En sporestrålingsutfordring

Dette er en øvelse i minimalisme. Jeg er vant til å ha full tilgang til Internett, til nedlasting av programvare; utallige artikler og videoer om alle tenkelige emner, rik programvare med tusenvis av hjelpemidler som gjør mye av jobben for meg. Nå må jeg klare meg med DFS, nesten uten Internett-tilgang, helt uten mulighet for installasjon av programvare og med minimalt av hjelpemidler. Jeg skal utdype disse begrensningene og hva jeg gjør for å komme rundt dem.

# Utfordringen

Utfordringen kommer i form av en bok, «The Ray Tracer Challenge» av Jamis Buck, utgitt 2019 på The Pragmatic Programmers. Denne boken tar leseren gjennom alle stegene for å lage en «3D Renderer», en programvare som lager bilder av 3D-modeller, ved beregne hvordan lys reflekteres og brytes når det treffer de ulike 3D objektene som utgjør modellen. Dette skal man lage selv – boken har ingen ferdig kode, kun «pseudokode» som beskriver omtrentlig hva de ulike funksjonene skal gjøre, og så er det opp til leseren å implementere dette i sitt valgte programmeringsspråk. I tillegg er fremgangsmåten TDD (Test-Driven Development), som jeg skal redegjøre for lenger ned i dette dokumentet.

## Miljøet

Gitt begrensningene har jeg kun funnet èn mulig tilnærming til denne utfordringen, og det er med HTML, CSS og JS i nettleser. Ingen webserver, ingen API, ingenting annet enn lokale filer som kjører i Google Chrome.

Jeg har tilgang til nettsiden w3schools.com hvor de har en begrenset kode-editor som jeg vil bruke til å teste noe av koden.

I tillegg til referansene hos w3schools.com har jeg tilgang til stackoverflow.com som gir svar på det meste jeg kan tenkes å lure på.

DFS er et ekstra nedlåst Citrix-miljø. Det er ingen tilgang til å installere egen programvare, så jeg kan ikke bruke noen biblioteker. Det er ingen tilgang til Chrome’s Developer Mode eller konsoll-logg, så jeg får ikke se en eneste feilmelding. Det er heller ingen kommandolinje eller versjonskontroll (git), så jeg må huske å ta manuelle daterte kopier med jevne mellomrom. «Kode-editoren» er notepad, med noe bedre funksjonalitet i [w3schools nettbaserte editor](https://www.w3schools.com/js/tryit.asp?filename=tryjs_editor), men der kan jeg ikke referere eksterne filer, så alt må legges inn i samme vindu.

Filstrukturen som etter hvert har dukket opp er en rot-katalog (/html) som inneholder alle html-filer, og underkataloger for js-filer (/js) og etter hvert for css (/css). I tillegg ligger dette dokumentet under en dokumentasjonskatalog (/docs), der jeg også vil lagre skjermbilder etc. Strukturen blir til mens vi går.

Backup tas ved å kopiere hele /html strukturen til et eget område (/Backup/ÅÅÅÅMMDD/) for hver dag. Dette er på ingen måte sammenlignbart med versjonskontroll, men der det beste jeg kan komme på.

# Test-dreven utvikling

Boken baserer seg som nevnt på TDD, som jeg lenge har hatt lyst til å praktisere. Metoden er som følger:

1. For hver ny funksjon man skal lage, lager man først en test som verifiserer at funksjonen gjør det den skal.
2. Så kjører man testen, som forventet vil feile, siden man ikke enda har implementert funksjonen som testes.
3. Så skriver man funksjonen og kjører testen til den verifiseres korrekt.

Et «litt ikke-trivielt» eksempel: Jeg trenger en funksjon som, gitt to sider av en trekant, representert ved tallene a og b, returnerer lengden på hypotenusen, altså «kvadratroten av a2 + b2».

Først lager jeg en test som kjører denne ikke-eksisterende funksjonen, gir den to kjente tall u og v og sjekker hva funksjonen svarer, mot det forventede resultat som jeg har sjekket på kalkulatoren min ( ) og returnerer true eller false avhengig av resultatet av testen:

**function test\_hypotenuse\_function() {  
 var u = 5  
 var v = 3  
 return (hypotenuse(u, v) == 5.830951895)  
}**

Siden funksjonen ikke finnes, vil testen feile. Så lager jeg funksjonen, men uten å gjøre den ferdig:

**function hypotenuse(a, b) {  
 return 0  
}**

Kjører testen på nytt, den feiler igjen. Jeg implementerer funksjonen ferdig:

function hypotenuse(a, b) {  
 return Math.sqrt(a\*\*2 + b\*\*2)  
}

Og kjører testen igjen. Denne gang skal den vel returnere true? Eller?

Det er her elegansen i denne fremgangsmåten blir synlig. På grunn av forskjell i presisjon mellom JS og min kalkulator vil ikke testen passere. JS gir svaret **5.830951894845301**. At testen nå feiler er dens eneste mening og skal reise et rødt flagg hos meg, hvor jeg må gå grundigere inn i årsaken til at testen feiler. Hvis jeg sammenligner med kvadratroten av 34, så passerer testen som den skal.

Dette kan ved første øyekast virke tungrodd og masse ekstra kode for å teste EN kodelinje, men dette er bare det grunnleggende, og det virker jo unødvendig å sammenligne **1 == 1**… Når jeg senere tester sammensatte funksjoner som feiler uventet, eller jeg gjør endringer som får tester til å feile som tidligere har fungert fint, vil jeg kunne avdekke slike subtile feil som avrunding etc. som ellers kunne gått ubemerket hen i lang tid før den senere skaper problemer helt andre steder i koden. Mikser vi inn et team av utviklere som gjør mer eller mindre tilfeldige endringer i hele kodebasen samtidig, mangedobles verdien av å ha gode grundige tester som sikkerhetsnett.

