Andre hendelsestyper

Vi har foreløpig bare sett på klikk-hendelsen (`click`), men det finnes mange flere hendelser (og i tabellen vi så på tidligere, vises bare et fåtall av alle tilgjengelige hendelser). Mange av de tilgjengelige hendelsene er beregnet på avansert bruk, men vi skal se på to grupper med hendelser som vi kan ha glede av, nemlig musebevegelser og tastaturtrykk.

Musebevegelser

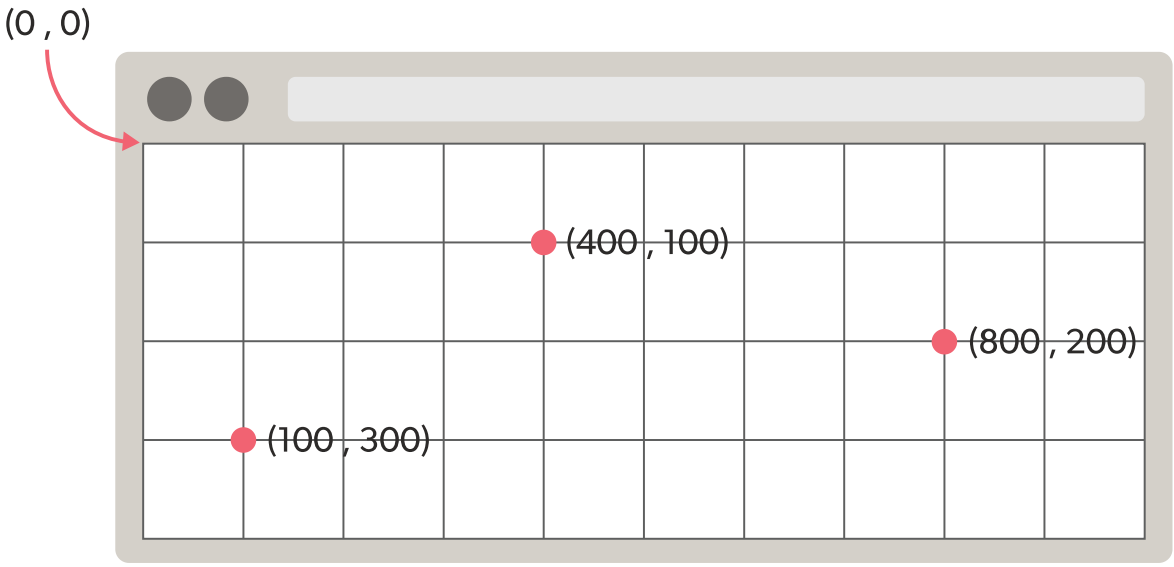
I hendelsesobjektet som vi har sett på tidligere, finner vi også informasjon om musepekerens posisjon idet hendelsen fyres av. Hvis vi kombinerer denne informasjonen med hendelsen `mousemove`, kan vi til enhver tid si hvor musepekeren befinner seg i et element. I koden nedenfor kan du se hvordan vi kan bruke denne hendelsen.

Legg merke til at vi har angitt `height: 100%;` for både `<html>` -elementet og `<body>` -elementet. Hvis vi ikke gjør det, vil `<body>` -elementet strekke seg til det nederste `<h2>` -elementet, mens vi ønsker å registrere musepekeren på hele siden. Vi har valgt å «lytte» etter musepekeren i `<body>` -elementet fordi vi kan få det til å dekke hele nettleseren.

Nettleserkoordinater

I koden bruker vi egenskapene `clientX` og `clientY` som hører til hendelsesobjektet. Disse egenskapene inneholder musepekerens *x*-koordinat og *y*-koordinat i nettleservinduet. Legg merke til at origo (0,0) befinner seg øverst til

venstre i nettleseren. Nettleserens koordinatsystem starter nemlig øverst til venstre, slik at *x*-koordinaten øker når vi beveger musepekeren mot høyre, og *y*-koordinaten øker når vi beveger musepekeren nedover.



Ved å bruke nøyaktig posisjonering av HTML-elementer, med CSS-egenskapen `position` , kan vi bruke det vi har sett på nå, for å lage elementer som følger musepekeren.

Snakk!

