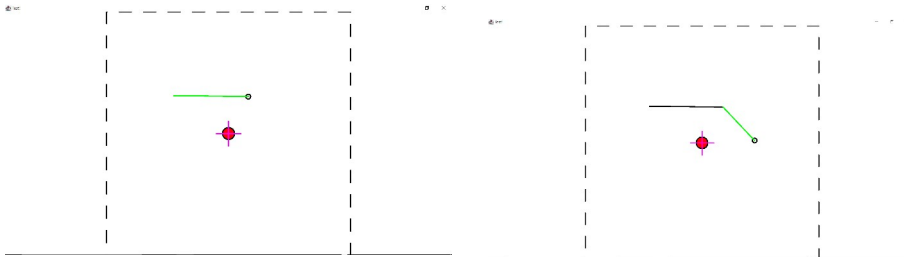
Om du vill påbörja en ny nivå skriver du in en sökväg till en .json fil som inte finns i ditt lokala filsystem. Där kommer nivån senare sparas. Om du vill ändra på en nivå som finns lagrad i ditt filsystem anger du dess sökväg.

Efter du har angett rätt sökväg klickar du på "OK". Då hamnar du i redigeringsläge som ser ut på följande vis om du påbörjade en ny nivå:



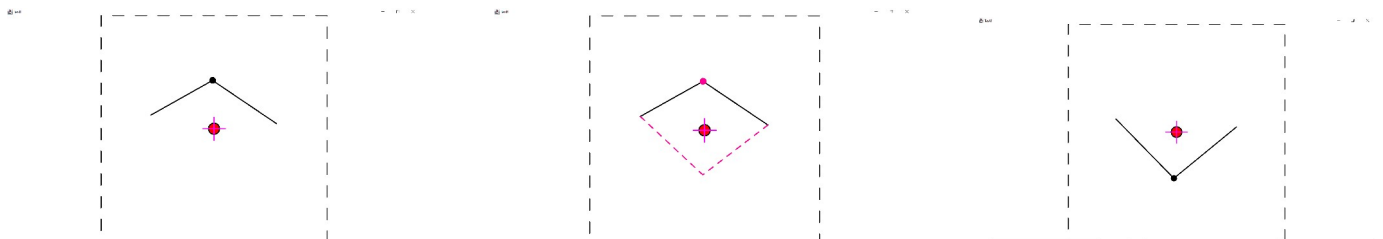
Även här kan du flytta på kameran genom att hålla ner höger musknapp och svepa med musen, samt zooma in och ut med mushjulet. Om du klickar på vänster musknapp märker du att ett grönt streck från där du klickade på musen till musens nuvarande position dyker upp. Om du trycker på musen igen blir strecket svart och det är då en plattform. Då dyker också ett nytt grönt streck upp. Detta är hjälpsamt om du vill göra en kedja av linjesegment. Du kan stänga av/sätta på denna

funktion genom att trycka på C ("Chain"). Om du trycker på ESCAPE försvinner det gröna strecket utan att "chain" funktionen stängs av eller sätts på. Testa att sätta ut ett flertal linjesegment, och bli bekant med hur du avslutar och påbörjar kedjor av linjesegment. När du har ett flertal linjesegment på skärmen, testa att hålla ner SHIFT. Som du säkert märker gör detta det möjligt att påbörja/avsluta ett linjesegment vid ett hörn som redan är utplacerat.

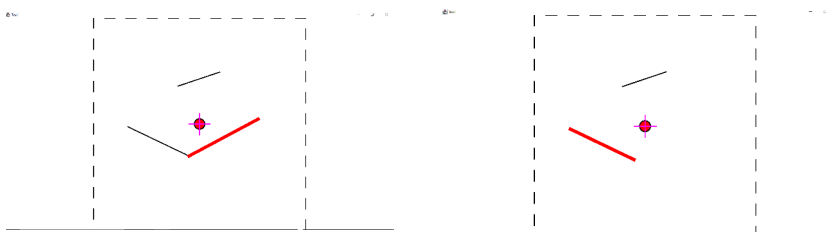


Nu ser det säkert ganska rörigt ut på skärmen, så tryck CTRL+Z några gånger för att backa bakåt i tiden. Om du vill hoppa framåt i tiden igen, tryck CTRL+Y. Att hoppa tillbaka och fram i tiden fungerar även i alla andra lägen som jag nu kommer gå igenom ett i taget.