13.04.2024: Finner [en funksjon som kan vise feilkoder i HTML her](https://stackoverflow.com/questions/14481381/how-to-catch-all-javascript-warnings-and-errors-to-an-output-div#33321914)… ☺

# Prosjektet

Jeg starter i begynnelsen av April 2024 med dette prosjektet. Jeg leser noen kapitler og tester litt miljøet jeg kan programmere i. Det er ikke vits å starte på noe dersom jeg for eksempel ikke kan kjøre JS i eksterne filer, eller ikke kan bruke HTML canvas elementet. Heldigvis ser det ikke ut til at det ikke er restriksjoner her.

Rammeverket for testing blir sakte til mens jeg går. Jeg starter med manipulering av «hardkodede» HTML-elementer, men kommer fort på bedre tanker og lar hver test kjøre som en egen funksjon runTest()som både kjører testen og viser navnet på test-funksjonen og resultatet av testen i rød eller grønn tekst. Foreløpig holder dette.

test\_runner.html laster JS-skriptene js/functions.js og js/tests.js som inneholder hhv funksjonene som skal testes og testene som benyttes.

## Kapittel 1, Tuples, Points and Vectors

Det første kapitlet handler om de mest grunnleggende byggesteinene i dette systemet. En tuple er en datatype med x, y og z koordinater, samt en verdi w som forteller om dette er en vektor (**w=0**) eller et punkt (w=1). I første implementasjon er en vektor bare en tuple med w=0 og et punkt er bare en tuple med w=1. Det er ingen restriksjoner mellom de ulike datatypene. Dette kan det bli nødvendig å endre senere, men inntil videre er distinksjonen kun for egen klarhet.

Vi lager en funksjon equal(a, b) for å sammenligne to flyttall med «slingringsmonn», siden man ellers vil få problemer grunnet avrunding. Dette er et generelt problem i programmering, siden flyttall ikke kan gjengis helt nøyaktig. Denne brukes også i funksjonen equal\_tuples() som sammenligner om to tupler er like.

Videre oppretter jeg en konstant **PRECISION=5** som brukes til å angi presisjonen i avrundingsoperasjoner med .toFixed(PRECISION) der det blir nødvendig i koden, og til å angi konstanten EPSILON = 1.0\*10\*\*(-PRECISION) som er sammenligningsverdien som brukes i **equal()-**funksjonen. Dersom a-b < EPSILON er tallene ansett som like. Når PRECISION = 5 er EPSILON 1\*10-5. Dette må jeg sannsynligvis justere senere, dersom det skaper problemer.

Funksjoner for å legge samme og trekke vektorer fra hverandre er neste. Her kommer w-verdien til sin rett, for hvis man legger sammen et punkt (w=1) og en vektor (w=0) får man et nytt punkt (w=1). Og dersom man legger sammen to punkt får man w=2, som ikke er verken punkt eller vektor. Dersom man legger samme to vektorer får man fortsatt w=0, som er en ny vektor. Helt riktig! ☺

13.04.2024: Jeg jobber med å legge sammen og trekke fra tupler, og så sammenligne resultatene etterpå. Dette må fungere bra, ellers blir det trøbbel down the line. Men det er ikke så enkelt! :O

På et tidspunkt virker det som at tests.js ikke lenger blir importert – funksjonen test\_tuple\_function() er ikke definert. Men i Tryit editor fungerer det (tilsynelatende). Her blir det en del feilsøking før jeg lærer to viktige ting: a) en JS-fil med feil i, blir ikke lastet inn, og b) JS-variabler kan ikke ha bindestrek (–) i navnet.

Jeg fortsetter å implementere tuple-funksjoner og –tester til jeg er gjennom kapittel 1. På slutten av kapitlet er det en liten øvelse for å teste ting, der man skal lage en virtuell «kanon», og skyte prosjektiler. Prosjektilet har en posisjon og en hastighetsvektor. Miljøet har gravitasjon (vektor) og vind (vektor) som skal regnes med.

Her går jeg for første gang på en Out Of Memory krasj, på grunn av en evig loop. Jeg får funksjonen til å regne posisjoner helt fint, og lagrer forsøket i ./projectile.html.

## Kapittel 2, Drawing on a Canvas

Her starter vi med litt farge-teori, og noterer at fargenes r,g,b verdier til forveksling ligner på tuples! ☺ Jeg ser at boka vil at vi skal lage PPM filer, som er et tekstbasert grafikk-format. De anbefaler at man bruker GIMP til å se disse på Windows, og jeg får sjekket at GIMP er installert på maskinen. Yey!

Jeg starter kodingen med å implementere (test av , og) color(r, g, b) datatypen, og funksjoner for å legge sammen og multiplisere farge med skalarverdi og to farger med hverandre.

Deretter skal vi ha en metode som lager et spesifisert canvas-objekt og tester at det blir laget, og deretter skrive en fargeverdi til en spesifisert pixel og lese den ut.

Jeg undersøker hvilke metoder vi har tilgjengelig; kan jeg for eksempel lese ut width og height fra det canvas-elementet jeg nettopp har laget? Finner ikke ut av det, og nøyer meg med å teste at det finnes et element med spesifisert id.

For å skrive til canvas, henter man ut konteksten med canvas.getContext(2d) og skriver til den. (Ja, det skal være 2d, for raytraceren skal bare kalkulere fargeverdier på pixler.)

Kan jeg så lese ut fargeverdien til en gitt pixel?