Plix: En spelmotor

En projektrapport av Marcus Stenbeck i kursen Teknik för avancerade datorspel (TSBK03).

marst241@student.liu.se

Länkar

Källkod

Koden för projekten finns på GitHub.

Plix: https://github.com/marcusstenbeck/plix

physix: https://github.com/marcusstenbeck/physix

fsm: https://github.com/marcusstenbeck/fsm

Spelbara exempel

Spelen finns att spela på CodePen.

Pong

Kontroller: A/S och K/L.

http://codepen.io/marcusstenbeck/pen/yYgRBa

Jump Dude

Kontroller: A/S och K.

http://codepen.io/marcusstenbeck/pen/YXwNXZ

Introduktion

Projekten i denna kurs fokuserar oftast på en specifik avancerad teknik. För mig var det mer intressant att studera helheten när man bygger en spelmotor från grunden. Webbteknik ligger mig nära om hjärtat och i dagens webbläsarsituation finns många JavaScript API:er som tillsammans uppfyller alla lågnivåfunktioner som behövs för att skriva en spelmotor.

Till hjälp längs vägen har jag konsulterat boken Game Programming Patterns, Design Patterns och specifikationsdokumentet för WebGL 1.1. En del inspiration till spelmotorns uppbyggnad är tagen från koden för spelmotorn Goo Engine.

Det finns en hel del kodexempel i denna rapport och det mesta är helt vanlig JavaScript tagen direkt ur spelmotorn, men förenklad för tydlighets skull. Här och där finns också kod som inte är valid JavaScript, men där ändringarna har gjorts för

att förtydliga. Till exempel skriver jag ofta funktioner som `myFunction(Time t, Float size, Body b)` för att understryka vilken typ av argument funktionen förväntar sig.

Designmönster och metoder som använts

Det stora arbetet i att konstruera en spelmotor är att skapa ett system av byggklossar som går att återanvända i skapandet av flera olika typer av spel. Det har gjorts många gånger innan och lösningarna är lika många som meningsskiljaktigheterna kring vilka som är bäst. Jag listar i detta kapitel de designmönster eller metoder som jag medvetet har implementerat, samt vad vardera är tänkt att lösa.

Game Loop / Update function

Det enklaste mönstret som fortfarande är värt att lyftas fram är *Game Loop*. Funktionen uppdaterar spelet mellan varje gång det ritas eller skrivs ut på skärmen, och det händer antingen vid ett fixt intervall eller vid specifika händelser. För turbaserade spel är det t. ex. när en schackpjäs har flyttats, och för realtidsspel händer det automatisk många gånger per sekund. Jag har hittils inte hittat ett enda spel eller spelmotor som inte har en uppdateringsfunktion.

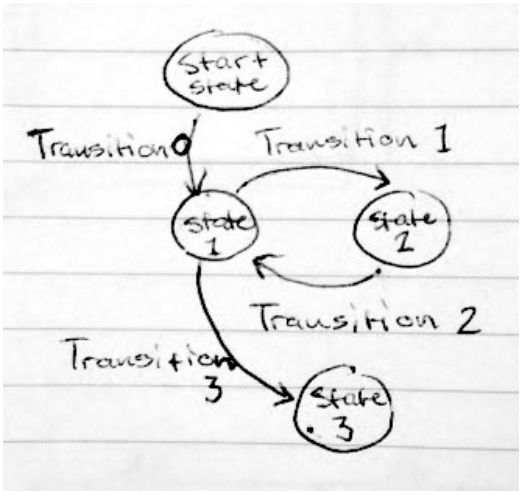
Alla som är intresserade av att skapa realtidsspel utan att använda en befintlig spelmotor kommer att programmera en *Game Loop*-funktion väldigt tidigt. I JavaScript kan man börja, och komma relativt långt, med endast följande kod. Detta är en aningen förenklad variant av den kod jag började med.

```
function update(time) { /* ... */ }

function draw() { /* ... */ }

function loop(time) {
  // Be om att köra funktionen igen
  requestAnimationFrame(loop);

  // Uppdatera spelets tillstånd
  update(time);

  // Rita spelet
  draw();
}

// Starta loopen
loop(0);
```

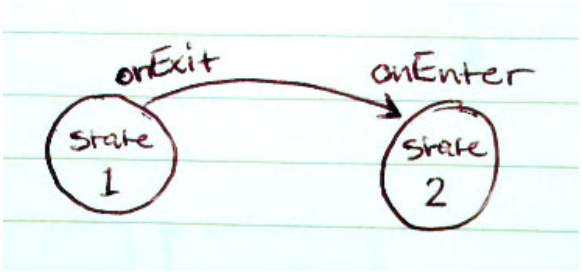
Här finns tre funktioner; `update(Float time)`, `draw()` och `loop(Float time)`. I den första funktionan samlas kod som uppdaterar spelets tillstånd, i den andra kod som hanterar att rita ut spelet i dess nuvarande tillstånd. Den sista funktionen, `loop (Float time)`, kallar de två föregående samt ber webbläsaren att kalla funktionen igen vid webbläsarens nästa uppdatering (i dess egen uppdateringloopfunktionalitet!) med `requestAnimationFrame(Function f)`.

Finite State Machine (FSM)

FSM är ett vanligt sätt att hålla reda på tillstånd i diverse program. Det finns ett antal tillstånd något kan befinna sig i. Ett specifikt tillstånd har regler för när det flyttar till ett annat tillstånd. Vid en viss input reagerar tillståndet endast om det har en regel (en *transition*) för den input den får.



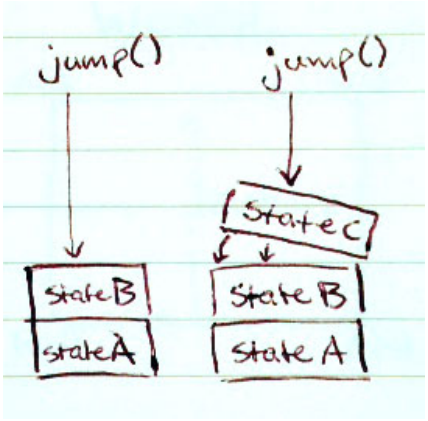
Om vi tänker oss att input är knapptryckningar så kan spelarens figur reagera på olika sätt beroende på vilket tillstånd figuren är i när knapptryckningen händer. Det trevliga är att man kan skicka alla knapptryckningar till figuren och vara säker på att den bara reagerar på de kommandon som för tillfället är har *transitions*. Tillstånden dikterar inget mer än vilken input som orsakar en förflyttning till ett annat tillstånd. Det är upp till programmeraren att fylla tillstånden med annat.



I min implementation finns två funktioner som en spelprogrammerare kan använda för att konstruera ett beteende. När ett tillstånd går från inaktivt till aktivt och tvärtom så exekveras in- och utträdesfunktioner, om de finns. Under tiden däremellan, när tillståndet är aktivt, så händer ingenting. Däremot kan in- eller utträdesfunktionen ha ändrat beteendet hos något i spelet.

Stack för scenes

Inspiration till denna metod är hur spelmotorn *Cocos2D* hanterar *scenes*. En *scene* i *Cocos2D* är ett objekt som representerar en typ av behållare för allt som kan tänkas existera samtidigt, t. ex. allt i en spelbana. Dessa *scenes* läggs i *Cocos2D* ovanpå varandra som lager och när det översta lagret försvinner återgår *Cocos2D* till ett eventuellt undre lager. Med inspiration från *Cocos2D* och kapitlet om *state* i Game Programming Patterns upptäckte jag något som kallas *pushdown automata* som är basen för min scenstack.



Pushdown automata fungerar ungefär som en LIFO stack. Det som är aktivt ligger uppe på toppen och hanterar all input som kommer in. När något i scenen signalerar att det är dags att ta av den översta scenen plockas den bort och stacken börjar använda det som ligger näst under i stacken. Alternativ kan man omedelbart lägga på något på toppen, t. ex. när man byter bana. På så vis kan ett standardläge alltid ligga på botten av stacken, och ovanpå det kan man slänga på en bana. Utöver detta så kan spelaren få för sig att pausa spelet — då kan man helt enkelt lägga på "pausläget" överst i stacken. Nackdelen med denna approach är att om man inte är försiktig och bygger en för "hög stack" så äter man snabbt upp arbetsminnet.

Entity / Component / Builder

Entity / Component

När man programmerar objektorienterat så finns ett antal sätt att återanvända kod. Troligen det vanligast förekommande exemplet på återanvändning av kod i objektorienterade språk är klassarv. I spelvärlden kan detta se ut som att fiender härstammar från en klass `Enemy`, och utökar eller beskriver en viss typ av fiende i subklasserna `BigEnemy` och `SmallEnemy`. När det är väldigt få antal variationer som i exemplet så kan arv vara utmärkt, men när antalet variationer ökar eller när en `Enemy` delar funktionalitet med saker som inte är fiender så finns en risk att antalet klasser blir svårhanterligt många. Dessutom finns det stor risk att vissa klasser innehåller funktionalitet som de inte behöver.

