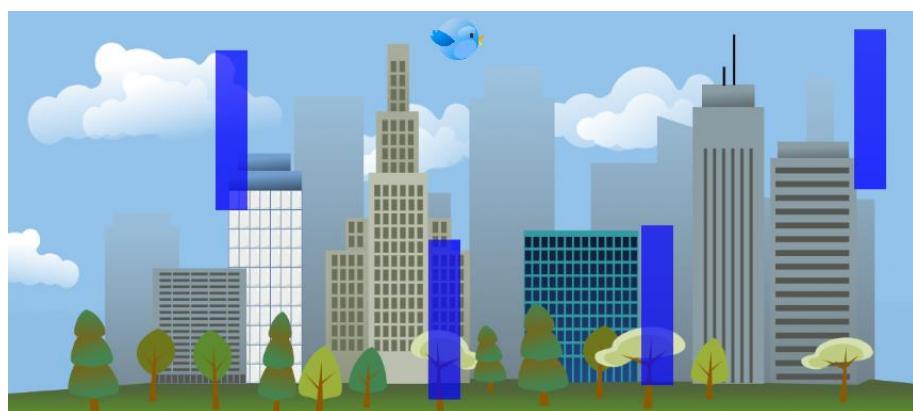
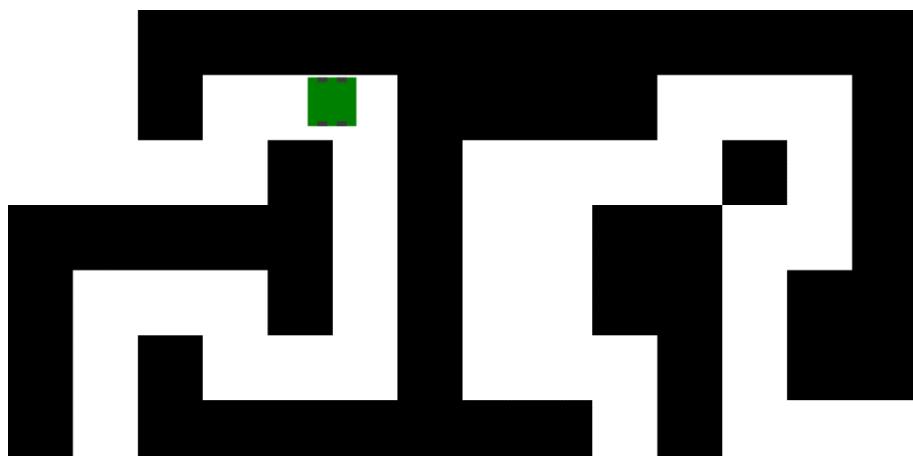


games maken en ervaren



Domein J: Keuzethema Programmeerparadigma's
Domein P: Keuzethema User experience

INHOUD

Inleiding op de module	3
H1 kennismaken met P5	4
1.1 het canvas: ons speelveld	4
1.2 P5-functies: parameters en argumenten	7
1.3 variabelen	9
1.4 draw is een loopfunctie	10
1.5 zelf functies maken	13
1.6 Voorwaarden: if en else	15
1.7 herhalingen met een for-loop	18
1.8 VERDIEPING: recursie	22
H2 objecten	26
2.1 Inleiding (camelCase)	26
2.2 afbeeldingen	26
2.3 lijsten: arrays	29
2.4 animatie: een array van plaatjes	31
2.5 VERDIEPING: spritesheets	34
2.6 zelf objecten maken	36
2.7 een object dat ja of nee antwoordt	41
2.8 een klasse van objecten	44
2.9 een array van objecten	47
H3 game design	51
3.1 Inleiding (Wat is een spel?)	51
3.2 Het object spel & flowcharts	52
3.3 Acties en gebeurtenissen	57
3.4 Vaste patronen: design patterns	62
3.5 Uitdaging en beloning	65
3.6 Tijd	71
3.7 Maak je eigen game	73
H4 gamification	74
4.1 Inleiding	74
4.2 Levels	74

INLEIDING OP DE MODULE

In deze module leer je stap voor stap programmeren in de taal Javascript met P5. P5 of *Processing* is een aanvulling op Javascript. Zo'n aanvulling heet een bibliotheek (*library*). P5 is speciaal gemaakt om het programmeren makkelijker te maken voor mensen die net beginnen met programmeren. Maar vergis je niet: je kunt het zo ingewikkeld maken als je zelf wilt!

In hoofdstuk 1 leer je de basisprincipes van het programmeren en specifieke javascript- en P5-elementen. Als je al eerder geprogrammeerd hebt, kun je sneller door dit hoofdstuk heen. In hoofdstuk 2 zoomen we in op een speciale programmeertechniek die *object-georiënteerd programmeren* (OO of OOP) heet. Het werken met objecten gaat ons helpen bij ons uiteindelijke doel: spellen programmeren in Javascript. Hoofdstuk 3 is helemaal gewijd aan het ontwerpen en ervaren van games.

Voordat je kunt beginnen met programmeren in P5, moet je een omgeving inrichten om in te werken en te testen. Dat kan op vele manieren online of met speciale programma's. Daarom leggen we dat hier niet uit. Jouw leraar vertelt je hoe jullie op school gaan werken en zorgt dat jij de bestanden krijgt om mee te werken. Als je coderegels wilt lezen of aanpassen kijk je in een (tekst-) *editor*. Om het resultaat te bekijken, gebruik je een *browser* (zoals Chrome of Firefox).

Deze module is zo gemaakt dat je leert programmeren door opdrachten te maken. Dit betekent dat niet alles wat je leert ook in de theorie terug te vinden is: je leert het door het maken van opdrachten. Wel zijn er voorbeelden beschikbaar om je op weg te helpen. Heel handig is bovendien de *reference* van P5, waar je informatie vindt bij alle P5-commando's.

Zoals je misschien hebt gemerkt, worden er bij informatica veel Engelse termen gebruikt. We proberen hier zoveel mogelijk het Nederlands te gebruiken, maar het is belangrijk om de Engelse termen te kennen, bijvoorbeeld om op het internet te kunnen zoeken. Onderstaande links zijn handige bronnen van informatie:

Overzicht en uitleg van P5-commando's: <https://p5js.org/reference>

Interactieve site met *keycodes*: <http://keycode.info>

Overzicht van html-kleurennamen: <https://htmlcolorcodes.com/color-names>

Uitgebreide naslag voor ontwikkelaars #1: <https://developer.mozilla.org/nl/docs/Web/JavaScript>

Uitgebreide naslag voor ontwikkelaars #2: <http://devdocs.io/javascript>

Uitgebreide Engelstalige uitleg: <https://javascript.info>

H1 KENNISMAKEN MET P5

1.1 het canvas: ons speelveld

Als je voorbeeld 1 in je browser opent, zie je figuur 1.1 als resultaat. Deze tekening is geprogrammeerd met Javascript. De code bestaat uit twee belangrijke delen: de **setup** en **draw**.

In de **setup** van een programma (figuur 1.2) creëer je de beginsituatie van het programma. We gaan hier niet alle regels één voor één uitleggen. Belangrijk voor nu zijn:

- `createCanvas(1000, 500);`
Maak een speelveld / canvas van 1000 pixels breed en 500 pixels hoog
- `background('orange');`
Geef het canvas een oranje achtergrondkleur

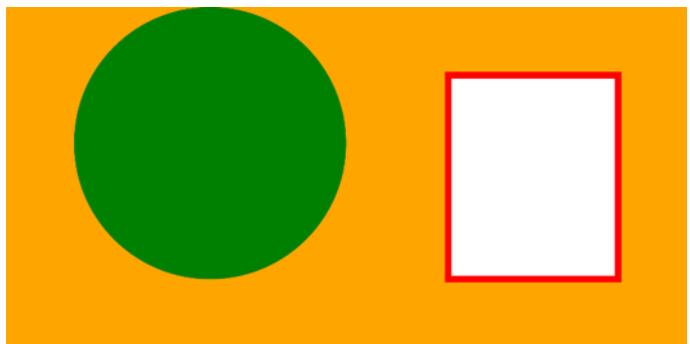
Het tweede deel van de code staat in figuur 1.3. Dit is ons hoofdprogramma: de **draw**. In dit voorbeeld staan hier commando's om de cirkel en de rechthoek te tekenen in het canvas.

Het **canvas** kun je zien als een veld van pixels. Je verwijst naar een bepaald punt met een x-waarde en een y-waarde. Belangrijk om te weten: het punt [0,0] ligt linksboven en een grotere y-waarde betekent dat je naar beneden gaat (en niet omhoog, zoals in een assenstelsel bij wiskunde).

In figuur 1.4 zie je het resultaat van figuur 1.1 met een raster van hokjes van 100×100 pixels. Die helpen je bij het begrijpen van de code in figuur 1.3:

- `ellipse(300, 200, 400);`
Teken een ellips (een cirkel is een ellips die net zo breed is als hij hoog is) met het **middelpunt** in [300,200] en een breedte en hoogte van 400 pixels.
- `rect(650, 100, 250, 300);`
Teken een rechthoek met als **positie linksboven** [650,100] en een breedte van 250 en een hoogte van 300.
- `fill('white');`
Kies een vulkleur. Merk op dat je eerst een vulkleur kiest en dan pas een vorm tekent.
- `stroke('red');`
`strokeWeight(10);`
Teken de figuur met een rode rand van 10 pixels.
- `noStroke();`
Teken de figuur zonder rand.

We gaan nu ons eerste programma schrijven.



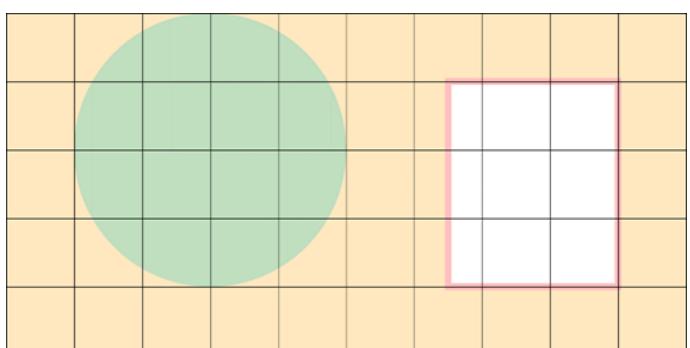
FIGUUR 1.1

```
function setup() {  
    canvas = createCanvas(1000, 500);  
    background('orange');  
    canvas.parent('processing');  
    noLoop();  
}
```

FIGUUR 1.2

```
function draw() {  
    // groene cirkel zonder rand  
    noStroke();  
    fill('green');  
    ellipse(300, 200, 400);  
    // witte rechthoek met rode rand  
    stroke('red');  
    fill('white');  
    strokeWeight(10);  
    rect(650, 100, 250, 300);  
}
```

FIGUUR 1.3

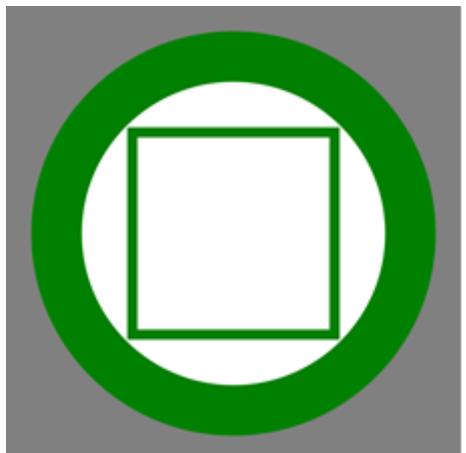


FIGUUR 1.4



Opdracht 1 vormen tekenen I

1. Open *H1001.js* (Hoofdstuk 1 Opdracht 1) in jouw *editor*. Dit is de code van *voorbeeld 1*. Bekijk het resultaat in de *browser*.
2. Pas de code aan zodat het canvas grijs (*grey*) van kleur wordt.
3. Pas de code aan zodat het canvas nog maar 450 pixels hoog is.
4. Zorg dat de groene cirkel zowel boven als links op een afstand van 25 pixels van de rand van het canvas komt te staan.
5. Voeg een tweede, witte cirkel toe met hetzelfde middelpunt als de groene cirkel en een diameter van 300.
6. Pas de regel voor de rechthoek aan, naar:
`rect(125, 125, 200, 200);`
7. Pas de code aan zodat het eindresultaat uit figuur 1.5 verschijnt met een vierkant canvas.
8. Er zijn nu twee regels met *ellipse*. Voorspel wat je zal zien als je de beide coderegels van plek laat wisselen, zodat hun volgorde andersom is. Probeer daarna uit.