Nettleserens origo

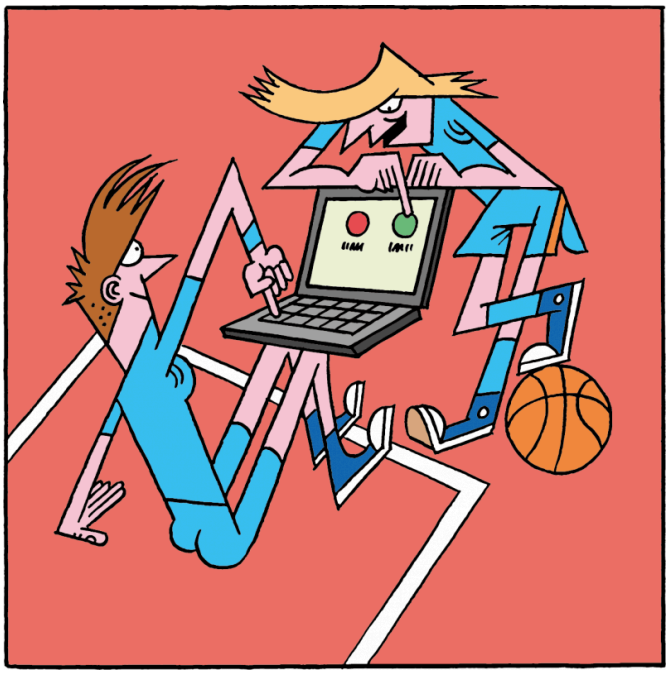
I matematikkgrafer er origo ofte nederst til venstre. Hvorfor er origo øverst til venstre i nettleservinduet?

Tastaturhendelser

Vi kan også bruke hendelser til å registrere når vi trykker på knappene på tastaturet. I koden nedenfor kan du se et eksempel på bruk av `keyDown` -hendelsen, som fyrer av hver gang vi trykker på en knapp.

HTML CSS JS Result EDIT ON

Resources 1× 0.5× 0.25× Rerun



Denne koden oppfører seg ganske likt som `mousemove` -koden. Vi lytter etter tastetrykk i `<body>` -elementet, så vi må først klikke i `<body>` -elementet med musepekeren for at koden skal fungere.

Når en tast trykkes, fyres en `keyDown` -hendelse av, og vi bruker egenskapen `keyCode` for å finne ut *hvilken* tast som ble trykket. Koden vi får, er den samme som brukes i ASCII-tegnsettet for små bokstaver. Du kan bruke

denne koden til å finne ut hvilken `keyCode` de ulike tastene har; på den måten kan du sjekke hvilken tast som ble trykket.

```
if (e.keyCode === 87) {
  console.log("Du trykket på
}
```

Alternativt kan du bruke metoden `fromCharCode()` som hører til `String` -objektet. Den gjør om verdien i `keyCode` til sitt tilhørende ASCII-tegn:

```
let tast = String.fromCharCode(e.keyCode);
console.log(tast);
```

Her blir for eksempel «W» skrevet ut hvis `keyCode` har verdien 87.

Det finnes også muligheter for å lage kombinerte tastetrykk med Ctrl, Alt og Shift. Hvis for eksempel Ctrl-tasten holdes nede mens en annen tast trykkes, vil egenskapen `e.ctrlKey` ha verdien `true` .

Oppgaver

0 Trykk deg gjennom hele alfabetet i programmet ovenfor. Ser du noe mønster? Er det noen av bokstavene som bryter med mønsteret? Forklar.

1 Lag et «spill» som oppgir et tall mellom 65 (ASCII-koden for bokstaven «A») og 90 (ASCII-koden for bokstaven «Z»). Brukeren skal trykke på tasten som hører til koden som

oppgis. Spillet skal til slutt skrive ut antall forsøk som ble brukt for å finne riktig tast.

I dybden

Hendelsesfunksjoner og argumenter

Som vi har sett, bruker vi funksjonsnavn uten parenteser i `addEventListener()` . Hvis det hadde stått parenteser der, ville nettleseren ha kjørt funksjonen umiddelbart, men vi vil at den skal vente til hendelsen inntreffer; derfor utelates parentesene. Det betyr at vi ikke kan sende med noen argumenter om vi skulle ønske det. Hvis vi likevel ønsker å sende med argumenter til en slik funksjon, finnes det en løsning.

```
element.addEventListener("click", function() {
  minFunksjon(argument);
});
```

Her bruker vi det som kalles en *anonym funksjon*, som vi sender til lytteren. En anonym funksjon er en funksjon uten noe navn. Inni den anonyme funksjonen skriver vi funksjonen vi egentlig ønsker å bruke, med tilhørende argumenter. Som oftest kan vi klare oss uten denne framgangsmåten fordi argumentet vi ønsker å sende, er å finne i hendelsesobjektet (for eksempel verdien til et av attributtene til elementet vi klikket på).

Vi kan for eksempel bruke en anonym funksjon i Monty Hall-eksemplet vårt. I stedet for å bruke dørenes id for å finne ut hvilken dør vi trykket på, kan vi heller gjøre det slik:

```
rodDor.addEventListener("click", function() {
  sjekkDor("rød");
});
```

Da kan funksjonen `sjekkDor()` skrives om til å motta dørens farge som argument:

```
function sjekkDor(farge) {
  console.log(farge);
}
```