En lösning till detta är att använda designmönstrena Entity och Component tillsammans. Dessa kompletterande mönster bryter ut funktionalitet i komponenter som kan kombineras ihop i en så kallad *entity* för att skapa olika typer av "saker". Man kan föreställa sig en *entity* som en "grej" som existerar i spelvärlden. Exakt vad det är för något beror helt och hållet på vilka komponenter den innehåller. Det är kombinationen av komponenter som gör en *entity* till en fiende, pickup, vägg, pistolkula eller spelare.

Spelmotorn håller reda på och hanterar alla *entitys* som existerar och ber dem att uppdatera sig själva. Detta kan t. ex. ske direkt i en spelmotors huvudklass eller i en klass som har som sitt syfte att hålla reda på alla *entitys*, en s.k. *Entity Manager*.

Builder/Factory pattern

Eftersom fiendetyperna `BigEnemy` och `SmallEnemy` inte längre är sina egna klasser så kan de inte längre instansieras med en rad kod, e.g. `new BigEnemy()`. När man använder Entity/Component krävs det att man skapar en `Entity`-klass samt alla komponentklasser som ska användas. Dessutom behöver varje komponentinstans "fästas" i en entity. Det finns med andra ord risk att mycket kod upprepas i onödan.

För att råda bot på detta kan man använda designmönster som Builder eller Factory. Dessa enkapsulerar skapandet av mer komplicerade objekt till en enstaka funktion som innehåller all kod som behövs för att skapa objektet. Kodmängden som behövs för att skapa en viss sak reduceras till ett funktionsanrop istället för många. I likhet ger man som kund på ett café sällan instruktioner steg-för-steg utan oftast tittar man på menyn och beställer t. ex. en "cappucino".

```
var myApple = AppleFactory.create({
                type: 'Pink Lady',
                peeled: false
              });

var myCitrusBuilder = (new CitrusBuilder())
                        .setType('Blood Orange')
                        .setPeeled(true);

var myFirstOrange = myCitrusBuilder.build();
var mySecondOrange = myCitrusBuilder.build();
```

Skillnad mellan *Builder* och *Factory* är att den föregående definieras som ett enda funktionsanrop. *Builder* skapar istället ett objekt vars funktioner används för att konfigurera objektet det ska skapa, och slutligen ber man *Builder*-instansen om det färdiga objektet.

Fysikmotorn

En spelmotor och en fysikmotor är oftast två skilda moduler. En fysikmotor används ofta, men inte alltid, när man skapar ett spel, och därför passar den bra att bryta ut som separat komponent. Med det som motivering utvecklade jag en enkel fysikmotor, *physix*, som en separat modul som jag sedan använde i min spelmotor *Plix*. Under skapandet av *physix* har jag tagit inspiration av hur fysikmotorn *Box2D* fungerar.

Simulering

Varje simuleringssteg görs med Euler-integrering, e.g. man mäter tiden som fortlöpt sedan föregående beräkningssteg och beräknar nya positioner för alla fysikkroppar med de klassiska fysikaliska formlerna för mekanisk rörelse. Krafter appliceras på fysikkroppen `Body`, och sedan beräknas kroppens nya acceleration, hastighet och position i fysikmotorns integrationssteg.

```
    // Calculate acceleration
    switch(body.type) {
      case Body.DYNAMIC:
        body.acc.x = this.gravity.x + (body.accumulatedForce.x / body.mass);
        body.acc.y = this.gravity.y + (body.accumulatedForce.y / body.mass);
        break;
      case Body.KINEMATIC:
        body.acc.x = body.accumulatedForce.x / body.mass;
        body.acc.y = body.accumulatedForce.y / body.mass;
        break;
    }

    // Zero out accumulated force
    body.accumulatedForce.x = 0;
    body.accumulatedForce.y = 0;

    // Calculate velocity
    body.vel.x += body.acc.x * timestep;
    body.vel.y += body.acc.y * timestep;

    // Calculate position
    body.pos.x += body.vel.x * timestep;
    body.pos.y += body.vel.y * timestep;
```

De två typerna av fysikkroppar `Body.DYNAMIC` och `Body.KINEMATIC` bestämmer hur kroppen beter sig i fysiksimuleringen. En kropp med typen `Body.DYNAMIC` beter sig helt i linje med vad man förväntar sig av en "vanlig" fysikkropp. Typen `Body.KINEMATIC` påverkas inte av gravitationskrafter eller dynamiska kroppar. En kollision mellan en `DYNAMIC` och en `KINEMATIC` hanteras som om `KINEMATIC` har oändligt stor massa. Skulle två `KINEMATIC` kollidera så hanteras denna kollision som fullständigt oelastisk.

Fixed timestep from variable duration

Inte direkt ett designmönster, utan mer en metod för att se till att fysikmotorn inte tar för stora steg i varje iteration. Ifall fysikmotorn tar för stora integrationssteg på grund av lång tid mellan beräkningar kan det hända att två fysikkroppar som skulle ha kolliderat med varandra förflyttas för långt och "missar" varandra. En lösning är att med kod försäkra att fysiksimuleringen körs med tillräckligt litet tidssteg.

```
// physics: the physics engine
// frametime: time for the frame to render
var dt = 10,
var timeleft = frametime;

while(timeleft > 0) {
  var step;

  if(timeleft < dt) {
    step = timeleft;
  } else {
    step = dt;
  }

  physics.integrate(step);

  timeleft -= dt;
}
```

Detta gör att fysiksimuleringen potentiellt kommer att köras mer än en gång per frame. Fördelen är att fysiksimuleringen blir mer precis. Dock tar det naturligtvis längre tid att beräkna fysiksimuleringen mer än en gång per frame, och därför kan denna metod orsaka en ökning i renderingstid, vilket i nästa steg orsakar att fysikmotorn potentiellt behöver köra ännu fler steg, och så vidare. I praktiken verkar det som att denna metod fungerar någorlunda stabilt.

Vill man komma förbi detta potentiella problem så finns mer avancerade metoder, se artikeln *Fix Your Timestep!* (http://gafferongames.com/game-physics/fix-your-timestep/).

AABB-kollisioner

Axis-Aligned Bounding Box (AABB) är ett sätt att representera en area (eller volym i tredimensionella applikationer). Ett enkelt sätt att föreställa sig hur ett objekt representeras av en AABB är en rektangel på rutat papper. Om man ritar en figur på det rutade pappret så kan man också rita en rektangel som omsluter figuren med hjälp av rutnätet på pappret. Om man ritar fler figurer och respektive omslutande rektanglar kommer de omslutande rektanglarnas över- och undersidor samt vänster- och högersidor vara respektivt parallella med varandra. Om man använder denna representation av en kropp i fysikmotorn så är det enkelt att beräkna ifall en kropp överlappar (kolliderar) med en annan kropp.

```
dh1 = bodyA.right - bodyB.left;
dh2 = bodyB.right - bodyA.left;
dv1 = bodyA.top - bodyB.bottom;
dv2 = bodyB.top - bodyA.bottom;
```

```
if(dh1 <= 0 || dh2 <= 0 || dv1 <= 0 || dv2 <= 0) {
  // no overlap
} else {
  // there is overlap
}
```

Algoritmen baseras på fyra enkla test. Om ett utav de fyra testen är sant så överlappar *inte* de två kropparna som testet utfördes på; om den första kroppens högersida befinner sig till vänster om den andra kroppens vänstersida så är det garanterat att de båda kropparnas AABB inte överlappar. De andra sidorna av kropparnas AABB jämförs på liknande sätt.

Bitmask Layers

Ibland vill man ha fysikkroppar som ignorerar vissa andra fysikkroppar. Bitmasking kan användas för effektivt skilja fysikkroppar åt i ett eller flera "lager". I spelexemplet *Jump Dude* finns två lager; ett förgrunds- och ett backgrundslager. Förgrundlagret innehåller spelaren och allt som spelaren kan kollidera med, dvs mark, power-ups, fiender och så vidare. Bakgrundslagret innehållet det mesta som förgrundslagret innehåller förutom spelaren och fiender. Notera att en fysikkropp kan tillhöra mer än ett lager.

```
if(bodyA.layer & bodyB.layer) {
  // The bodies share at least one layer
} else {
  // The bodies do not share any layer
}
```

Om `bodyA.layer == 01` och `bodyA.layer == 10` så är de på olika lager. Operatorn `&` fungerar så att `01 & 10 == false`, `01 & 01 == true`, `01 & 11 == true`, `10 & 10 == true` och så vidare. Detta exempel visar två lager, men kan utökas till fler genom att använda ett värde med tre tecken i binär representation.