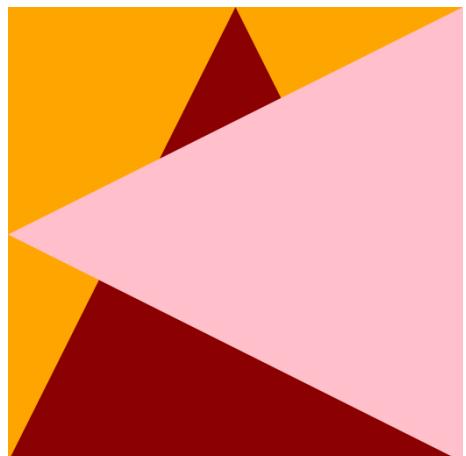
FIGUUR 1.5

Opdracht 2 vormen tekenen II

Als je in de P5-reference kijkt onder vormen (*shape*:

<https://p5js.org/reference/#group-Shape>) dan zie je een flink aantal commando's om vormen te tekenen. We proberen er een aantal uit.

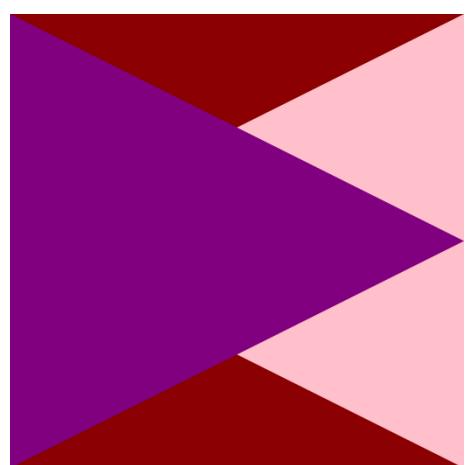
9. Open *H1002.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
10. De code bevat het commando *triangle*. Gebruik de *reference* om uit te zoeken wat de betekenis is van de zes getallen tussen de haakjes.
11. Pas de code aan, zodat het beeld uit figuur 1.6 ontstaat. (Het Engelse woord voor roze is *pink*.)
12. Gebruik de kleur paars (*purple*) om figuur 7 na te bouwen. Wie goed, kijkt maar twee driehoeken nodig!



FIGUUR 1.6

Per hoekpunt van de driehoek gebruiken we een x-waarde en een y-waarde. Op dezelfde manier kun je met het commando *quad* ook een vierhoek maken die niet per se rechthoekig is. Dat wordt al gauw onoverzichtelijk. Bovendien: hoe maak je dan een vijf- of zeshoek? P5 heeft een algemenere manier om een vorm met hoekpunten te tekenen. Zo'n vorm heet toepasselijk een *shape*. Hier een voorbeeld:

```
strokeWeight(5);
stroke('pink');
fill('darkred');
beginShape();
vertex(225,115);
vertex(300,225);
vertex(225,335);
vertex(0,225);
endShape(CLOSE);
```



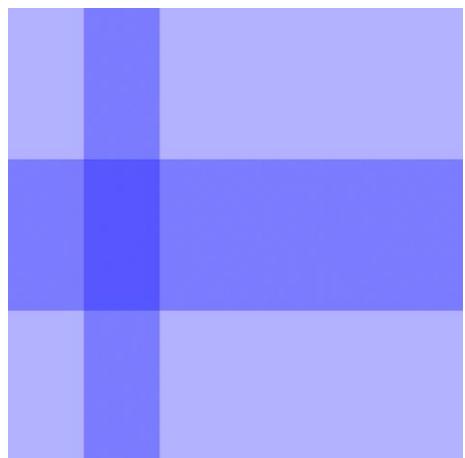
FIGUUR 1.7

13. Bekijk *voorbeeld 2* dat laat zien hoe je met *shape* werkt. Wat is een *vertex*? Wat doet *CLOSE*?
14. Neem de code hierboven over en plaats het in de *draw* onder de regels voor de driehoeken.

Opdracht 3 kleuren en doorzichtigheid

In de eerste opdrachten en voorbeelden hebben we kleur aangegeven met Engelse woorden zoals *pink* en *blue*. Als je wel eens een website hebt gemaakt, ben je misschien bekend met **hexadecimale kleurcodes** zoals `#FFC0CB`. Van een tekenprogramma zoals Photoshop of de natuurkunde-les ken je misschien het maken van kleuren met de primaire kleuren rood, groen en blauw in het **RGB**-systeem. Met P5 kun je ze allebei gebruiken. Zie de *reference*: <https://p5js.org/reference/#group-Color>

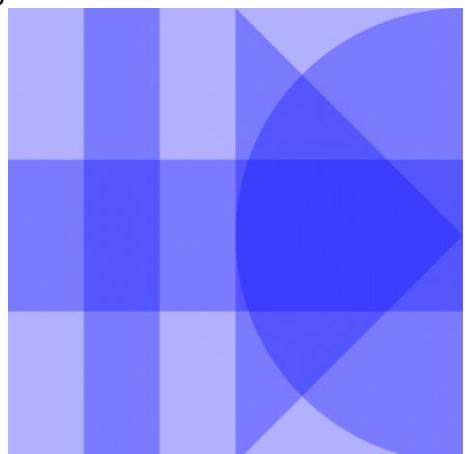
Met 255 rood (de maximale waarde), 192 groen en 203 blauw ofwel (255,192,203) maak je de kleur *pink*, net als met `#FFC0CB`. Deze site laat zien welke kleurnamen horen bij de kleurcoderingen: <https://htmlcolorcodes.com/color-names>



FIGUUR 1.8

Natuurlijk kun je ook je eigen kleurcombinaties maken.

15. Open *H1003.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
16. In regel 12 (kleurinstelling) en regel 13 (vorm) wordt een rood rechthoek getekend. Verklaar waarom je deze rode horizontale balk niet op het scherm ziet.
17. Vervang de regel die zorgt voor een groene kleur door `fill(0, 255, 0)`; Bekijk het resultaat.
18. We hebben nu een andere kleur groen. Gebruik bovenstaande link om de juiste kleurcodering voor de kleur *green* te vinden.
19. Vervang de regel die zorgt voor een blauwe kleur door een hexadecimale kleurcode die hetzelfde resultaat geeft als *blue*. Gebruik opnieuw de link naar de webpagina met kleurnamen.
20. In de code staan twee regels met `//` ervoor (er staat ook een `!` in de `setup!`). Verwijder de `//` voor beide regels. Eén van de regels die je nu actief hebt gemaakt is `fill(0, 0, 255, 0.3)`; Verwijder alle andere regels met het commando `fill`.
21. Als het goed is zie je als je de code uitvoert het resultaat uit figuur 1.8. We zien nu ook de horizontale balk die eerst onzichtbaar was. Wat is blijkbaar de betekenis van de `0.3` tussen de haakjes van `fill`?
22. Controleer je voorspelling door de waarde `0.3` te laten variëren tussen 0 en 1. Let op: gebruik een punt en geen komma. Wat zie je als je precies 0 of 1 invult?
23. Voeg een cirkel toe met een diameter van 450 en als middelpunt [450,225]. Zorg dat de vulkleur weer `0.3` bevat.
24. Voeg een driehoek toe zodanig dat de tekening in figuur 1.9 ontstaat. Kies daarna je eigen favoriete kleur.



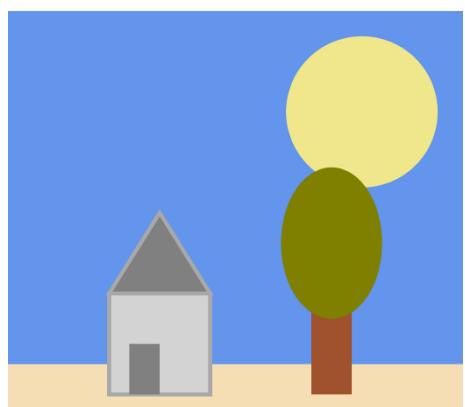
FIGUUR 1.9

Opdracht 4 huis met boom

In deze opdracht oefenen we nog een keertje met het tekenen van vormen, randen en kleuren. Ons doel is de afbeelding in figuur 1.10.

25. Open *H1004.js* in jouw *editor*. Bekijk het resultaat in de *browser*.

In de code staan meerdere commentaarregels met opdrachten om de tekening compleet te maken. Als je een programma aanpast, doe je er goed aan om dit in stapjes te doen. Kijk elke keer even of een stap gelukt is, voordat je verder gaat. Zo ben je uiteindelijk minder tijd kwijt.



FIGUUR 1.10

1.2 P5-functies: parameters en argumenten

In de eerste paragraaf heb je gewerkt met onder andere `fill`, `stroke`, `strokeWeight`, `rect`, `noStroke` en `ellipse` (figuur 1.11). We hebben dit tot nu toe commando's genoemd. Vanaf nu gebruiken we de term **functie**. Met een functie geef je de computer de opdracht om iets uit te voeren. Het was je misschien al opgevallen dat achter elke functie twee haakjes staan. Tussen die haakjes zet je de extra informatie die de functie nodig heeft om goed te werken.

Als je in de *reference* kijkt bij de uitleg van de functie `strokeWeight`, dan zie je het volgende:

Syntax

```
strokeWeight(weight)
```

Parameters

`weight` Number: the weight (in pixels) of the stroke

```
fill('green');
stroke('red');
strokeWeight(10);
rect(50,100,250,300);
noStroke();
```

FIGUUR 1.11

Met de term **syntax** (of: syntaxis) wordt het geheel aan taalregels van een programmeertaal bedoeld. De syntax vertelt je hoe je de taal, in ons geval Javascript, gebruikt om een programma te schrijven.

In de beschrijving staat een functie `strokeWeight`. In de uitleg zien we dat deze functie een gegeven nodig heeft om goed te kunnen werken. Zo'n gegeven heet een **parameter**. In dit geval is dit de lijndikte `weight`.

Een parameter heeft een algemene naam, zoals `weight`. Als je de functie wilt gebruiken, moet je een specifieke waarde voor de parameter meegeven: `strokeWeight(10)`

In dit voorbeeld wordt de waarde `10` als **argument** meegegeven. Niet alle functies hebben een argument nodig. Voorbeeld: met de functie `noStroke()` geef je aan dat je geen rand om een vorm wilt.

In de volgende opdrachten maak je kennis met een aantal P5-functies die erg handig zijn bij het tekenen.

Opdracht 5 verplaatsen: translate

Stel dat je het huis in figuur 1.10 naar rechts wilt verplaatsen met 15 pixels. Dan moet je zowel het huis, de deur als het dak aanpassen. Dat is veel werk, maar gelukkig kunnen we de functie `translate(x,y)` gebruiken. Hiermee verplaats je de oorsprong (met de coördinaten $x = 0$ en $y = 0$) naar een andere plaats op het canvas. Daarom heeft `translate` twee parameters `x` en `y` om de oorsprong horizontaal en verticaal te kunnen verplaatsen.

27. Open `H1O05.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Dit is het eindresultaat van opdracht 4 (figuur 1.10).
28. In regel 14 staat de functie `translate`. Hiermee willen we het huis verplaatsen.
Pas de argumenten aan, zodat $x = 90$ en $y = -10$. Wat is de betekenis van het minteken? Voorspel wat het resultaat van deze aanpassing zal zijn.
29. Klopte jouw voorspelling? In regel 26 staat nog een regel met `translate` die is uitgeschakeld met `//`. Haal de `//` weg. Begrijp je waarom nu alleen het huis verplaatst (en net nog niet)?