Nu ska vi testa att förflytta ett hörn. Tryck på siffran 2 på tangentbordet. Du bör nu se en svart prick vid det närmaste hörnet till musen. Tryck på musen för att markera hörnet. Då ser du hur hörnet och dess kanter kommer ändras efter förflyttningen. Tryck på musen igen för att genomföra förflyttningen. För att avsluta markeringen av ett hörn, tryck ESCAPE. Om du trycker på siffran 1 på tangentbordet kommer du tillbaka till läget där du placerar ut hörn.



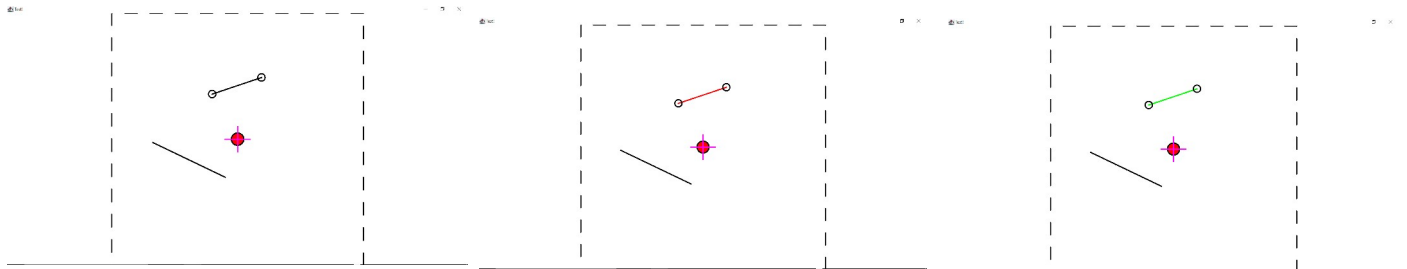
Nu vill vi kunna ta bort specifika linjesegment utan att behöva backa bakåt i tiden. För att göra detta trycker du på siffran 3 på tangentbordet. Då bör det närmaste linjesegmentet till musen markeras med röd färg. Tryck på musen för att ta bort linjesegmentet.



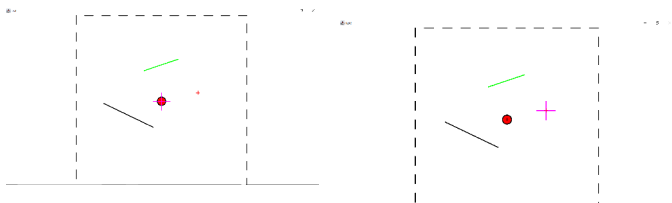
Nu ska vi ändra typ av linjesegment. Spelet har tre typer av linjesegment:

- Svart: En vanlig plattform som spelaren (bollen) kan kollidera med
- Röd: En plattform som spelaren vill undvika, annars tvingas spelaren börja om nivån
- Grön: Målet i nivån, som gör att spelaren tar sig vidare till nästa nivå

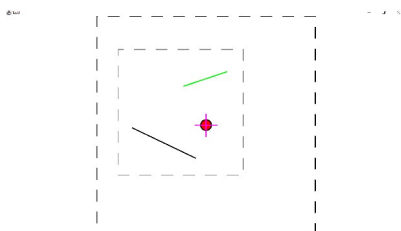
Börja med att trycka på siffran 4 på tangentbordet. Nu bör det närmaste linjesegmentet till musen markeras med två cirkclar runt dess hörn. Tryck på musen för att byta segmentets typ till Röd. Tryck på musen igen för att byta segmentets typ till Grön. Tryck ytterligare en gång för att återgå till ett vanligt svart linjesegment.



Nu är det dags att flytta på nivåns masscentrum. Tryck på tangenten med siffran 5. Ett rött kryss bör nu följa musen. När du trycker på musen sätts rotationspunkten där. Detta läge är endast till för att ange var nivåns masscentrum ska börja, men när man spelar nivån kan man ju placera om den fritt.



Det sista steget är att ändra startkameran. Tryck på siffran 6. Tryck och håll sedan ner musen samtidigt som du förflyttar musen i valfri riktning. Du ser nu hur kameran kommer bli när du släpper musen. Om du vill avbryta en aktiv utplacering av kameran, tryck ESCAPE. Tänk på att kameran är den minsta regionen som kommer visas på skärmen, men p.g.a. att bredd-till-höjd förhållandet kan vara olika mellan kameran och GUI:t kan mer visas av nivån.



Bra jobbat! Nu har du gjort din första nivå! Du kan testa nivån genom att trycka på ENTER. Är nivån inte särskilt rolig än? Tryck på ENTER en gång till för att hoppa tillbaka till redigeringsläget. Gör en rutschkana eller någonting annat kul som du kommer på! När du känner dig nöjd kan du spara nivån genom att trycka CTRL+S. Nivån sparas på den plats du angav när du startade redigeraren.