Denna konfiguration gör att man kan lägga in fysikkroppar som studsar omkring men som inte påverkar spelaren eller fiender. Ett exempel är om spelaren slår sönder en låda och fragment av lådan studsar omkring. Vi vill inte att spelarens figur ska påverkas av dessa små lådfragment, men vi vill att de ska kollidera med resten av världen. Fragmenten finns alltså endast på bakgrundslagret. När spelaren och lådfragmenten överlappar ser fysikmotorn att spelaren inte hör till det lager som fragmentet tillhör, och därför ignoreras den potentiella kollisionen.

Callback

Det är aningen meningslöst att ha en fysikmotor om spelmotorn inte har något sätt att få reda på när något intressant har hänt. Callbacks används i fysikmotorn för att spelmotorn ska kunna reagera på t. ex. en kollision; callbackfunktionen på båda kropparna som har kolliderat.

Spelexempel

En spelmotors styrkor och svagheter visar sig först när man börjar bygga spel med den. En viktig distinktion som måste göras är att spelmotorns kod och spelets kod inte är densamma. Även om olika spel har användning för liknande funktionalitet, t. ex. rendering, hantering av mus och tangentbord, fysik, ljud och så vidare, så är spellogiken i slutändan det som skapar spelet och ger det dess personlighet. Spellogik kan alltså inte vara en del av spelmotorkoden.

De två spelexempel jag byggt importerar spelmotorn och är konstruerade med hjälp av de funktioner som finns tillgängliga i den utsträckning som är möjligt. Innan vi tittar närmare på spelexemplen så ger jag först en översikt till hur spelmotorn är tänkt att användas.

Plix: en spelmotor

De viktigaste komponenterna för att bygga ett spel med spelmotorn *Plix* är klasserna `PlixApp`, `Scene` och `Entity`. Vi börjar med `PlixApp`.

Grunderna

När man instansierar ett `PlixApp`-objekt så kommer det att söka i DOM:en efter elementet `<canvas id="game"></canvas>`. I detta `canvas`-element kommer all rendering att ske. Utöver det så initieras spelmotorns tidshantering, mus- och tangentbordshantering startas, en scenstack allokeras och renderingsmotorn startas (och får en referens till det tidigare nämnda `canvas`-elementet).

```
var game = new PlixApp();
```

Spelmotorn är nu instansierad, men det är inte startad än. Eftersom scenstacken är tom väntar `PlixApp`-instansen i variabeln `game` på att medlemsfunktionen `PlixApp.runWithScene(Scene s)` ska köras. För att göra det behöver vi först skapa ett `Scene`-objekt.

```
var game = new PlixApp();

var myScene = new Scene();
```

Mer än så behövs inte för att skapa scenen. Och för att starta spelmotorn görs följande.

```
var game = new PlixApp();

var myScene = new Scene();

game.runWithScene(myScene);
```

När denna kod körs så har spelmotorn startat. Med anropet `game.runWithScene(myScene)` så har `myScene` lagts på toppen av scenstacken i `game`, och den interna uppdateringsloopen är igång. Varje uppdatering säger `PlixApp` till `myScene` att uppdatera sig själv. Men om man kör denna kod som den är just nu så kommer inget särskilt att hända. Inget ritas ut i `<canvas>`-elementet eftersom `myScene` ännu inte innehåller något som kan ritas (eller uppdateras).

Det som saknas är ett eller flera `Entity`-objekt i `myScene`. En `Entity` är det som rör sig, reagerar och lever i spelvärlden.

```
var game = new PlixApp();

var myScene = new Scene();

var myEntity = new Entity();
myScene.attachEntity(myEntity);

game.runWithScene(myScene);
```

Det vi gjort nu är skapat ett `Entity`-objekt i `myEntity` och lagt till det i `myScene` med `myScene.attachEntity(myEntity)`. Det betyder att nu när `game` ber `myScene` att uppdatera så kommer den att be `myEntity` att uppdatera sig också. Skapar vi och lägger till fler `Entity`-objekt i `myScene` så gäller samma sak för dem. Händelseförloppet vid varje uppdatering är alltså; `PlixApp` ber den översta scenen i scenstacken att uppdateras och i sin tur ber scenen alla entitys den känner till att uppdateras.

Vi har upp tills nu pratat mycket om uppdateringar, men inte något om vad som händer i en uppdatering?

I varje uppdatering skickar `PlixApp` den mängd tid som passerat sedan förra uppdateringen till den översta scenen. Den tidsmängden används för att uppdatera fysikmotorn (om den finns). Sedan körs för varje entity dess script (om det finns), och slutligen överförs entityns fysikkroppsposition till entityns transform (detta gäller om entityn har en fysikkropp) för att entityn ska ritas ut på rätt plats i `<canvas>`-elementet.

`myEntity` har ingen fysikkomponent än, men innan vi introducerar den så ska vi titta närmre på hur vi uppdaterar en entity via dess `script`-attribut.

```
var game = new PlixApp();

var myScene = new Scene();

var myEntity = new Entity();
myScene.attachEntity(myEntity);

var radius = 10;
myEntity.script = function(ent) {
  ent.transform.position.x = game.width/2 + radius * Math.cos(ent.scene.app.timeElapsed
/ 1000);
  ent.transform.position.y = game.height/2 + radius * Math.sin(ent.scene.app.timeElapsed
/ 1000);
}

game.runWithScene(myScene);
```

I en entitys `Entity.transform.position` finns dess `x`- och `y`-koordinatpositioner. Genom att tilldela `Entity.script` en funktion med argumentet `ent` så kommer entityn åt sig själv i funktionskroppen. I kodexemplet syns också att vi genom `ent.scene.app.timeElapsed` kommer åt `PlixApp.timeElapsed`, vilket är den mängd tid som fortlöpt sedan spelmotorn startades. Alla `Entity`-objekt har tillgång till det `Scene`-objekt det tillhör genom sitt `Entity.scene`-attribut. Detsamma gäller på liknande vis för `Scene`-objekt och `PlixApp` genom `Scene.app`-attributet.

Ovanstående kod kommer att förflytta `myEntity` i en cirkelbana med radien `10` och mittpunkt i centrum av spelmotorns `<canvas>`-element.

Gör mer med komponenter

För att göra mer än att flytta omkring entitys lägger man till `Component`-objekt i entityn. Dessa innehåller information som spelmotorn eller spelet kan använda i diverse syften, t. ex. fysiksimulering, rendering, hålla reda på antal liv och så vidare. I förra sektionen nämndes fysikkomponenten som vi ska titta lite närmare på.

```
var ent = new Entity();

var physicsOptions = {
  entity: ent,
  width: 2,
  height: 8,
  type: Body.DYNAMIC,  // or Body.KINEMATIC
  mass: 2
};

var pc = new PhysicsComponent(physicsOptions);
ent.addComponent(pc);
```

När `PhysicsComponent` skapas så gör den först något som alla `Component`-objekt gör; den lägger till `entity`-attributet som skickas med i `options`-objektet (i detta exempel `physicsOptions`). Detta för att ha tillgång till entityn i dess konstruktor. `PhysicsComponent` kontrollerar sedan om det finns en fysikvärld (`World`) och skapar den om den inte finns, följt av att skapa ett `Body`-objekt som registreras hos `World`. Notera att `Body` och `World` inte är en del av spelmotorn Plix utan är en del av fysikmotorn *physix* som skrevs i samband med spelmotorn. Mer om det snart.

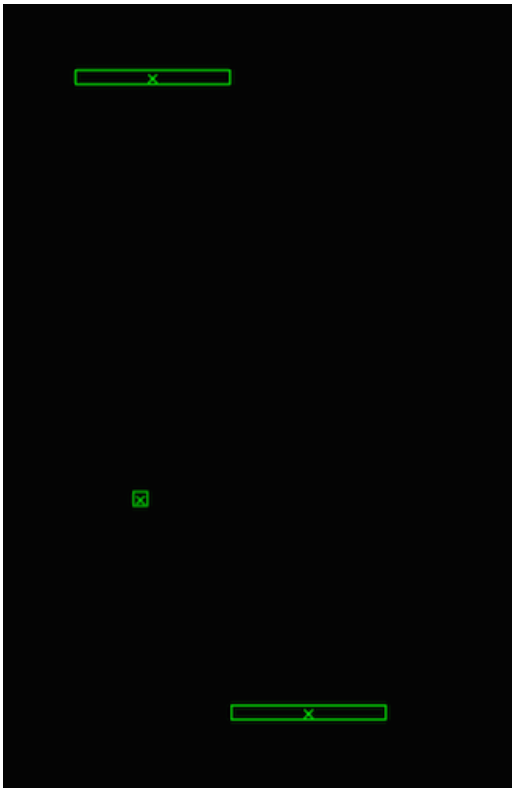
Slutligen lägger vi till fysikkomponenten `pc` till entityn `ent` med `ent.addComponent(pc)`. Nu är allt sammankopplat, och vi kan komma åt fysikkomponenten genom `ent.components.physics`. Precis vad `XXX` byts ut mot i `ent.components.XXX` beror på vad `Component`-klassen har angett i dess `ComponentXXX.type`-attribut. `PhysicsComponent` har angett `'physics'` och statemachine-komponenten `FSMComponent` har `'fsm'`.