Deze opdracht laat zien hoe `translate` werkt, maar normaal gesproken pas je dit wel anders toe: als je de oorsprong verplaatst, kun je juist makkelijk vanaf de coördinaten [0,0] tekenen. Dit proberen we uit.

```
function draw() {
  noStroke();
  fill('tan');
  //translate(25,25);
  rect(0,0,400,400);
  //translate(200,50);
  fill('peru');
  rect(0,0,150,150);
}
```

FIGUUR 1.12

30. Bekijk de code uit figuur 1.12 en maak een schets van de tekening die zal ontstaan en een tweede schets van de tekening wanneer de `//` voor beide regels met `translate` worden weggehaald.
31. Verwijder alle coderegels in de `draw` en kopieer de code uit figuur 1.12. Klopte jouw tekening?

Opdracht 6 push & pop

Een ontwerper heeft een rij van zes vierkanten gemaakt zoals in de bovenste afbeelding van figuur 1.13. Het uiteindelijke beeld moet één vierkant bevatten dat er anders uitziet, zoals in de onderste afbeelding. De ontwerper heeft een paar aanpassingen gedaan voor de vierde figuur, met de middelste afbeelding als resultaat.

32. De programmeercode van de ontwerper staat in *H1O06.js*.

Open dit bestand in jouw *editor*. De aanpassingen die de ontwerper heeft gedaan zijn in het bestand gemarkeerd.

Wat gaat er mis? Als je in P5 een andere tekeninstelling kiest zoals een vulkleur, dan zal de computer deze vulkleur normaal gesproken blijven gebruiken totdat je weer een andere kleur kiest. Dat geldt voor alle tekeninstellingen.

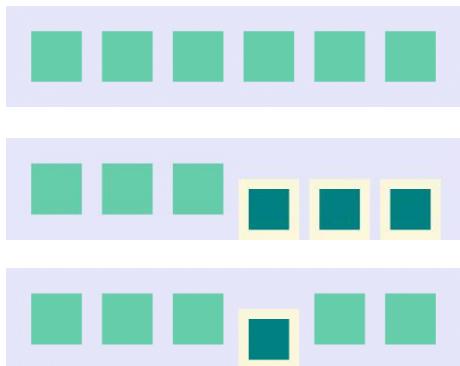
Als je terug wilt naar de vorige instellingen zou je alles weer terug moeten zetten zoals het was. Dat kan wel, maar is veel werk. De functies *push* en *pop* bieden uitkomst. Met *push* sla je de huidige tekeninstellingen op in het geheugen van de computer. Je kunt daarna de instellingen zoals *fill*, *stroke* maar ook *translate* tijdelijk aanpassen voor een stukje van de tekening. Ben je klaar en wil je weer verder tekenen met de bewaarde instellingen? Dan kun je die terughalen met *pop*.

33. Voeg voor de eerste aanpassing van het vierde vierkant de regel *push()*; in. Bekijk het resultaat: is er iets veranderd?

34. Voeg nu na het tekenen van het vierde vierkant de regel *pop()*; toe. Controleer of jouw tekening nu overeenkomt met de onderste afbeelding uit figuur 1.13.

35. Geef de vijf gelijke vierkanten in één handeling de kleur *thistle*.

36. Wat verwacht je te zien als we de regel *pop()*; niet na het vierde maar na het vijfde vierkant plaatsen? Probeer het uit en verklaar wat je ziet.



FIGUUR 1.13

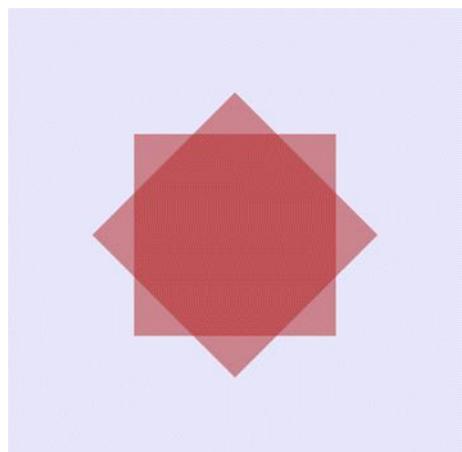
★ Opdracht 7 draaien I

In figuur 1.14 zie je twee half-doorzichtige vierkanten: het onderste vierkant is ‘normaal’, de bovenste is gedraaid over 45° zodat een ruit ontstaat.

In opdracht 2 hebben we gezien dat je met de functie *shape* willekeurige vormen kunt maken met hoekpunten. In principe zou je hiermee ook een ruit kunnen tekenen, maar dat is wel omslachtig. Het is makkelijker om gewoon een vierkant te draaien. Hiervoor is de functie *rotate* (Engels: draaien of roteren).

37. Open *H1O7.js* in jouw *editor* en bekijk het resultaat.

38. In regel 15 staat *rotate(0)*. Verander het argument van de functie in *7*. Rond welk punt is het vierkant gedraaid? En over hoeveel graden is hij gedraaid?



FIGUUR 1.14

Waarschijnlijk is het resultaat anders dan je verwacht. Eén van de redenen is dat Javascript hoeken niet in graden maar radialen rekent.

39. Voeg de regel *angleMode(DEGREES)*; toe aan *setup* en geef *rotate* als argument *45* mee.

P5 heeft een aantal standaard tekeninstellingen. Eén daarvan is dat bij een bewerking op een vierkant of rechthoek vanaf linksboven wordt gerekend. Daarom draait het vierkant niet om zijn as.

40. Voeg de regel *rectMode(CENTER)*; toe aan de *setup* en bekijk het resultaat.

41. Pas de argumenten van *translate* aan, zodat de figuur weer in het midden van het canvas staat.

42. Plaats een geel vierkant met zijde 50 in het midden van het vierkant. Neem als RGB-kleurcode 255,225,0. Gebruik *push* en *pop* om te zorgen dat het vierkant niet gedraaid staat als een ruit.

1.3 variabelen

In de laatste opdrachten hebben we meerdere vierkanten moeten tekenen met steeds hetzelfde formaat. Hiervoor hebben we telkens hetzelfde getal als argument meegegeven aan de functie `rect`. Voor zes vierkanten betekent dit twaalf keer (lengte en breedte) dezelfde waarde invullen!

Als je de vierkanten iets groter wilt maken, moet je dus ook op twaalf verschillende plaatsen de lengte van een zijde aanpassen terwijl die eigenlijk steeds gelijk is. Zou het niet fijn zijn als we de computer één keer kunnen vertellen dat de *zijde* van de vierkanten bijvoorbeeld 150 pixels moet zijn? En dat hij het daarna zelf onthoudt? Dat kan met een **variabele**.

```
var lengte = 5;
var breedte = 8;
var omtrek;

function draw() {
    omtrek = 2*lengte + 2*breedte;
    text("De omtrek is " + omtrek, 50, 50);
    rect(0,0,lengte,breedte);
}
```

FIGUUR 1.15

Als je een nieuwe variabele maakt – dat heet **declareren** –, dan geef je een naam aan een klein stukje van het geheugen van de computer. In de code van figuur 1.15 worden drie variabelen gedeclareerd. Twee daarvan krijgen bovendien een beginwaarde mee. De eerste keer dat een variabele een waarde krijgt noemen we **initialisatie**.

De variabele `omtrek` krijgt geen beginwaarde, omdat we die door de computer willen laten berekenen. De opdracht daarvoor staat in de `draw`. Dit heet een **toewijzing**: `omtrek = 2*lengte + 2*breedte;`

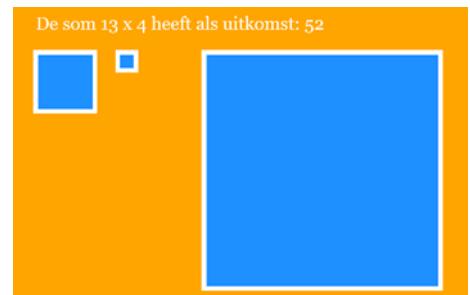
De functie `text` zet de tekst *De omtrek is 24* op het scherm met:

```
text("De omtrek is " + omtrek, 50, 50);
```

Gewone tekst zet je tussen aanhalingsstekens, getallen en variabelen niet. Je ‘plakt’ de losse stukjes aan elkaar door er een `+` tussen te zetten. Naast de variabelen die je zelf kunt maken, kent P5 ook een aantal eigen variabelen zoals `mouseX`, `mouseY`, `width` en `height`. Hun betekenis leer je in opdracht 9.

Opdracht 8 variabelen

43. Open `H1O08.js` in jouw *editor*. Dit is de code van *voorbeeld 3*. Bekijk het resultaat in de *browser*.
44. Geef variabele `B` de waarde 11 en bekijk het resultaat.
45. Zorg dat de rekensom $21 - 9 = 12$ op het scherm verschijnt.
(Let ook op de tekst!)
46. In figuur 1.16 wordt de rekensom $13 \times 4 = 52$ getoond. Doe dit ook voor jouw canvas. Pas de grootte van het canvas aan, zodat ook het grote vierkant in het beeld past.



FIGUUR 1.16

Opdracht 9 P5-variabelen en tekst

47. Open `H1O09.js` in jouw *editor*. Dit is de code van *voorbeeld 4*. Bekijk het resultaat in de *browser*.
48. In welke variabele slaat P5 het aantal pixels van de hoogte van jouw browserscherm op?
49. In welke variabele slaat P5 het aantal pixels van de hoogte van jouw canvas op?

Regel 17 bevat de code: `text("mouseX:" + mouseX + "\nmouseY:" + mouseY, mouseX, mouseY);` die ervoor zorgt dat er een stuk tekst meebeweegt met de x-positie (`mouseX`) en y-positie (`mouseY`) van de muis.

50. Verklaar dat deze gele tekst achter het blauwe tekstvak verdwijnt.
51. In regel 17 staat "`\nmouseY:`". Toch zie je alleen `mouseY` in beeld. Haal `\n` weg. Wat zie je?
52. We ronden de positie van de muis af naar hele pixels met de functie `round`. Verander regel 17 naar:
`text("mouseX:" + round(mouseX) + " mouseY:" + round(mouseY), mouseX, mouseY);`
53. In de code staan vier functies die beïnvloeden hoe de tekst wordt getoond (`textFont`, `textSize`, `textLeading` en `textAlign`). Zoek uit wat deze functies doen en experimenteer ermee.

Gebruik eventueel de *reference*: <https://p5js.org/reference/#group-Typography>

Opdracht 10 simpel tekenprogramma

We gaan de P5-variabelen `mouseX` en `mouseY` gebruiken om zelf een tekenprogramma te maken.

54. Open `H1O10.js` in jouw `editor`. Bekijk het resultaat in de `browser`. Hoe heeft de programmeur er zonder hoofdrekenen voor gezorgd dat de stip precies in het midden staat?
55. Pas de argumenten van de functie `ellipse` aan, zodat de stip jouw muis volgt.

Het is je misschien opgevallen dat `background('lavender');` in dit programma niet in de `setup` maar in de `draw` is geplaatst.

56. Waarom is dat zo? Verplaats de genoemde regel van de `draw` naar de `setup`. Wat is er veranderd?

Nu de achtergrondkleur in de `setup` wordt ingesteld, wordt deze alleen in het begin gekleurd. Alles wat daarna verandert, blijft in beeld staan. We kunnen nu tekenen, maar de tekst bovenaan wordt onleesbaar. Gelukkig is er een simpele oplossing:

57. Voeg aan het begin van de `draw` een regel toe die een rechthoek tekent met de (vul-) kleur `wheat`. Gebruik de standaard P5-variabele voor de canvasbreedte en zorg voor een hoogte van 30 pixels zodat bovenaan een balk wordt getekend zoals in figuur 1.17.



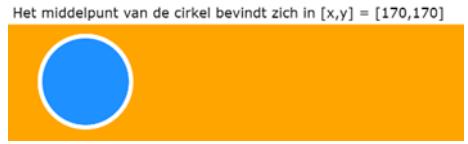
FIGUUR 1.17

1.4 draw is een loopfunctie

Alle programma's en voorbeelden die je tot nu toe hebt gezien hebben een `setup` met de regel `noLoop();`. Was het je opgevallen dat deze regel in de laatste twee opdrachten was uitgeschakeld met `//` (commentaar)? In paragraaf 1.1 is genoemd dat in de `setup` de begininstellingen van het programma worden beschreven en dat de `draw` het hoofdprogramma is. In P5 is `draw` wel een bijzonder geval, want de code binnen de `draw` wordt telkens opnieuw uitgevoerd. Een functie die regels code steeds opnieuw herhaalt noemen we een **loopfunctie** (*loop* is Engels voor 'lus' of 'herhalingslus'). In opdracht 9 en 10 hebben we voor het eerst het voordeel van zo'n herhalingslus gezien: omdat de `draw` steeds opnieuw wordt uitgevoerd, kan worden gevuld wat de actuele plaats van de muis is. Met die informatie krijgt de canvastekening steeds een update. Hoe vaak gebeurt dat eigenlijk?