Nu när `ent` har en `PhysicsComponent` så kommer spelmotorn att känna av det och göra sånt som behövs för en entity med fysikkomponent, t. ex. synkroniseras fysikvärldens position med entitetens transformposition. På samma sätt kommer spelmotorns renderingsmotor att använda `GraphicsComponent` (om den existerar) som instruktion för att rita något som är knutet till entityn. Det är i sammansättningen av dessa olika komponenter som en entity blir en fiende, spelare, vägg, power-up, etc.

Jag nämnde tidigare att klasserna `World` och `Body` som används i fysikkomponenten inte tillhör spelmotorn. Det stämmer, och vi skulle kunna skriva en `Box2DPhysicsComponent` som istället använder fysikmotorn *Box2D*. Det är tänkt att vi ska kunna skapa en mängd olika typer av spel med endast ett par få byggstenar; `PlixApp`, `Scene` och `Entity` som kryddas med `Component`-klasser för att bli unika spelelement.

Spel 1: Pong

Det klassiska spelet pong har återskapats säkert tusentals gånger. Denna variant är mycket simpel. När spelet har laddats in visas en startskärm och när vänster musknapp klickas hoppar spelet in i matchläge.



En kvadrat studsar på väggar och spelarna försöker hindra kvadraten från att åka i mål, dvs träffa väggen bakom respektive spelares rektangel. När rektangeln träffar mål är matchen slut och spelet återvänder till startskärmen.

Initialisering

Själva huvudprogrammet `main.js` är så pass kort att vi kan titta på det i dess helhet. `PlixApp` samt den kod som används för att "bygga" spelet, `PongFactory`, importeras. `PongFactory` innehåller en mängd funktioner med namn som `PongFactory.createXXX(...)`. Vi återkommer till dessa funktioner snart.

```
define([
    'plix/PlixApp',
    'js/PongFactory'
], function(
    PlixApp,
    PongFactory
) {
    'use strict';

    // Create an app. It's idle until told to run.
    var app = new PlixApp();

    // Create main menu scene
    var menu = PongFactory.createMenu(app);

    // Run the app with main menu scene
    app.runWithScene(menu);
});
```

Vi instansierar spelmotorn med `var app = new PlixApp()`. Spelet ska börja med en menyskärm, och den skapas med `var menu = PongFactory.createMenu(app)`. Den här funktionen är en factory-funktion som returnerar ett `Scene`-objekt innehållande spelets meny. Under huven skapas ett antal `Entity`-objekt som placeras på skärmen för att visa texten "CLICK 2 START".

```
// figLayout: 2D array with "grid-text"
...
var scene = new Scene('menu');

var ent;
for(var r = 0; r < figLayout.length; r++) {
  for(var c = 0; c < figLayout[r].length; c++) {
    if(figLayout[r][c]) {

      ent = new Entity();
      scene.attachEntity(ent);

      ent.transform.position.x = 10*c - 85 + (app.width / 2);
      ent.transform.position.y = 10*r - 100 + (app.height / 2);

    }
  }
}
...
```

Notera att variabeln `ent` innehåller en referens till den senast skapade entityn i loopen. Efter loopen definieras `script`-attributet på den sista entityn i loopen.

```
...
// Last entity waits for mouse input
ent.script = function(ent) {
```

```
    if(app.input.mouse.leftButton) {
      var level = PongFactory.createLevel(app);
      app.pushScene(level);
    }
  };
  ...
```

I detta fall undersöker funktionen om musens vänsterknapp befinner sig i aktivt (nedtryckt) läge och skapar i så fall spelets nästa scen med `var level = PongFactory.createLevel(app)` samt lägger scenen på toppen av scenstacken med `app.pushScene(level)`.

Nu när `var menu = PongFactory.createMenu(app)` har returnerat spelets menyscen så startas spelmotorn med `app.runWithScene(menu)` och vi får följande resultat i webbläsaren.

Matchläge

I förra sektionen nämndes att spelet övergår till matchläge när vänster musknapp klickas. Detta sker genom att matchlägets scen skapas med `PongFactory.createLevel(PlixApp app)` och sedan lades överst på scenstacken, `PlixApp.pushScene(Scene s)`.

Vad händer egentligen när dessa funktioner kallas?

`PongFactory.createLevel(PlixApp app)` skapar en ny scen, och i scenen skapas sju objekt; fyra väggar varav två är mål, två spelarkontrollerade rektanglar och en kvadrat (eller "boll"). Alla dessa objekt är av typen `Entity`, och de "tillverkas" på liknande vis med hjälp av de byggstenar som finns i spelmotorn. För att förenkla skapandet har varje spelelement en factory-funktion som ser ut som `PongFactory.createXXX(Scene s, Object options)`, där `XXX` är spelelementet.

Det simplaste av alla element är den vanliga väggen. `PongFactory.createWall(...)` skapar en `Entity` och adderar sedan en `PhysicsComponent` med konfiguration i argumentet `options`.

```
PongFactory.createWall = function(scene, options) {

  var wall = new Entity();
  scene.attachEntity(wall);

  wall.transform.position.x = options.x;
  wall.transform.position.y = options.y;

  var pc = new PhysicsComponent({
    tag: options.tag || 'wall',
    entity: wall,
    type: Body.KINEMATIC,
    width: options.width,
    height: options.height
  });
  wall.addComponent(pc);

  return wall;
};
```

Detta är grundstrukturen för att skapa nästan allt spelmotorn ska hålla reda på. Det `Scene`-objekt som är längst upp i scenstacken är aktivt och kommer att hålla reda på och uppdatera alla `Entity`-objekt i en intern `Entity`-lista när `PlixApp` säger till (kom ihåg att det bara är den översta scenen i scenstacken som blir ombedd av `PlixApp` att uppdateras; med andra ord ligger menyskärmens scen under matchlägets och "väntar" på att bli aktiv igen). Ett `Entity`-objekt skapas med `var wall = new Entity()` och registreras hos scenen med `scene.attachEntity(Entity e)`.

Efter att ha registrerats hos scenen konfigureras entityn och vi skapar de komponenter som behövs. I det här fallet ska en vägg skapas, och det behövs inte mer än en `PhysicsComponent` som läggs till i entityn med `wall.addComponent(Component c)`. Fysikkroppstypen är en `Body.KINEMATIC` eftersom väggen inte ska rubbas ur plats.

Funktionen `PongFactory.createWall(..)` anropas från `PongFactory.createLevel(...)` för att skapa både vanliga väggar och de som representerar mål. Det som skiljer den vanliga väggen från målväggen är att den senare har värdet `'goal1'` eller `'goal2'` i `options.tag`-argumentet. Det finns inget särskilt med dessa värden utan vi kommer senare använda dem för att undersöka om bollen har träffat ett mål eller en vägg.

```
// Bottom goal
PongFactory.createWall(scene, {
    x: app.width/2,
    y: app.height + 10,
    width: app.width,
    height: 20,
    tag: 'goal2'  // goal-tag
});

// Left wall
PongFactory.createWall(scene, {
    x: -9,
    y: app.height/2,
    width: 20,
    height: app.height
    //tag: 'wall'  not needed since it's the default
});
```

Bollen skapas i factory-funktionen `PongFactory.createBall(Scene s, Object options)` nästan exakt likadant som väggarna, men med fysikkroppstypen `Body.DYNAMIC`. Detta gör att den studsar och kolliderar som en "vanlig"

fysikalisk kropp. Vi tilldelar också en funktion som kommer att anropas när fysikmotorn registrerar en kollision som involverar bollens fysikkropp. Detta med `PhysicsComponent.on(EventName n, Function f(Body otherBody, Vec2 collisionVector))`.

```
// from: PongFactory.createBall(scene, options)
var pc = new PhysicsComponent({
  tag: 'ball',
  entity: ball,
  type: Body.DYNAMIC,
  width: options.width,
  height: options.height
});
pc.on('collision', function(otherBody) {
  if(otherBody.tag === 'goal1' || otherBody.tag === 'goal2') {
    console.log('Goal collision!', otherBody.tag);
    ball.scene.app.popScene();
  }
});
```

När en kollision skett med en fysikkropp som har ett `tag`-attribut som antingen är `'goal1'` eller `'goal2'` så plockar vi bort den översta scenen ur scenstacken med `ball.scene.app.popScene()`.

Den sista typen av spelelement, de spelarkontrollerade rektanglarna, är lite mer intressanta eftersom de behöver reagera på tangentbordstryckningar. Factory-funktionen `PongFactory.createPaddle(Scene s, Object options)` börjar på samma sätt som för de andra elementen men lägger till två ytterligare komponenter; `KeyboardComponent` och `FSMComponent`.

```
// from: PongFactory.createPaddle(scene, options)
// Add input component
var ic = new KeyboardInputComponent({
    entity: paddleEntity
});
paddleEntity.addComponent(ic);
```

`KeyboardComponent` behöver bara instansieras och sedan läggas till hos `paddleEntity`. Detta ger entityn tillgång till en kopia av tangentbordets tillstånd genom objektet `Entity.components.input.keys`. I koden som följer används tillgången till tangentbordet. En `FSMComponent` skapas och sätter upp det standard-`State` som spelarrektangeln har.

```
// from: PongFactory.createPaddle(scene, options)
// Create FSM component
var fsm = new FSMComponent();
paddle.addComponent(fsm);

// Configure FSM
fsm.createState('default')
  .onEnter(function(ent) {
    ent.script = function() {
      var xVel = 0;
      xVel += ent.components.input.keys[options.keys.left] ? -0.5 : 0;
      xVel += ent.components.input.keys[options.keys.right] ? 0.5 : 0;
      ent.components.physics.body.vel.x = xVel;
    };
  });
fsm.enterState('default');
```

Notera att `options.keys.left` och `options.keys.right` är argument till `PongFactory.createPaddle(...)`. De

två spelarnas rektanglar för Pong-spelet skapas med olika spelkontrollstangenter.

```
// from: PongFactory.createLevel(app)
// Create paddle 1
PongFactory.createPaddle(scene, {
  x: 100,
  y: app.height - 50,
  width: 100,
  height: 10,
  keys: { left: 'A', right: 'S' }
});

// Create paddle 2
PongFactory.createPaddle(scene, {
  x: 200,
  y: 50,
  width: 100,
  height: 10,
  keys: { left: 'K', right: 'L' }
});
```

Det sista som behöver göras nu när alla matchlägesobjekt har skapats är att sätta bollen i rörelse. Detta görs genom att applicera en kraft i fysikmotorn på bollens fysikkropp.

```
// from: PongFactory.createLevel(app)
// Give the ball a little push
var force = {
  x: (Math.random() - 0.5) * 0.1,
  y: Math.sign(Math.random() - 0.5) * Math.max(Math.random() * 0.05, 0.01)
};
ball.components.physics.body.applyForce(new Vec2(force.x, force.y));
```

Nu uppdateras alla `Entity`-objekt i scenen och matchläget pågår fram tills bollen har kolliderat med en utav målväggarna.

Resultat

Pong-spelet visade på en begränsning som finns när man använder en *FSM*. Användningen av `FSMComponent` kan skippas helt och hållet och ersättas med ett `Entity.script`. Anledningen är att vi endast har ett tillstånd som en spelarrektangel befinner sig i. Från början var så inte fallet utan kontrollerna styrdes genom att förflyttas mellan tre tillstånd; höger- och vänsterförflyttning samt stillastående. Eftersom man bara kan befinna sig i ett state i taget blir det problematiskt när båda vänster och höger kontrolltangent är nedtryckta samtidigt.

Tekniskt sett så tillhör inte *fsm*-biblioteket spelmotorn trots att det utvecklades i samband med den. Den var ursprungligen tilltänkt att hantera mycket av spellogiken, men som det visade sig var den inte särskilt passande till Pong-kontroller.

I övrigt fungerar spelet bra och det krävs inte allt för mycket kod för att skapa det.

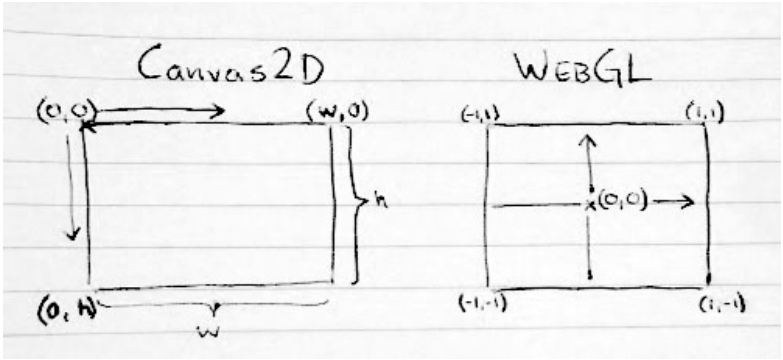
2D-rendering med WebGL och Canvas2D

I detta skede fanns både en Canvas2D- och WebGL-renderare som kunde bytas ut med varandra med knappt märkbar skillnad i den renderade bilden. För den som är intresserad av att se hur Canvas2D och WebGL skiljer sig finns koden för CanvasRenderer och WebGLRenderer nedan. För ett såhär simpelt spel är den klurigaste uppgiften att översätta mellan koordinatsystemen i spelet, Canvas2D och WebGL. Spelet och Canvas2D delar koordinatsystem; en pixel per enhet, x-

axeln pekar höger, y-axeln pekar nedåt och koordinaten `(0,0)` är i övre vänstra hörnet. I WebGL är koordinaten `(0,0)` i mitten, y-axeln pekar uppåt, och både x- och y-axel har värden `[-1,1]` där `1` är höger/topp och `-1` är vänster/botten av `<canvas>`-elementet.

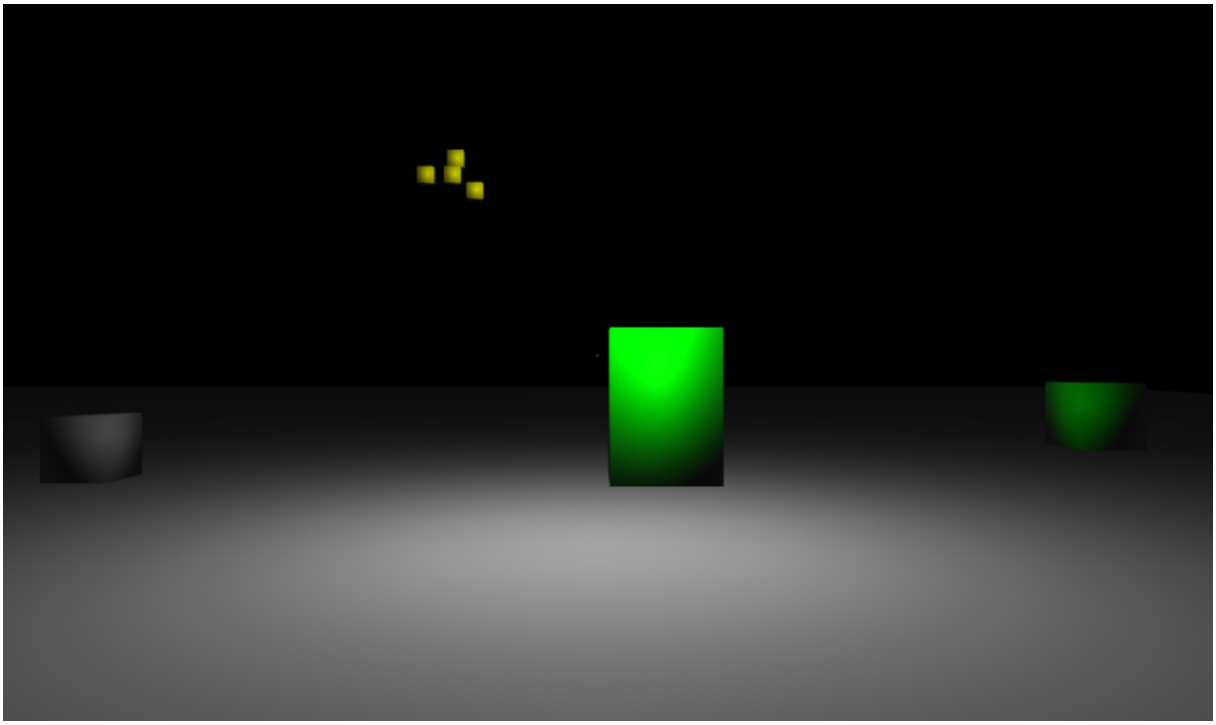
CanvasRenderer:
https://github.com/marcusstenbeck/plix/blob/e9f58401f7ad317a388ed14425f9ab21f256e35d/src/plix/CanvasRenderer.js

WebGLRenderer:
https://github.com/marcusstenbeck/plix/blob/e9f58401f7ad317a388ed14425f9ab21f256e35d/src/plix/WebGLRenderer.js



Spel 2: Jump Dude

Ett mer involverat spelexempel utvecklades också för att se hur spelmotorn klarade sig i detta mer avancerade exempel. I och med att vi i förra sektionen pratade mycket om hur vi med koden skapade spelet så tänker jag i denna sektion röra mig mindre steg-för-steg och istället diskutera ett antal utmaningar jag stötte på längs vägen. Dessa är alltså saker som uppstått under utvecklingen av detta lite mer involverade spelexempel.