In voorbeeld 5 (waarvan figuur 1.18 een screenshot toont) zie je hoe een cirkel zich van links naar rechts verplaatst. Hierbij gebruikt men:

- in de `setup`: `frameRate(10);`
Dit zorgt ervoor dat de animatie met 10 *frames* (beeldjes) per seconde wordt getekend, ofwel: dat de functie `draw` 10 keer per seconde wordt uitgevoerd.
- aan het einde van de `draw`: `horizontaal += 2;`
Dit zorgt ervoor dat de variabele `horizontaal`, die bepaalt hoe ver naar rechts de cirkel wordt getekend, elk frame met 2 wordt opgehoogd, zodat de cirkel steeds verder naar rechts wordt getekend.



FIGUUR 1.18

De regel `horizontaal += 2;` had ook geschreven kunnen worden als `horizontaal = horizontaal + 2;` Dat is iets langer, maar voor sommige mensen wel duidelijker. Je moet deze regel lezen als: *de nieuwe waarde van de variabele 'horizontaal' is gelijk aan de oude waarde van de variabele 'horizontaal' plus 2*. Javascript heeft handige, korte notaties voor berekeningen. Wil je één optellen of aftrekken van de waarde van een variabele, dan gebruik je `horizontaal++;` en `horizontaal--;`. Verdubbelen? Gebruik `horizontaal *= 2;` Halveren? Gebruik `horizontaal /= 2;` (of `horizontaal *= 0.5;`)

Het is niet altijd nodig om de `draw`-functie eindeloos uit te voeren. Als de code binnen de `draw` maar één keer moet worden uitgevoerd of als je het herhalen wilt stoppen, dan gebruik je daar de functie `noLoop()`.



Opdracht 11 automatische bewegingen

58. Open `H1O11.js` in jouw *editor*. Dit is de code van voorbeeld 5.
Bekijk het resultaat in de *browser*.

De blauwe cirkel noemen we cirkel A. Voor de beginpositie van A gebruiken we de variabele `horizontaalA`. We gaan een tweede cirkel B maken. Zie figuur 1.19.

positie A = 454 positie B = 667



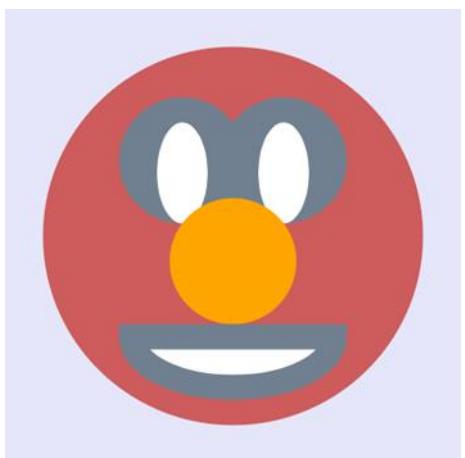
FIGUUR 1.19

59. Declareer voor de beginpositie van B een variabele `horizontaalB` en initialiseer die met de waarde 500.
60. Teken cirkel B even groot als A, maar met de kleur `darkred`. Natuurlijk maak je voor de horizontale positie gebruik van `horizontaalB`.
61. Zorg dat de positie van B elke keer dat de `draw` wordt uitgevoerd met 1 wordt verhoogd.
62. Pas het argument van de functie `frameRate` aan naar 50. Wat is het resultaat?
63. Pas de tekst aan zodat er komt te staan:
positie A = 454 positie B = 667 (Dit is maar één voorbeeld: de waarden veranderen steeds.)

Opdracht 12 maak kennis met JOS

In deze opdracht maken we kennis met JOS. JOS is een *game character*: een poppetje dat we in meerdere opdrachten tegen zullen komen (figuur 1.20). Zijn naam is een afkorting van *Javascript Object Sprite*. Een *sprite* is veelgebruikte term voor een tweedimensionaal plaatje of animatie. In hoofdstuk 2 leer je wat een object is.

64. Open `H1O12.js` in jouw *editor*. Bekijk het resultaat. Je ziet een behoorlijk aantal regels die ervoor zorgen dat JOS wordt getekend, maar het is niet nodig om die nu te bestuderen.
65. Voeg de regel `xJOS--;` toe aan het eind van de `draw` en bekijk het resultaat. Wat betekent deze regel?
66. Gebruik `yJOS--;` zodat JOS naar linksboven beweegt.
67. Pas de regel aan naar `yJOS -= 2;` zodat JOS twee keer zo snel omhoog beweegt als dat hij naar links beweegt.
68. Pas regel 16 (`translate`) aan, zodat JOS meebeweegt met jouw muis.



FIGUUR 1.20

Opdracht 13 P5-functies voor beperken en schalen

In de vorige opdracht was het mogelijk om JOS van het canvas te laten verdwijnen. In veel spellen is het de bedoeling dat je *game character* zichtbaar blijft, ofwel: op het canvas blijft. Voor jou als programmeur betekent dit dat je de beweging van JOS moet inperken.

69. Open `H1O13.js` in jouw *editor*. Bekijk het resultaat. Wat gebeurt er als je de muis beweegt?

Als het goed is heb je gemerkt dat je JOS wel kunt bewegen, maar dat je niet meer alle vrijheid hebt. Oorzaak is de regel `xJOS = constrain(mouseX, 100, 450);` (`constrain` is Engels voor beperken). JOS mag bewegen volgens de horizontale muispositie (`mouseX`), maar alleen tussen de waarden 100 en 450.

70. Pas de code aan, zodat JOS links en rechts precies tot de rand van het canvas kan bewegen.
HINT: in welke regel kun je zien hoe groot JOS is?
71. Breid de code uit, zodat JOS ook boven en onder precies tot de rand van het canvas kan bewegen.

Als we Jos wat kleiner willen tekenen, lijkt dat een hele klus, omdat hij is opgebouwd met flink wat regels programmeercode. Gelukkig heeft P5 een functie om de omvang van tekeningen te schalen: `scale(1);` Met de waarde 1 wordt alles op normale grootte (100%) getekend, met b.v. `scale(0.5);` op 50%.

72. Teken Jos op 50% van zijn normale grootte. Wat zie je? LET OP: volgt Jos de muis nog wel goed?
73. Verplaats de regel `scale(0.5);` zodat hij meteen na `push();` staat. Probleem opgelost?



Opdracht 14 snelheid

In deze opdracht gaan we JOS uit zichzelf laten bewegen.

74. Open *H1O14.js* in jouw *editor* en bekijk het resultaat.
75. In regel 15 is `yJOS -=`; uitgezet met `//`. Haal deze weg en bekijk het resultaat.
76. Wat gebeurt er als JOS bovenaan is? Waarom?
77. Declareer een variabele `snelheidJOS` en geef deze de waarde 17.
78. Pas de regel `yJOS -=`; aan zodat JOS niet met stapjes van 1 maar van 17 naar boven beweegt. Gebruik hiervoor de variabele `snelheidJOS`.

Jos beweegt nu sneller naar boven, maar zijn beweging is niet heel natuurlijk. Als jij iets omhoog gooit, dan gaat het steeds langzamer omhoog, omdat de snelheid afneemt.

79. Maak onder de regel die je net hebt aangepast een nieuwe regel die ervoor zorgt dat de snelheid van JOS bij elke loop van de `draw` afneemt met 0,5.
(LET OP: in Javascript gebruik je niet een komma maar een punt.)
80. Zorg dat de *snelheid* bovenin wordt getoond, naast x en y.
81. Verklaar de beweging die je nu ziet. Leg bovendien uit waarom de snelheid bovenin blijft veranderen, ook als JOS weer 'op de grond' staat.

Opdracht 15 een functie om JOS te tekenen

We hebben kennis gemaakt met JOS, een poppetje dat je nog vaker zal tegenkomen in deze module. Er zijn behoorlijk wat coderegels nodig om hem te tekenen. Het zou mooi zijn als we JOS met één commando (functie) op het scherm zouden kunnen krijgen.

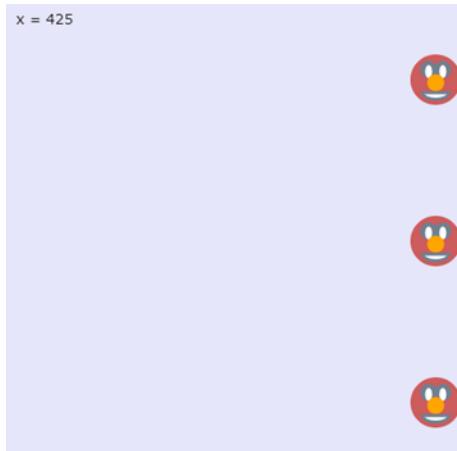
82. Open *H1O15.js* in jouw *editor*. In regel 16 wordt de functie `tekenJos` aangeroepen. Deze functie heeft twee parameters die bepalen waar JOS wordt getekend.
83. JOS wordt nu horizontaal in het midden getekend. Zorg dat (het middelpunt van) JOS 75 pixels vanaf de linkerkant van het canvas wordt getekend.

```
translate(0, 160);  
tekenJos(xJOS, yJOS);
```

FIGUUR 1.21

De functie `tekenJos` is natuurlijk geen standaard-functie van JS of P5. Hij wordt gemaakt in de regels 19 t/m 39. Daar zie je alle regels terug waarmee JOS kan worden getekend. Het gebruik van de functie lijkt meer werk (hoewel de code in de `draw` nu wel overzichtelijker is). Functies worden pas echt handig als je ze vaker gebruikt.

84. Voeg onder regel 16 de twee coderegels toe uit figuur 1.21 en bekijk het resultaat. Voeg beide regels daarna nogmaals toe, zodat je JOS drie keer op het scherm ziet.
85. Gebruik de eigenschappen van de loopfunctie `draw` om alle drie de versies van JOS naar rechts te laten bewegen met stapjes van 3 pixels.
86. Gebruik de functie `constrain` om te zorgen dat de poppetjes stoppen op het moment dat ze de rechterkant van het canvas hebben bereikt (zie figuur 1.22).



FIGUUR 1.22

Opdracht 16 obfuscator I: schuivende bollen

Een *obfuscator* is een klein programmaatje dat programmeercode onleesbaar kan maken. Dit kan handig zijn wanneer je jouw code niet met anderen wil delen.

87. Bekijk *OBF01*. De bijbehorende code is onleesbaar gemaakt m.b.v. <https://obfuscator.io>
88. Open *H1O16.js* in jouw *editor*. Breid deze code uit zodat je hetzelfde resultaat bereikt als in *OBF01*.

1.5 zelf functies maken

In opdracht 15 heb je voor het eerst kunnen zien dat je zelf **functies** kunt maken; in dit geval: `function tekenJos(x,y)`.

Aan de hand van *voorbeeld 6* gaan we functies nader bestuderen.

In figuur 1.23 zie je een functie die een huis tekent. Als je zelf een functie maakt, begin je met `function` gevolgd door de (door jou bedachte) naam van de functie. Achter die naam moet je altijd haakjes `()` zetten, dus ook als de functie geen **parameters** heeft. Tussen de accolades `{}` zet je vervolgens alle regels code die moeten worden uitgevoerd als de functie wordt aangeroepen. Dit noemen we een **functiedefinitie**.

Let op: de functie maak je buiten de `draw`. Het **aanroepen** van de functie doen we (nu) wel in de `draw` met (alleen) `tekenHuis()`. Alleen als je de functie aanroept, wordt het huis ook echt getekend!

figuur 1.24 toont een functie met een **parameter**: `tekenBoom(x)`. Dit betekent dat je een argument aan de functie meegeeft, waarmee de functie aan de slag kan. Deze waarde wordt bijvoorbeeld gebruikt voor de rechthoekige stam van de boom: `rect(x, 130, 40, 130)`.