Jag utvecklade motorn i samband med att göra Pong, men med Jump Dude så var utmaningen att bygga ett spel med de verktyg jag hade. Förutom att fixa buggar så var det bara 3D-grafik som tillkom i motorn. Fysikmotorn *physix* fick också några buggfixar och uppgraderingar (layers, sensorkroppar, fixed timestep integrering, callback-manager).

Utmaning: När något händer med A, gör också något med B

Att från en entity få något att hända med en annan entity var den utmaning jag stötte på mer än andra problem. Två exempel ur spelet är (1) när spelaren dör animeras kameran runt spelarens position och (2) när spelarens figur landar så skakar kameran till.

```
// Player just entered the death state

ent.script = function(ent) {

  /* ... code stopping controls and slowing down movement ... */

  camEnt.transform.position.x = /* Math.cos(time) animate camera */;
  camEnt.transform.position.y = /* Math.sin(time) animate camera */

  /* ... more code for death animation, etc ... */

  if(ent.scene.app.timeElapsed > /* timeToRestart */) {
      scene.app.playerDied();  // reset level
  }
};
```

Problemet var att få tag på den entity som jag ville få tag på. Spelar-entityn har inte naturligt tillgång till `camEnt`. Men eftersom alla entitys har tillgång till den scen de tillhör så går det att titta i scenens lista över entitys. Dock måste man undersöka en entity i taget. I värsta fall går man igenom hela listan av entitys för att hitta den man letar efter. Resultatet är följande typ av kod varje gång vi vill hitta en specifik entity.

```
var ent;

for(var i = 0; i < scene.entities.length; i++) {
  if(scene.entities[i] == /* some condition */) {
    // found it!
    ent = scene.entities[i];
    break;
  }
}

// do something with `ent`
```

Om man tittar på ovanstående kod kan man tänka sig att den borde vara i en funktion som återanvänds, men riktigt så enkelt är det inte. Det skapas inte något unikt ID för entiteter, så villkoren som behöver uppfyllas beror lite på användningsområdet. Om vi vill få tag i spelets kameraentity så är det enda tillvägagångssättet att undersöka om entityn har en `CameraComponent`.

```
var camEnt;
for(var i = 0; i < scene.entities.length; i++) {
  if(typeof scene.entities[i].components.camera !== 'undefined') {
    // found it!
    camEnt = scene.entities[i];
    break;
  }
}
```

Visserligen går det att lösa genom att skapa en funktion som tar en jämförelsefunktion som inparameter, men då accepterar vi grundfelet som orsakar det här problemet. När något händer i entity `A` så vill vi att det ska avfyra något i

entity `B`. Med ovanstående tillvägagångssätt hämtar vi i koden för `A` först entity `B` och sedan gör vi något. Men detta är inte optimalt eftersom det då potentiellt finns en mängd kod som har att göra med `B` på ställen i kodbasen som annars inte har något alls att göra med `B`.

Det vi vill göra är att separera avfyrandet från beteendet; vi vill skapa ett meddelandesystem som t. ex. en eventbuss. Skillnaden blir att när något händer i `A` så kan `A` direkt skicka iväg ett meddelande, t. ex. `'jump'`, i eventbussen. Efter meddelandet är iväg behöver `A` inte längre göra något, och vi har istället kod i `B` som väntar på att meddelandet `'jump'` ska komma. När meddelandet kommer finns kod hos `B` som tar hand om det. Meddelandesystemet skickar med en referens till entiteten som skickade meddelandet om det behövs.

Exempel (2) från ovan, när spelarens figur landar så skakar kameran till, hjälps också av meddelandesystemet. För att hålla reda på när spelaren har landat *och* innan kameran har börjat skaka tilldelades ett attribut `scene._groundedHappened` till scenen. Detta för att båda entitys enkelt kommer åt den scene de båda tillhör genom `Entity.scene`. Den nya variabeln innehåller `true` om spelaren precis har landat och `false` när kameran börjat skaka.

```
// ... in code for entering `grounded` state ...
playerEntity.scene._groundedHappened = true;
```

Senare i koden för kameran.

```
// in cameraEntity.script function
if(ent.scene._groundedHappened == true) {
  ent.scene._groundedHappened = false;
  ent.startTime = ent.scene.app.timeElapsed;
} else {
  // ... Do shake animation ...
}
```

Detta separerar koden, men om vi hade använt ett meddelandesystem så hade vi inte behövt hålla reda på `scene._groundedHappened` på ett ställe som egentligen inte är till för att hålla reda på en individuell entitys tillstånd, men i detta fall var det snabbt och enkelt eftersom båda entitys enkelt kommer åt värdet.

Med meddelandesystemet får vi bättre isolering av kod. Vi har inte längre spelarentitys som kontrollerar kameraentitys, utan istället spelarentitys som “ropar ut” när något händer och kameraentitys som lyssnar och reagerar.

Callback från fysikvärlden

För att vi i spelvärlden ska kunna reagera på kollisioner som händer i fysikvärlden har vi sedan tidigare sett att en callback-funktion anropas.

```
// from PlatformGameFactory.createPlayer(scene, options)

myPhysicsComponent.on(
  'collision',
  function(otherBody, collisionVector) {
    if(Math.abs(collisionVector.x) < Math.abs(collisionVector.y)
        && collisionVector.y > 0) {
      // Landed, so don't bounce!
      this.body.vel.y = 0;

      // Trigger grounded state
      playerEntity.components.fsm._fsm.triggerEvent('ground');
    }

    if(otherBody.tag === 'goal') {
      finishLevel(scene);
```

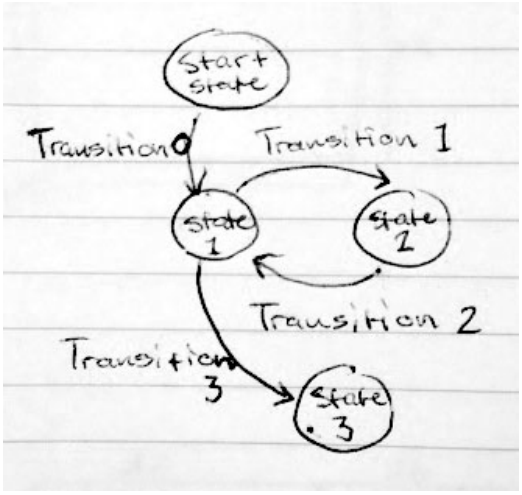
```
    }

    if(otherBody.tag === 'enemy') {
      if(!finished) {
        playerEntity.components.fsm._fsm.triggerEvent('die');
      }
    }
  });
```

Denna lösning fungerade förträffligt bra i och med att båda påverkade fysikkroppar kunde reagera “på sitt eget håll” men fortfarande basera beslut på information om den andra fysikkroppen (`otherBody`).

Utmaning: FSM är bra när man vill vara väldigt noggrann

Från början var idén att använda FSM för att hantera om spelet var pausat och vilken bana man var på, men jag upptäckte snart att det blev väldigt mycket att hålla reda på. När denna känsla infinner sig bör man andas in lugnt och försöka lista ut om det finns ett enklare sätt att lösa saker.

Problemet med att använda en FSM var att varje bana då behövde känna till vilka andra banor som finns. Det är problematiskt när man bestämmer sig mitt i att en viss bana behöver tas bort, eller om man vill lägga till en bana i en serie av andra banor.



En FSM kräver att man specificerar alla transitions mellan olika states. Om alla states ska ha en transition till alla andra states, och man har `S` antal states och `T` antal transitions, så behöver man `S-1` transitions per state. Totalt blir det `T = S*(S-1) = S^2+S` transitions. Det passar alltså bäst till fall där det krävs väldigt få transitions per state eller där antalet states är få.