FIGUUR 1.25

```
function tekenHuis() {  
  push();  
  strokeWeight(4);  
  stroke('darkgrey');  
  fill('lightgray');  
  rect(100,180,100,100);  
  // zie vb 6 voor meer code  
  pop();  
}
```

FIGUUR 1.23

```
function tekenBoom(x) {  
  push();  
  noStroke();  
  fill('sienna');  
  rect(x,130,40,130);  
  fill('olive');  
  ellipse(x+20,130,100,150);  
  pop();  
}
```

FIGUUR 1.24

Het gebruik van parameters heeft enorme voordelen. Het huis wordt altijd op dezelfde plek getekend, maar de plek waar de boom getekend wordt kun je nu zelf sturen door bij het aanroepen een argument mee te geven: `tekenBoom(700)`.

Als je één keer een functie hebt gemaakt, kun je hem zo vaak aanroepen als je wilt. Twee of drie bomen tekenen is nu een fluitje van een cent.

Merk op dat we binnen de functies steeds `push()` en `pop()` hebben gebruikt. Dit voorkomt dat je per ongeluk tekeninstellingen (zoals de vulkleur) ‘aan laat staan’ als je de functie gebruikt.



Opdracht 17 functies maken en aanroepen

89. Open *H1O17.js* in jouw *editor*. Deze gebruikt de code van *voorbeeld 6*. Bekijk het resultaat in de *browser*.
90. Voeg drie bomen toe aan de tekening. Vul voor de horizontale positie de argumenten 50,150 en 250 in. Zorg dat de bomen achter het huis staan.
91. Maak een nieuwe functie (buiten de `draw`!) door de code uit figuur 1.26 over te schrijven of te kopiëren.
92. De functie regel `tekenZon` heeft twee parameters. Welke betekenis hebben deze parameters?
93. Voeg de regel `tekenZon(500,1)` toe na regel 11.
Wat verwacht je te zien? controleer of je voorspelling klopt.
94. Pas de regel aan naar: `tekenZon(mouseX,schaal)`.
Wat verwacht je te zien? Kijk opnieuw of je voorspelling klopt.

```
function tekenZon(x,s) {  
  push()  
  fill('red');  
  scale(s);  
  ellipse(x,200,300,300);  
  pop();  
}
```

FIGUUR 1.26

Opdracht 18 vallende ster

In de theorie hebben we functies met nul en één parameter gezien, maar je kunt zoveel parameters toevoegen aan een functie als je zelf wilt. In deze opdracht kijken we naar een functie met drie parameters.

95. Open `H1O18.js` in jouw *editor*. Bekijk het resultaat in de *browser*.
96. In de loopfunctie `draw` staan maar twee regels. Die roepen de twee functies `background` en `tekenSter` aan. Voeg twee regels toe, zodat bij elke loop de variabele `yPositie` met 1 wordt verhoogd en de variabele `xPositie` met 5 wordt verhoogd.
97. Voorspel wat je gaat zien als je `//` voor `background` zet. Controleer je voorspelling.
98. Misschien herinner je jezelf nog dat de vierde parameter van `background` de mate van doorzichtigheid aangeeft. Haal de `//` voor `background` weer weg en geef de laatste parameter het attribuut 0.1 mee. Bekijk het resultaat. Kun je verklaren wat je ziet?

De functie `tekenSter` heeft op dit moment twee parameters. In regel 3 is een variabele `schaal` gedeclareerd, waarmee we zelf willen regelen met welke grootte de ster wordt getekend. Daarvoor is binnen `tekenSter` al de regel `scale(1);` toegevoegd.

99. Voer de volgende opdrachten uit:
 - Pas de functie `tekenSter` aan zodat een derde parameter `s` wordt gebruikt voor het tekenen van de ster op een bepaalde schaal
 - Zorg dat er bij de aanroep van de functie `tekenSter` (in de `draw`) gebruik gemaakt wordt van de variabele `schaal`.
100. Zorg dat de variabele `schaal` bij elke loop met 0,05 wordt verhoogd. Bekijk nu het eindresultaat.



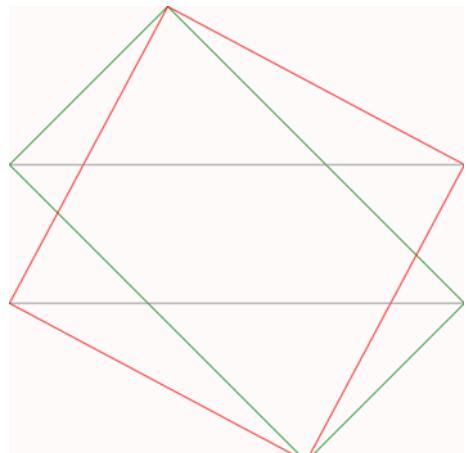
Opdracht 19 obfuscator II: lijnenspel

101. Bekijk `OBF02`. De bijbehorende code is onleesbaar gemaakt m.b.v. <https://obfuscator.io>
102. Open `H1O19.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet nu alleen twee grijze lijnen.

De bedoeling is dat je `OBF02` probeert na te bouwen. Om je te helpen is `function tekenLijnen(p)` alvast voor je gemaakt. Hierin zie je voor ons nog onbekende code `line(0,p,width,p);`. Gebruik eventueel de `reference` om de precieze werking te achterhalen.

In de code is al een begin gemaakt met de twee functies `tekenRechthoek(p)` en `tekenVierkant(p)`.

103. Vul beide functies aan zodat het beeld overeenkomst met `OBF02`.
104. Wat is de betekenis van de regels 16 t/m 18, denk je?



FIGUUR 1.27

Opdracht 20 JOS laten groeien

Tot nu toe heb jij als programmeur beslist hoe de tekeningen eruitzien en hoe vormen bewegen. Het leuke van spellen is nu juist dat de speler zelf iets kan veranderen. In deze opdracht kijken we naar coderegels die daarvoor kunnen zorgen. In de volgende paragraaf krijg je er meer uitleg bij.

105. Open `H1O20.js` in jouw *editor*. Bekijk het resultaat in de *browser*.
106. Bekijk de coderegels. We maken hier opnieuw gebruik van dezelfde functie om JOS te tekenen.
107. Sommige programmeerregels lezen als Engelse zinnen. Wat betekent `if (keyIsPressed == true)` denk je?
108. Hoe zou jij het woord `else` vertalen?
109. Verander deze regel in `if (mouseIsPressed == true)`. Dus: verander het woord `key` in `mouse`. Welk resultaat verwacht je?
110. Wat gebeurt er als de variabele `zoomniveau` kleiner dan 0 wordt?

1.6 Voorwaarden: if en else

Als je opdracht 20 hebt gemaakt, dan heb je al gewerkt met voorwaarden. Bijna alle computerprogramma's reageren op de invoer van de gebruiker van het programma. Ze doen dus niet zomaar iets, maar alleen als de gebruiker daar om vraagt.

Wanneer een opdracht niet altijd moet worden uitgevoerd, maar alleen in speciale gevallen, spreek je van een **voorwaarde**. Alleen wanneer aan de voorwaarde is voldaan, wordt de opdracht uitgevoerd. In voorbeeld 7 zie je een bewegende bal die, als hij de rand bereikt, omkeert qua snelheid. De bijbehorende code zie je in figuur 1.28.

Een voorwaarde begint met `if ()`. Tussen de haakjes zet je de voorwaarde (-n) of eis (-en) waaraan moet worden voldaan om één of meerdere coderegels uit te voeren, zoals `if (x > 880)`. Wat de computer moet doen als aan de eis is voldaan, staat tussen `{ }`.

Als je in voorbeeld 7 met de muis klikt, wordt de cirkel soms groen, maar soms ook niet, ook al klik je met de muis. Dat komt omdat er een dubbele voorwaarde is gebruikt:

```
if (mouseIsPressed == true && snelheid == 5) { fill('green'); }
```

De snelheid wisselt hier tussen (+) 5 (naar rechts) en -5 (naar links). Alleen als het waar is dat er op de muis wordt gedrukt **en** tegelijkertijd ook geldt dat de snelheid 5 is, wordt de vulkleur groen gebruikt. Als je meerdere eisen wilt stellen, gebruik je `&&`. Merk op dat je om te kijken of de snelheid 5 is een dubbele `=` (`==`) moet gebruiken. Een enkele `=` gebruik je om een waarde aan een variabele toe te kennen.

Als je nog eens kritisch kijkt naar de code in figuur 1.28 dan zie je dat dat er eigenlijk twee voorwaarden zijn waarbij dezelfde handeling moet plaatsvinden (namelijk: snelheid omkeren). Dit kunnen we samenvoegen tot één voorwaarde: als `x` groter dan 880 **of** kleiner dan 120 is, moet de snelheid omkeren. Dat doe je zo: `if (x > 880 || x < 120) { snelheid = -1*snelheid; }`

Als je **of** wilt aangeven gebruik je dus `||`. De elementen `||`, `&&`, `==` en `>` zijn voorbeelden van **logische operatoren**. In de opdrachten gaan we er mee oefenen.

In voorbeeld 7 wordt de cirkel groen als aan bepaalde eisen is voldaan. Maar wat moet er gebeuren als dat niet zo is? Dat staat beschreven in het gedeelte met `else { }`. Een `else` maak je altijd in combinatie met een `if`. De vaste vorm zie je in figuur 1.29.

Merk op dat er achter de `else` geen nieuwe voorwaarde komt. De voorwaarde tussen `()` is altijd iets dat WAAR (`true`) of NIET WAAR (`false`) is. Later komen we hier nog uitgebreid op terug.

```
if (x > 880) {  
    snelheid = -1*snelheid;  
}  
  
if (x < 120) {  
    snelheid = -1*snelheid;  
}  
  
x += snelheid;  
ellipse(x, 170, 200);
```

FIGUUR 1.28

```
if ( VOORWAARDE ) {  
    wat moet er gebeuren  
    als het WAAR is?  
}  
  
else {  
    wat moet er gebeuren  
    als het NIET WAAR is?  
}
```

FIGUUR 1.29

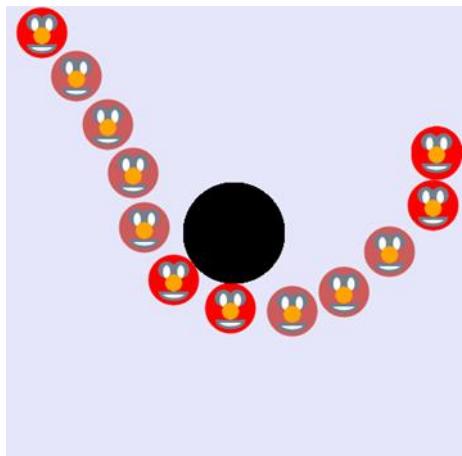
Opdracht 21 oefenen met if & else

111. Open `H1O21.js` in jouw *editor*. Dit is de code van voorbeeld 7. Bekijk het resultaat in de *browser*.
112. De code bevat de regels uit figuur 1.28. In de theorie wordt uitgelegd hoe deze twee losse voorwaarden kunnen worden samengevoegd tot één `if`. Voer dat uit in jouw bestand.
113. In de voorwaarde voor de vulkleur staat `&& snelheid == 5`. Voorspel wat je ziet als `==` wordt veranderd in: A) `snelheid < 5`; B) `snelheid > 5`; C) `snelheid >= 5`; Check vervolgens of je voorspelling klopt.
114. Zorg dat de cirkel groen wordt als er wordt geklikt **of** als de cirkel naar rechts beweegt.
115. Zorg dat de cirkel groen is als hij naar rechts beweegt, blauw als hij naar links beweegt en rood als er iemand klikt.
116. Zorg dat de diameter van de cirkel alleen 100 is als er iemand klikt en weer 200 als dat niet zo is.
117. Wat gebeurt er als je bij `mouseIsPressed == true` in plaats van `true` kiest voor `false`?

Opdracht 22 pas op de randen: afstand bepalen

Bij veel spellen is het belangrijk om vast te stellen of je als speler iets hebt geraakt of dat je er voldoende dichtbij in de buurt bent. Maar hoe stel je nu vast of dit het geval is?

In figuur 1.30 zie je JOS op verschillende plaatsen in een canvas met een zwarte bol. Als je goed kijkt zie je dat hij zowel in de buurt van een rand als in de buurt van de zwarte bol een extra rood gezicht krijgt. Dit gedrag van JOS gaan we stap voor stap programmeren.



FIGUUR 1.30

118. Open `H1O22.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Beweeg hierbij JOS langs alle randen van het canvas. Jos verandert alleen rechts van kleur. De functie `tekenJOS` heeft hiervoor een extra parameter `kleur` gekregen.
119. Bestudeer de code. In welke regels is ervoor gezorgd dat JOS niet buiten beeld kan verdwijnen?
120. Verklaar het gebruik van de waarde 25 in deze coderegels.
121. Hoe dicht moet JOS bij de rand zijn om van kleur te veranderen? Hoeveel pixels is dit? In welke coderegel kun je dit zien?
122. Pas de code aan zodat JOS ook aan de linkerkant rood kleurt als hij te dichtbij de rand komt. Houd dezelfde marge aan als aan de rechterkant.
123. Pas de code verder aan zodat JOS ook boven en onder rood kleurt (met dezelfde marge).

Nu JOS rood kleurt bij de randen van het canvas, is de zwarte bol aan de beurt. In figuur 1.31 bevindt het middelpunt van Jos zich in het punt $[x,y] = [315,105]$. Omdat het canvas 450 pixels breed en hoog is en de zwarte bol in het midden staat, bevindt het middelpunt van de zwarte bol zich in $[x,y] = [225,225]$ ($= 450/2$).

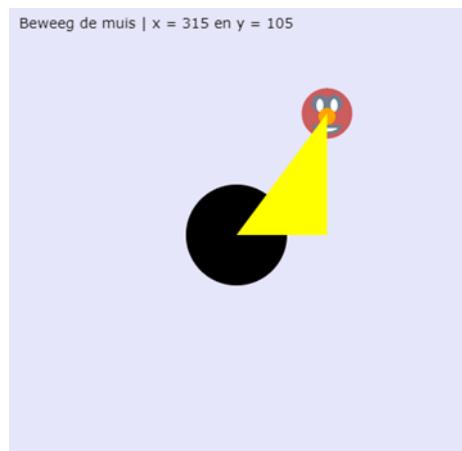
In de onderbouw heb je geleerd hoe je met de Stelling van Pythagoras de afstand tussen de twee punten kunt bepalen. Als we in de gele driehoek van figuur 1.31 de horizontale zijde a noemen en de verticale zijde b dan kunnen we over de schuine zijde c zeggen:

$$a^2 + b^2 = c^2$$

Met $a = 315 - 225 = 90$ en $b = 225 - 105 = 120$ volgt dan voor c :

$$c^2 = 90^2 + 120^2 = 22.500 \Leftrightarrow c = \sqrt{22.500} = 150$$

P5 heeft een ingebouwde functie die dit voor je uitrekent: de *dist*-functie (*distance* is Engels voor afstand). De coderegel
`afstand = dist(315, 105, 225, 225);`
zorgt ervoor dat de variabele `afstand` nu de waarde 150 krijgt.



FIGUUR 1.31

124. Gebruik bovenstaande coderegel (met `dist`) en pas deze aan zodat de afstand tussen de punt van de muis (de cursor; het middelpunt van JOS!) en het middelpunt van de zwarte bol wordt berekend.

We willen dat JOS rood kleurt als hij zich 5 pixels (of minder) van de zwarte bol bevindt. Dit doen we door de functie *dist* te gebruiken.

125. Leg uit dat de variabele `afstand` op dat moment **niet** de waarde 5 heeft. Welke waarde heeft de variabele `afstand` dan wel?
126. Pas de code aan, zodat JOS niet alleen rood kleurt in de buurt van de randen van het canvas, maar ook wanneer hij in de buurt van de zwarte bol komt.
127. Zorg dat bovenin, behalve de reeds getoonde tekst, ook de actuele waarde van de variabele `afstand` wordt getoond.
128. Pas de code aan, zodat de achtergrond *geel* (*yellow*) kleurt als JOS zich op de linkerhelft van het canvas bevindt en wit (*white*) kleurt als JOS zich op de rechterhelft van het canvas bevindt.
Natuurlijk zorg je ervoor dat de tekst bovenaan nog steeds zichtbaar is!

Opdracht 23 keyIsDown: raak het andere blokje

We hebben al kennis gemaakt met een standaard P5-functie die reageert als er wordt geklikt (bij `mouseIsPressed`). Er zijn ook functies die reageren op het toetsenbord. In deze opdracht kijken we naar bij `keyIsDown`. De functie `mouseIsPressed` heeft geen parameter, want er is maar één muis), maar aan `keyIsDown` moet je een argument meegeven. In `H1O23.js` hebben we al een beginnetje gemaakt.

129. Open `H1O23.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Druk op de pijltjestoetsen. Welke richtingen werken?

130. Het blokje blijft in het canvas, door regel 23:

```
y = constrain(y, 0, height - 100);
```

Verklaar het gebruik van `height - 100` in deze functie.

131. Pas de code aan, zodat het blokje ook naar links en rechts kan bewegen (met de bijbehorende pijltjestoetsen).

HINT: Gebruik **LEFT** en **RIGHT**.

132. Het blokje kan nu links en rechts het canvas verlaten. Blokkeer dit met de functie `constrain`.



FIGUUR 1.32

Het was je vast al opgevallen dat het rechterblokje fel groen (*chartreuse*) kleurt, als het vierkantje zich op een bepaalde hoogte bevindt zoals in figuur 1.32. Om precies te zijn is gezorgd dat het blokje kleurt als de hoogtes van beide blokjes elkaar (deels) overlappen. Dit is bereikt met `if (y >= 75 && y <= 225)`.

133. Leg uit waarom hier voor de waarden `75` en `225` is gekozen.

134. Waarom staat er in deze coderegel `&&` en niet `||`? Voorspel wat er gebeurt als we dit aanpassen.

Controleer jouw voorspelling. Vergeet niet om daarna de `&&` weer terug te zetten.

135. Breid de code uit, zodat het rechterblokje groen kleurt wanneer het geraakt wordt door het vierkant.

★ Opdracht 24 jager en prooi

In de vorige opdracht hebben we kunnen zien hoe je de computer kunt laten vaststellen of een vierkant een blokje raakt. Als we het blokje ook (met andere toetsen) kunnen laten besturen, kunnen twee spelers op hetzelfde toetsenbord tegen elkaar spelen. De ene is de jager (vierkant) die de prooi (blokje) moet pakken.

Als uitgangspunt nemen we het eindresultaat van de vorige opdracht. De prooi moet te besturen zijn met de toetsen `w`, `a`, `d` en `s`. Voor de pijltoetsen gebruiken we `keyIsDown(DOWN_ARROW)`. Voor de `a` is dit we `keyIsDown(65)`. Maar hoe weet je nu dat 65 hoort bij de 'a'? De site <http://keycode.info> helpt je daar bij.

136. Open `H1O24.js` in jouw *editor* en bestudeer de code. Let in het bijzonder op de vier variabelen die bovenin gedeclareerd zijn. Deze moet je gebruiken voor deze opdracht.

137. Breid de code uit, zodat een tweede speler het blokje (prooi) kan besturen met `w`, `a`, `d` en `s`.

138. De prooi kan nu het canvas verlaten. Los dit probleem op.



FIGUUR 1.33

De prooi kan nu bewegen. In figuur 1.33 heeft de prooi zich naar linksboven verplaatst terwijl de jager zich rechtsonder bevindt. Toch kleurt de prooi felgroen, alsof hij geraakt wordt. Begrijp je waarom?

139. Pas de code aan, zodat de prooi alleen felgroen kleurt wanneer de jager hem ook echt raakt.

140. Onderaan is een functie `eindScherm` gemaakt. Zorg dat deze wordt aangeroepen op het moment dat het 'raak' is.

★ Opdracht 25 obfuscator III: raak het doel

141. Bekijk `OBF03`. Gebruik hierbij de pijltjestoetsen (links / rechts).

142. Open `H1O25.js` in jouw *editor* en lees de aanwijzingen in de code. Bekijk het huidige tussenresultaat in de *browser*. De bal weerkaatst boven en onder, maar verdwijnt rechts uit beeld.

143. Pas de code aan tot het resultaat van `OBF03`.



FIGUUR 1.34

1.7 herhalingen met een for-loop

In paragraaf 1.5 hebben we functies gebruikt voor het tekenen van een huis en een boom. Het schrijven van een aparte functie is vooral handig, wanneer je de functie vaker wilt gebruiken. In plaats van alle code voor het tekenen van een huis steeds opnieuw in te typen, kun je dan volstaan met het herhalen van de functie `tekenHuis` (figuur 1.35). Dit lukt nog wel voor vier huizen, maar wat nu als je een stad wilt tekenen? We kunnen de computer vragen om een aantal stappen te herhalen met een **for-loop** zoals in figuur 1.36. Deze code doet exact hetzelfde als de code uit figuur 1.35!

Qua vorm lijkt een for-loop op het maken van een voorwaarde met `if`. Eerst is er een gedeelte tussen `()` waarin staat voor welke situaties er iets moet gebeuren, gevolgd door `{ }`. Hiertussen zet je de code die moet worden uitgevoerd: *teken een huis en ga daarna een stukje opzij zodat het volgende huis op andere nieuwe plek komt.*

Bij een for-loop zet je tussen de haakjes `()` drie dingen:

- `var n = 1;`
Je declareert een variabele die als een teller gaat werken. Je geeft hem een naam (in dit geval `n` maar het hoeft niet per se één letter te zijn) en een **beginwaarde** (`1`) (initialisatie)
- `n <= 4;`
De teller gaat een aantal stapjes doorlopen tot een zekere **eindwaarde**. De code `n <= 4;` is een **voorwaarde**: de herhaling gaat door tot en met `n = 4`, ofwel: zolang aan de voorwaarde wordt voldaan.
- `n++;`
Dit betekent: hoog de variabele `n` steeds met 1 op. In de meeste gevallen wordt dit gebruikt, maar bijvoorbeeld `n--` of `n += 3` mag ook. Dit heet de **stapgrootte**.

```
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);
```

FIGUUR 1.35

```
for (var n = 1;n <= 4;n++) {  
    tekenHuis();  
    translate(200,0);  
}
```

FIGUUR 1.36

```
for (var n = 0;n < 4;n++) {  
    tekenHuis();  
    tekenBoom(n);  
    translate(200,0);  
}
```

FIGUUR 1.37

Met de code uit figuur 1.37 loopt de variabele `n` niet langs 1, 2, 3 tot 4, maar van 0 langs 1 en 2 tot 3, want de 4 doet niet mee vanwege het `<`-teken. Dat zijn nog steeds vier stapjes en dus worden er nog steeds vier huizen getekend! Het is onder programmeurs gebruiklijker om het tellen te beginnen bij 0 (dan bij 1).

In figuur 1.37 zie je nog een truc die we in de opdrachten zullen gaan gebruiken. De variabele `n` kun je in de herhaling gebruiken voor een berekening of, in dit geval, als attribuut voor de functie `tekenBoom(n);`

Opdracht 26 oefenen met vaste herhalingen

144. Open `H1O26.js` in jouw editor. Dit is de code van voorbeeld 8. Bekijk het resultaat in de browser.
145. Zet `//` voor regel 12 (`translate(125,0);`) zodat deze niet meer wordt uitgevoerd.
146. Pas de for-loop aan zodat er geteld wordt vanaf 0 tot (en dus niet tot en met!) 5. Bekijk het resultaat.



FIGUUR 1.38

De functie `tekenBoom(x)` heeft een parameter `x` waarvoor als waarde de waarde van `n` wordt ingevuld. Als `n` groter wordt, wordt de boom ook groter getekend (zie figuur 1.38).

147. Hoe breed en hoe hoog is de ellips waarmee de grootste boom is getekend?
148. Met een paar extra aanpassingen kun je figuur 1.39 nabouwen. Kijk goed wat er allemaal veranderd is en bouw de tekening na.