Istället användes pushdown automata, då det inte kräver samma mängd “bokföring”.

Entity-tillstånd

I slutändan använde jag FSM enbart för spelarens fyra tillstånd: `grounded` , `jumping` , `finished` och `dead` . Det mesta i spelet är rätt enkelt, så det behövs inte mer än ett eller ett par tillstånd.

```
fsm.createState('jumping')
  .onEnter(function(ent) {
      ent.script = function() {
          sideMove(ent);
          scaleJump(ent);
      };
  })
```

```
    .addTransition('ground', 'grounded')
    .addTransition('die', 'dead')
    .addTransition('finish', 'finished');
```

Notera den mängd `.addTransition(String event, String stateName)` som används.

Spelvärldens tillstånd

När spelaren dör eller klarar en bana ändras spelets tillstånd. Från början hade jag tänkt använda FSM till detta, men det blev bökigt att implementera. Istället skapades ett antal `app.funktioner()` som håller reda på spelets tillstånd, uppdaterar spelet, osv.

```
// Game state
app.levelIds = [1, 2, 3];
app.currentLevelIndex = 0;
app.lives = 3;

// Game state functions
app.resetGame(): Resets level state
app.nextLevel(): Goes to next level, or start screen if at last level
app.loadLevel(levelId): Loads level with `levelId`
app.playerDied(): Decrement number of lives and restart level
```

Det är klart att jag skulle kunnat bygga eller refaktorera spelmotorn för att bättre hantera dessa saker, men syftet var att sluta utveckla spelmotorn och istället utveckla ett spel!

“Give a man a game engine and he will deliver a game. Teach a man to make a game engine and he will never deliver anything.”

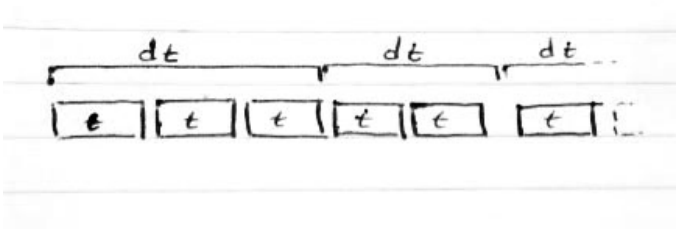
Utmaning: Gör mer (eller mindre) med fysikmotorn

Fysikmotorn har en mängd användbara funktioner, och det vore trevligt att använda dessa styrkor för att förenkla spelutvecklingen. Kollisionsdetektering är en av grundpelarna i en fysiksimulering, men om man kan vara flexibel med hur eller när den används så blir den ännu mer användbar för spel.

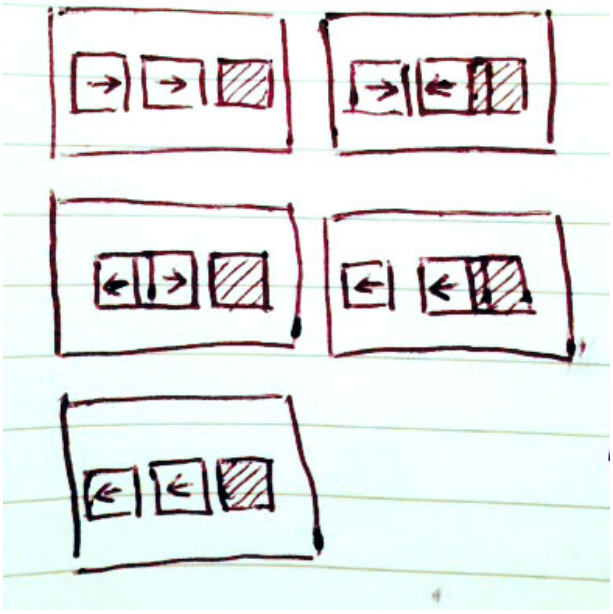
Inspiration har tagits från fysikmotorn Box2D, men jag är säker att många andra fysikmotorer har samma funktionalitet. Vi har redan pratat om callbacks, men det finns mer.

Fysiksimulering med fixt tidssteg

Detta är en optimering, och det gjordes för att fysiksimuleringen skulle bli mer robust. Detta orsakade problem med callbacks eftersom callbacken köas när en kollision upptäcks. Med fixt tidssteg introduceras relativt ofta fall där två eller fler simuleringssteg körs mellan varje spelmotoruppdatering.



Det som kan hända är att en kropp flyttas “fram och tillbaka” under simuleringsstegen och rapporterar en kollision för mer än ett simuleringssteg. Då registreras callback-funktionen mer än en gång och blir också anropad samma antal gånger.



I figuren ovan “kolliderar” den mellersta kvadraten två gånger med den skuggade kvadraten. För att lösa denna bugg implementerades en callback-kö som försäkrar att en callback mellan två kroppar endast kan köas en gång.

```
World.prototype.queueCallback = function(bodyA, bodyB, collisionVector) {

  for(var i = 0; i < this.callbackQueue.aBodies.length; i++) {
    if(this.callbackQueue.aBodies[i] === bodyA
      && this.callbackQueue.bBodies[i] === bodyB) {
      // pair already exists
      return;
    }
  }

  this.callbackQueue.aBodies.push(bodyA);
  this.callbackQueue.bBodies.push(bodyB);
  this.callbackQueue.collisionVectors.push(collisionVector);
};
```

Bitmask-lager

Fysikmotorn innehåller en värld, och i den världen studsar saker omkring. Jag ville använda fysikmotorn till så mycket som möjligt av spelets rörelser, men kunna kontrollera huruvida vissa objekt påverkar varandra eller inte. I spelet skapas ett antal små “skärvor” när spelaren rör vid ett gult block. Dessa skärvor bör studsa på mark och hinder, men inte påverka spelaren.

Ett angreppssätt är att ha två fysikvärldar och synkronisera objekt mellan de båda världarna. Ett problem med detta angreppsätt är att vi behöver ta ställning till vilken av de två fysikvärldarna som är “facit” om de skulle skilja sig åt. Ett annat problem är att man behöver ha dubbla kopior för varje fysikkropp som finns i båda fysikvärldarna, och det kostar i minne. Vi vill gärna undvika dessa problem.

Ett annat angreppsätt är att använda bitmasking för att hålla reda på en eller flera “lager” som en fysikkropp tillhör. Exakt hur dessa bitmasker fungerar finns beskrivet tidigare i rapporten. I kollisionsdetekteringen ignoreras kollisioner mellan kroppar som inte delar åtminstone ett fysiklager med varandra.

```
var pc = new PhysicsComponent({
    tag: options.tag || 'wall',
```

```
      entity: wall,
      type: Body.KINEMATIC,
      width: options.width,
      height: options.height,
      layer: PHYSICS_LAYER_ONE | PHYSICS_LAYER_TWO
    });
  wall.addComponent(pc);
```

I ovanstående kod resulterar `PHYSICS_LAYER_ONE | PHYSICS_LAYER_TWO` i ett värde som betyder att fysikkroppen kommer att kunna kollidera med andra fysikkroppar i både `PHYSICS_LAYER_ONE` och `PHYSICS_LAYER_TWO`, även om dessa kroppar bara finns i en utav de två lagren. Om `PHYSICS_LAYER_ONE == 01` och `PHYSICS_LAYER_TWO == 10` så är `(PHYSICS_LAYER_ONE | PHYSICS_LAYER_TWO) == 11`.

Sensorkroppar

Namnet sensor är lånat från terminologin som fysikmotorn *Box2D* använder. Det syftar till en fysikkropp som existerar endast med syfte att anropa en callback (eller ett event). När en kollision sker kommer sensorn alltså inte ha någon fysikalisk påverkan på fysikkroppen som den kolliderar med. Detta åstadkoms simpelt nog genom att inte registrera en kollision, utan bara en callback.

I *Jump Dude* finns ett spelelement som är en sensor; de gula lådorna som exploderar till skärvor när spelaren vidrör dem.

Utmaning: Dynamisk text

Dynamisk text i WebGL (och GL i allmänhet) är knepigt. I webbläsaren kan man använda den inbyggda textrenderingen om man vill. Jag ville gärna att texten var en del av "världen", så jag behövde lösa det på annat vis. Dessutom hade jag begränsat mig till att använda spelmotorns befintliga kapabilitet för att lösa mina problem.