FIGUUR 1.39

Opdracht 27 JOS op herhaling

In deze opdracht gebruiken we een for-loop om JOS meerdere keren op het scherm te tekenen. Afhankelijk van het aantal keren dat we JOS tekenen wordt het canvas hiervoor opgedeeld in blokken van gelijke *breedte*.



FIGUUR 1.40

149. Open *H1O27.js* in jouw *editor*. Bekijk het resultaat in de browser. Wat gebeurt er als je op het pijltje naar rechts drukt?

150. Welke waarde heeft de variabele *breedte* op dit moment?

151. Zorg dat de waarde van *breedte* afhangt van de variabele *aantal*.

Voorbeeld: als *aantal* = 5, dan moet gelden dat *breedte* = $1000 / 5 = 200$

152. Hoewel we *aantal* nu kunnen verhogen, zien we nog steeds maar vier versies van JOS. Zorg dat dit afhankelijk wordt van de variabele *aantal*.

153. Zorg dat we *aantal* ook (met 1) kunnen verlagen als op het pijltje naar links wordt gedrukt.

154. Het is nu mogelijk dat het aantal gelijk wordt aan 0 of zelfs negatief wordt. Zorg dat *aantal* minimaal 1 blijft.

155. De functie *tekenJos* krijgt als argument 2 mee. Welke betekenis heeft de waarde 2 hier?

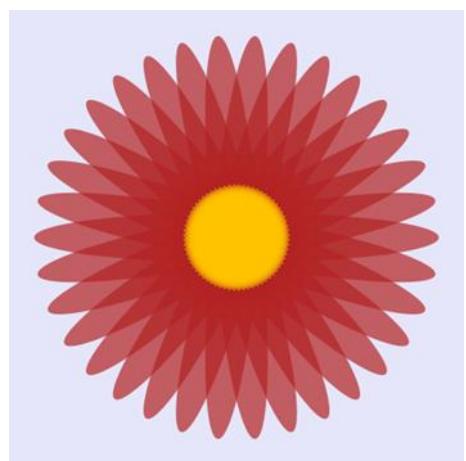
156. Wat verwacht je te zien als deze (2) wordt veranderd in (2 + n / 2)? Controleer je voorspelling.

★ Opdracht 28 draaien II

In opdracht 7 heb je kennis gemaakt met de functie *rotate* om dingen te laten draaien. We gaan deze functie gebruiken binnen een for-loop.

157. Open *H1O28.js* in jouw *editor* en bekijk het resultaat. Met behulp van regel 22 wordt nu één *blad* van een rode bloem getekend. Daarna wordt er eenmalig gedraaid m.b.v. regel 23.

158. Voeg een for-loop toe zodat regel 22 en 23 vaker (gebruik de variabele *aantal*) worden herhaald.



FIGUUR 1.41

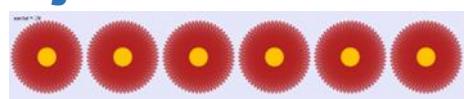
De gele binnenkant van de bloem in figuur 1.41 is gemaakt door een vierkant met zijdes van 75 pixels en een reeds in de code gegeven gele vulkleur te draaien op vergelijkbare manier als de rode bladeren.

159. Maak een aparte herhaling om de gele binnenkant te tekenen.

160. Pas de ellips en het vierkant aan naar eigen smaak en kies je eigen vulkleur. Hoe ziet jouw mooiste bloem eruit?

★ Opdracht 29 obfuscator IV: bloemenrij

De bloemenrij in figuur 1.42 is gemaakt door slim gebruik te maken van het eindresultaat van de vorige opdracht. Het volgende is gedaan:



FIGUUR 1.42

- Alle coderegels die gezamenlijk één bloem tekenen zijn samengevoegd in een functie *tekenBloem*.
- Aan deze functie is de functie *scale* toegevoegd, zodat een bloem kleiner kan worden getekend.
- Er is een nieuwe variabele voor het aantal bloemen gemaakt genaamd *Nbloemen*.
- Deze variabele is gebruikt in een for-loop waarin de functie *tekenBloem* wordt aangeroepen.

161. Bekijk *OBF04* die de code verbergt waarmee de tekening van de bloemenrij in figuur 1.42 is verkregen (met *aantal* = 29).

162. Open *H1O29.js* in jouw *editor*. Dit is het eindresultaat van de vorige opdracht.

Als je een programma schrijft is het verstandig dat je dit in stapjes doet, waarbij je steeds even tussentijds controleert of je nog op de goede weg bent, zodat je bij een fout snel weet in welke coderegels deze zit.

163. Pas de code aan volgens de gegeven stappen, zodat een bloemenrij verschijnt.

Opdracht 30 raster: een herhaling in een herhaling

Er zijn veel games en andere programma's, waar het handig is om te werken (en te denken) met een raster (denk ook: *Dammen* en *Schaken*). figuur 1.43 toont een raster van 9×9 vakjes. Het canvas is hier 450×450 dus één vakje of **cel** heeft een zijde van $450 / 9 = 50$.

Bij een raster (b.v. in Excel) spreek je horizontaal van een **rij** en verticaal van een **kolom**. De oranje gemarkeerde cel in figuur 1.43 bevindt zich in de 7^e kolom en de 4^e rij. Let op: omdat we bij 0 beginnen te tellen, hoort hier kolomnummer 6 en rijnummer 3 bij.

Dat lijkt onhandig, maar is het niet, want om een vierkant te tekenen moeten we aan de functie **rect** als parameters meegeven hoeveel we opzij (300) en omlaag (150) moeten gaan voordat we gaan tekenen. Omdat bij 0 beginnen te tellen, moeten we rekenen met 6 en 3.

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									
8									

$$\text{rij} = 3 \cdot 50 = 150$$

$$\text{kolom} = 6 \cdot 50 = 300$$

FIGUUR 1.43

164. Open *H1O30.js* in jouw *editor*. Bekijk het resultaat in de *browser*.

165. Bestudeer de code: er wordt hier één rij getekend met negen kolommen (van één cel).

Het raster in figuur 1.43 is een herhaling in een herhaling: met de eerste herhaling maken we de rij met negen cellen die je nu ziet. Het tekenen van die rij wordt vervolgens negen keer herhaald.

166. Breid de code uit, zodat je een raster krijgt. Gebruik een variabele **rij** op dezelfde manier als nu de variabele **kolom** in de for-loop is gebruikt.

167. Gebruik een **if** binnen de herhaling om dezelfde cel als in figuur 1.43 oranje (*orange*) te kleuren.



Opdracht 31 blokpatronen

In figuur 1.44 zie je een blokpatroon dat is verkregen op basis van de code uit de vorige opdracht met de volgende aanpassingen:

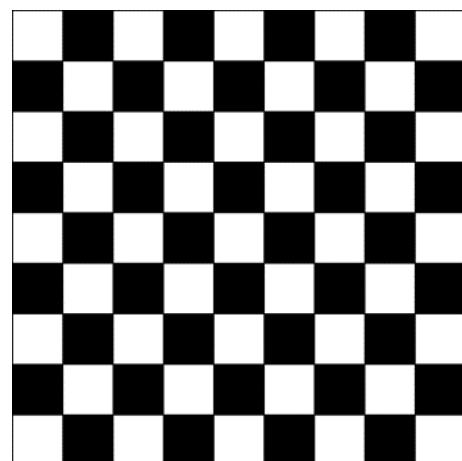
- Er is een variabele **kleur** gemaakt voor de vulkleur van de cellen
- Er is een **if - else** aan toegevoegd volgens de redenatie: *Als de huidige kleur wit is, dan moet de kleur voor de eerstvolgende cel zwart zijn en andersom*

168. Open *H1O31.js* in jouw *editor* en bekijk het resultaat.

169. Pas de code aan volgens bovenstaande aanpassingen, zodat het patroon uit figuur 1.44 verschijnt.

Een dambord is niet 9×9 maar 10×10 met dezelfde structuur.

170. Pas het canvas aan tot 501×501 en maak een dambord.



FIGUUR 1.44

Opdracht 32 de random-functie

Met de functie **random** kun je de computer een willekeurig getal laten kiezen. Dit getal kun je vervolgens gebruiken om iets op een willekeurige plaats te tekenen of bijvoorbeeld een willekeurige kleur te geven.

171. Open *H1O32.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Hoe wordt de celkleur bepaald?

172. Haal de **//** in regel 14 weg en bekijk het resultaat.

Met de functie **random** kiest computer nu zelf een getal tussen 0 en 255 voor de variabele **R**.

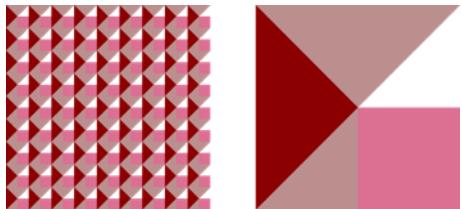
173. Zorg dat de computer ook voor de variabelen **G** en **B** een getal tussen 0 en 255 kiest.

174. Voorspel wat je ziet als je de regel voor de vulkleur verandert in **fill(R,R,R);**.

175. Pas de code aan zodat alle cellen in één rij dezelfde willekeurige kleur krijgen.

Opdracht 33 tegelpatronen

In veel culturen en in het bijzonder de islamitische cultuur zijn decoraties ontstaan die gebaseerd zijn op de herhaling van elementaire afbeeldingen of **tegels**. We kunnen het raster uit de vorige opdrachten inzetten om zelf een mooi tegelpatroon te ontwerpen.



FIGUUR 1.45

176. Open *H1O33.js* in jouw *editor*. Je ziet nu de code waarmee het patroon in figuur 1.45 is getekend.
177. Pas kleuren en vormen aan en ontwerp je eigen tegel.



Opdracht 34 obfuscator V: random ringen

De 100 ringen in figuur 1.46 zijn op een willekeurige plek getekend.

178. Bekijk *OBF05* die het resultaat toont maar de code verbergt.

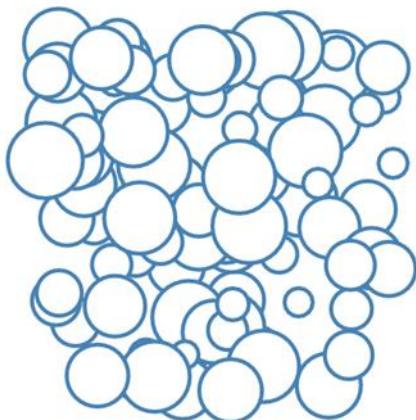
Het programma is gemaakt op basis van de volgende keuzes:

- Het middelpunt van de cirkels is zo gekozen, dat zowel voor de *x*-waarde als de *y*-waarde geldt dat dit een willekeurig gekozen waarde is tussen de 50 en 400.
- De diameter van de cirkels is een willekeurige waarde tussen de 25 en 75

179. Open *H1O34.js* in jouw *editor*. Hierin is al een begin gemaakt.

180. Breid de code uit zodat deze aan bovenstaande eisen voldoet.

181. Zet `//` voor `noLoop()`; zodat er steeds 100 nieuwe cirkels verschijnen.



FIGUUR 1.46

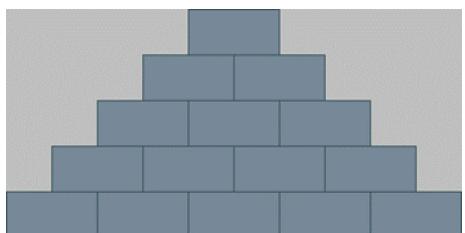


Opdracht 35 piramide

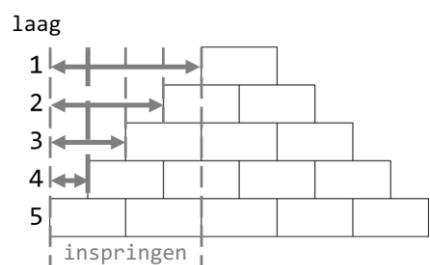
De *stenen* in figuur 1.47 zijn gestapeld in de vorm van een piramide. Er zit een duidelijke vorm van herhaling in het patroon, maar hoe leg je die uit aan een computer?