Likt hur jag skapade startskärmen i Pong-spelet så skapar jag block som formar bokstäver och ord, med skillnaden att i Pong är startskärmens text statisk. Lösningen var ett par funktioner som tillsammans generar en "layout" för varje bokstav i en textsträng, skapar block-entitys i spelmotorn, placerar ut dem på skärmen och möjligtvis också lägger till ett script. Scriptet användes för att animera texten.

```
// Creates entitys based on a string and a cube side length
function createBlockText(String s, Float cube_size)

// Creates a layout array from a string
function createStringLayout(String s)

// Creates a layout array from a character
function createCharacterLayout(Char c)

// Sets up text position animation script based on anchor point
function setupTextBlockBehavior(Entity e, Vec2 anchor_point)
```

Fördelen är att detta är ett enkelt sätt att få till dynamisk text. Den stora nackdelen är att det är onödigt minnesintensivt i och med att det för varje "pixel" i en bokstav skapas ett separat `Entity`-objekt.

Utmaning: Att lägga till "känsla"

När ett spel reagerar på input och saker som händer i spelvärlden så blir hela spelupplevelsen mer inbjudande. I *Jump Dude* visar detta sig genom att t. ex. kameran skakar till när spelarens figur landar efter ett hopp. Figuren deformeras och blir aningen smalare eller tjockare under hoppet för att ge intrycket av att figuren trycker ifrån och sedan kurar ihop sig inför landningen. Detta ger spelarfiguren en upplevd tyngd som inte hade funnits utan dessa extraeffekter.

För att implementera effekterna letar vi upp tillståndsvariabler som kan användas för att skapa effekten. För att deformera spelarens figur använder vi den vertikala hastigheten från dess fysikkropp.

```
function scaleJump = function(Entity ent) {
  var yVel = -ent.components.physics.body.vel.y;
  var factor = 0.5;

  var squash = factor * yVel;
  squash = squash > 1 ? 1 : squash;
  squash = squash < -0.8 ? -0.8 : squash;

  if(!ent.components.graphics.graphic._scale) {
    ent.components.graphics.graphic._scale = [
      ent.components.graphics.graphic.scale[0],
      ent.components.graphics.graphic.scale[1]
    ];
  }

  ent.components.graphics.graphic.scale[0] = ent.components.graphics.graphic._scale[0] *
(1 - squash);
  ent.components.graphics.graphic.scale[1] = ent.components.graphics.graphic._scale[1] *
(1 + squash);
};
```

Tidsbaserade effekter som kamerans skakningar beror på hur lång tid som passerat sedan spelaren landade på marken; kameran börjar skaka mycket och sedan avtar skakningseffekten ganska snabbt. Detta görs genom att skala effektens intensitet med en funktion som avtar med tiden.

```
// function inspired by easings.net
function easeOutExpo(duration, time) {
  var t = time;
  var b = 0;
  var c = 1;
  var d = duration;

  return (t==d) ? b+c : c * (-Math.pow(2, -10 * t/d) + 1) + b;
}

/* ... */

var dt = ent.scene.app.timeElapsed - ent.startTime;
var shakeScale = 10*(1-easeOutExpo(2*landingTime, dt));

camEnt.transform.position.x = camEnt.transform.position.x
  + shakeScale*Math.cos(ent.scene.app.timeElapsed/50);

camEnt.transform.position.y = camEnt.transform.position.y
  + shakeScale*Math.sin(ent.scene.app.timeElapsed/70);
```

3D-rendering med WebGL

För att göra saker lite mer spännande i *Jump Dude* bestämde jag mig för att ta ett steg bort och använda WebGL på sätt som inte är enkla att återskapa i Canvas2D. Spelet är fortfarande tvådimensionellt, men renderingen görs istället i 3D. För att skapa 3D-objekt från 2D-representationen av världen behövdes djup läggas till. Det gjordes på simplast möjliga vis genom att använda objektets bredd också som djup. För själva 3D-renderingen implementerades hemisphere- (se boken Real-Time Graphics) och per-pixel lighting.

För 3D-matematik användes biblioteket *toji/gl-matrix*.

Obs: Det går fortfarande att rendera spelet med Canvas2D-renderaren.

Sammanfattning

Det är väldigt mycket kod i ett spel som inte är en del av spelmotorn. Alla små detaljer som gör spelets personlighet och spellogik tillkommer utöver spelmotorns kod. Att spendera en överdrivet lång tid på att försöka komma på "bra lösningar" är ett säkert kort för att spendera tid på saker man inte vet kommer komma till användning. I detta projekt var ett tydligt exempel *fsm*; den användes väldigt lite och där den användes kunde det lika gärna hanterats av mer primitiva metoder.

Det finns inget som går upp mot att faktiskt skapa ett spel. Om målet är att skapa en spelmotor så kommer man mycket längre på att göra två spel. När man gör det första spelet stöter man på problem som man måste hitta lösningar till, och när man gör det andra spelet kommer man att upptäcka vilka saker som var i stort sett nästan likadana för båda spelen. Efter dessa två spel kan man bygga en simpel spelmotor från kod som är liknande i båda spelen, och sedan skriva om spelen i spelmotorn. Försök sedan skriva ett tredje spel med bara spelmotorn. Det är min rekommendation till andra som går i tankebanor om att bygga en spelmotor.

Med det sagt uppskattar jag insiktera jag fått genom att ge mig på utmaningen. Min strategi för att programmera är nu väldigt annorlunda från då jag påbörjade projektet; jag kom till insikten att det är dumdristigt att hitta på ett problem att lösa innan problemet visat sig i koden.

Det fanns ingen mening att fokusera på hur snabb spelmotorn var under utvecklingstiden. Det är något jag kände till redan innan, och jag var ganska bra på att hålla fingrarna borta från att försöka bättra på hastigheten. Det brukar kallas *premature optimization* när man fokuserar på prestanda innan behovet har uppstått.

Däremot föll jag ner i fällan planera och arkitektera utifrån gissningar om vad som är viktigt, och mycket tid spenderades på att bygga för dessa gissningar. Jag grävde helt enkelt i fel hörn alldeles för länge, och detta misstag kallas *premature architecture*. Det är inte fel att arkitektera ifall man faktiskt vet med säkerhet att problemen kommer att uppstå. Om man är det minsta osäker så är det bättre att skriva upp idén och återbesöka den när problemet verkar ha uppstått. Vet man av erfarenhet att en *game loop* behövs för spelet så är det helt okej att implementera den i förebyggande syfte. Men det är en hal stig och man bör vara mycket försiktig och observant. Helt plötsligt har man jobbat en månad i tron om att man jobbat på spelet, när själva spelet ser det likadant ut som fyra veckor tidigare.

Make it work. Make it right. Make it fast.

I praktiken skapar jag en ofta en funktion som är tänkt att innehålla all kod för den nya saken. Funktionen döps till det jag vill att funktionen ska göra. Jag placerar funktionsanropet på den plats i koden där den verkar passa. Detta samlar allt på en fokuserad plats och jag kan när det fungerar lista ut hur jag kan göra saker bättre. Om det är okej att det "bara fungerar" så lämnar jag det som det är. Fördelen är att det blir väldigt explicit vad dessa rader kod är tänkt att åstadkomma.

Denna approach förebygger "skrivkramp" då vi tillåter oss själva att skriva lite "fulare" kod för att komma igång. Dessutom är det svårare att fastna i att prematurt arkitektera snygga lösningar och riskera att skapa en fantastisk lösning för problem som inte finns.

```
// TODO: This shows up a lot, DRY
var vecSolve = new Vec2(
  (0.5 * (body1.shape.width + body2.shape.width) - Math.abs(vectorAtoB.x) +
stabilityHack) * Math.sign(vectorAtoB.x),
  (0.5 * (body1.shape.height + body2.shape.height) - Math.abs(vectorAtoB.y) +
stabilityHack) * Math.sign(vectorAtoB.y)
);
```

Detta är kod ur *physix*. Dessa rader finns på fler än en plats i koden, och det är en varningsflagga. När det visat sig på något vis att en approach inte fungerar så börjar jag markera med kommentarer i koden som påminnelse att detta behöver ses över. När dessa små påminnelser blivit tillräckligt många så brukar jag se över dem och åtgärda de som orsakar störst trubbel. Ibland får de leva i koden länge.

Det sista steget är att optimera koden så att den kör så snabbt som möjligt. Anledningen till sistaplatsen är att det kan vara ganska klurigt att få vissa delar av koden att köras snabbare. Därför är det ganska vanligt att tillåta icke optimal kod tills den visat sig vara en flaskhals i hela systemet.

Dessa tankar är inte nya utan har funnits länge i Unix-världen: http://c2.com/cgi/wiki?MakeItWorkMakeItRightMakeItFast

Ett ord angående prioritering

En förvånansvärt svår sak att välja är vad man ska jobba på. För min del blev det absolut roligast och mest produktivt att prioritera saker baserat på en enkel fråga “Vad kan jag göra nu för att göra spelet lite häftigare eller bättre för spelaren?”. Kort sagt; prioritera sånt som gör skillnad i slutprodukten!