Een logische keus is om de piramide op te delen in lagen en die te nummeren. Aan de hand van figuur 1.48 merken we het volgende op:

- Het aantal stenen in een laag is gelijk aan het nummer van de laag
- Bij alle lagen behalve de onderste moet je eerst een stukje inspringen voordat je de eerste steen kunt tekenen.
- Het inspringen gaat met stapjes van een halve steenbreedte
- Een steen is twee keer zo breed als hij hoog is



FIGUUR 1.47



FIGUUR 1.48

182. Open *H1O35.js* in jouw *editor*. Bekijk het resultaat.

183. Vervang `tekenRij(4)` door een herhaling die voor elke naam opnieuw `tekenRij(1aag)` aanroept voor elk laagnummer en vervolgens omlaag gaat naar de volgende laag.

In de functie `tekenRij(aantalStenen)` is een variabele `inspringen`. Het is de bedoeling dat de waarde van deze variabele gelijk is aan het aantal pixels dat je eerst naar rechts moet voor je een rij stenen tekent.

184. Pas de regel met deze variabele aan, zodat wordt berekend hoeveel pixels er naar rechts moet worden ingesprongen voordat de rij met stenen wordt getekend.
185. Verander de waarde van `aantalLagen` naar 10. Krijg je een keurige piramide met tien lagen?



1.8 VERDIEPING: recursie

In de vorige opdracht heb je het tekenen van een piramide geprogrammeerd. In figuur 1.49 A staan de coderegels die we daarvoor hebben gebruikt. De code in figuur 1.50 B levert dezelfde piramide op.

```
function draw() {  
    for (var laag=1;laag<=aantalLagen;laag++) {  
        tekenRij(laag);  
        translate(0,hoogte);  
    }  
}  
  
function tekenRij(aantalStenen) {  
    inspringen =  
        (aantalLagen-aantalStenen)*0.5*breedte;  
    push();  
    translate(inspringen,0);  
    for (var steen = 0;steen < 4;steen++) {  
        rect(breedte*steen,0,breedte,hoogte);  
    }  
    pop();  
}
```

FIGUUR 1.49 A

Hoewel beide codes hetzelfde eindresultaat opleveren, is de gebruikte programmeertechniek totaal anders.

De variant rechts gebruikt **recursie**. Recursie is een programmeertechniek die je gebruikt in situaties waarbij een opdracht logisch opgedeeld kan worden in meerdere opdrachten die hetzelfde zijn qua vorm, maar dan eenvoudiger.

Kenmerkend voor recursie is dat er een functie is die zichzelf aanroeft. De functie `tekenPiramide` in figuur 1. bevattet zelf weer de coderegel `tekenPiramide`. Het idee daarachter is als volgt:

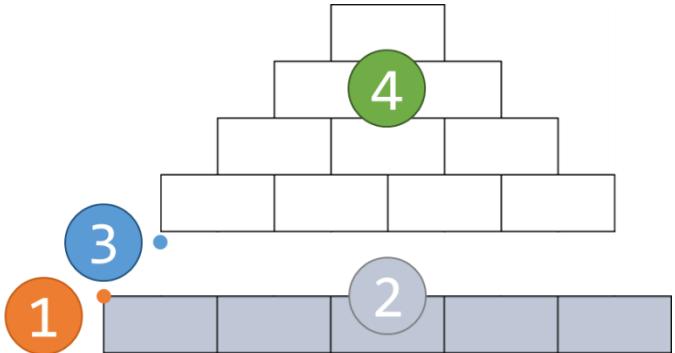
Als je een piramide van $n = 5$ lagen van onderaf begint te tekenen, dan heb je na het tekenen van de onderste laag stenen daarna nog 4 lagen te gaan. Die 4 lagen vormen zelf ook een piramide! Als we na het tekenen van één laag de functie `tekenPiramide` voor $n = 4$ uitvoeren, krijgen we uiteindelijk een piramide van 5 lagen. In stappen (zie figuur 1.50):

- Verplaats (`translate`) de tekenpositie naar het begin van de eerste laag (1)
- Teken de eerste laag stenen van in totaal $n = 5$ (2)
- Verplaats de tekenpositie naar het beginpunt van de bovenliggende laag (3)
- Geef de opdracht om vanaf dat punt een nieuwe piramide te tekenen voor $n - 1 = 4$ (4)
- Herhaal dit, zolang de piramide nog niet klaar is, dus zolang $n > 0$.

In figuur 1. zie je deze stappen vertaald naar programmeercode. Dit is de code van voorbeeld 9. In deze paragraaf gaan we recursie inzetten voor het oplossen van problemen.

```
function draw() {  
    translate(0,height-hoogte);  
    tekenPiramide(aantalLagen);  
}  
  
function tekenPiramide(n) {  
    if (n>0) {  
        for (var nr = 0;nr < n;nr++) {  
            rect(nr*breedte,0,breedte,hoogte);  
        }  
        translate(breedte/2,-hoogte);  
        n--;  
        tekenPiramide(n);  
    }  
}
```

FIGUUR 1.50 B



FIGUUR 1.50

★ Opdracht 36 piramide

In opdracht 35 heb je een piramide geprogrammeerd. In *voorbeeld 9* wordt dezelfde piramide geprogrammeerd, maar nu met recursie.

186. Open *H1O36.js* in jouw *editor*. Bekijk het resultaat in de browser.

187. Zorg dat een piramide met 10 lagen wordt getekend.

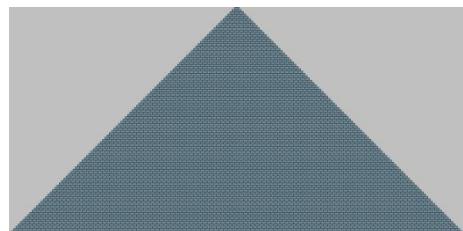
Hoewel de functie `tekenPiramide` zichzelf aanroeft, gaat het programma niet oneindig lang door. Dit komt omdat de programmeur een voorwaarde heeft ingebouwd.

188. Wanneer stopt de functie met de uitvoer? Ofwel: wat is de **stopconditie**?

Als we het aantal lagen groter maken, neemt de grootte van het canvas ook toe. Het canvas is op dit moment $10 \times 90 = 900$ pixels breed en 450 pixels hoog. We willen bereiken dat het canvas altijd een grootte van 900×450 pixels heeft en dat het programma zelf uitrekent hoe breed en hoog de stenen dan kunnen worden.

189. Pas de code aan, zodat bovenstaand doel wordt bereikt.

190. Maak een piramide met 100 lagen.



FIGUUR 1.51

★ Opdracht 37 Droste-effect

In figuur 1.52 zie je een wereldberoemd cacaoblik van de Nederlandse firma Droste. Op het cacaoblik staat een serveerster die een dienblad vasthoudt met daarop een cacaoblik met daarop een verpleegster die een dienblad vasthoudt met een cacaoblik met daarop... etc. Deze eindeloze herhaling wordt, ook in het buitenland (!), het **Droste-effect** genoemd. Het is een voorbeeld van recursie.

191. Open *H1O37.js* in jouw *editor*. Bekijk het resultaat in de browser. Je ziet een kamer met een deur. Aan de wand hangt een groot zwart schilderij.

We willen dat op het schilderij het beeld van de kamer wordt herhaald. Dit beeld wordt gemaakt met de functie `tekenKamer`. Als we hier recursie willen toepassen, moeten we zorgen dat de functie `tekenKamer` zichzelf aanroeft.

192. Voeg aan het eind van de functie de volgende regel toe:
`tekenKamer(0.5);`

Als het goed is zie je nu het Droste-effect.

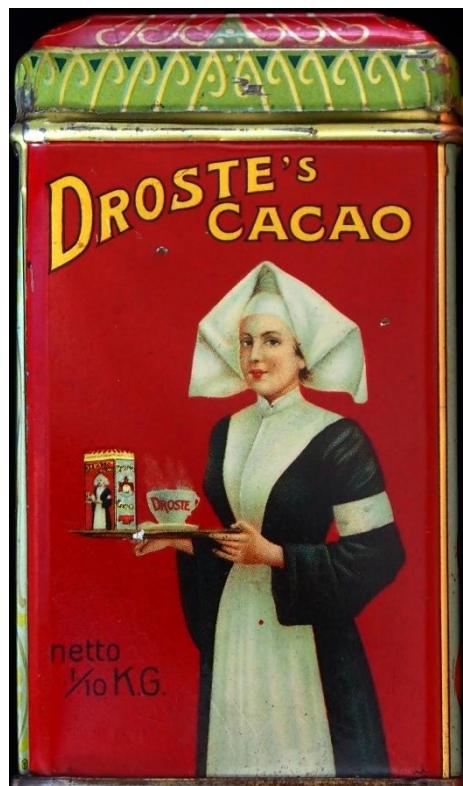
193. Gaat dit programma oneindig door? Of is er een stopconditie?
Zo ja, onder welke voorwaarde stopt dit programma?

We kunnen nu onderdelen toevoegen aan de kamer en kleuren aanpassen, zoals het voorbeeld in figuur 1.53.

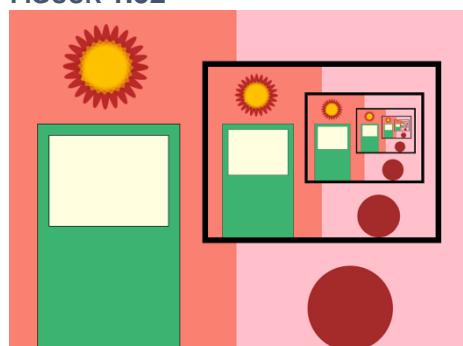
194. Voeg een grote bal toe onder het schilderij. Bekijk het resultaat.

195. Pas de kleuren van de kamer aan, zodat ze passen bij jouw smaak.

196. Voeg minimaal één ander object toe aan de ruimte. In figuur 1.53 hebben wij gekozen voor de bloem die we eerder dit hoofdstuk hebt getekend, maar je kunt ook kiezen voor iets geheel nieuws.



FIGUUR 1.52



FIGUUR 1.53

★ Opdracht 38 Cantorverzameling

Bij de vorige opdracht hebben we de functie `scale` gebruikt om het Droste-effect te creëren. Dat is een handige truc, maar zeker niet noodzakelijk. In deze opdracht kijken we naar de **Cantorverzameling**.

De tekening in figuur 1.54 is gemaakt door een functie `cantor` te schrijven. Deze functie voert de volgende stappen uit:

- Teken een rechthoek met een zekere lengte (breedte canvas)
- Verplaats een stukje naar onderen
- Teken nu twee keer een rechthoek met $1/3$ deel van de vorige lengte, namelijk éénmaal vooraan en eenmaal tot aan het eind, ofwel vanaf $2/3$ deel van de rechthoek erboven
- Herhaal bovenstaande stappen



FIGUUR 1.54

197. Open `H1O38.js` in jouw *editor*. Bestudeer de functie `cantor`. Bekijk het resultaat in de *browser*.

Op dit moment zien we maar één rechthoek en, voor het gemak, de lengte van die rechthoek. We kunnen van de functie `cantor` een recursieve functie maken door te zorgen dat de functie zichzelf aanroept.

198. Haal de `//` weg in regel 23 en bekijk het resultaat.

199. Kopieer regel 23 naar regel 24 en pas de parameter `x` aan zodanig dat de tweede rechthoek op $2/3$ deel van de vorige rechthoek eveneens verschijnt.

De code bevat een stopconditie. Als de lengte kleiner wordt dan 1, dan stopt de herhaling.

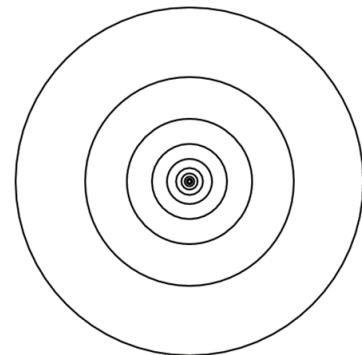
200. Verwijder de stopconditie uit de code, zodat het programma steeds doorgaat.

201. Verklaar dat de herhalingen aan de linkerkant nog te zien zijn, maar aan de rechterkant niet.

★ Opdracht 39 fractal van cirkels

Als je inzoomt op de Cantorverzameling in figuur 1.54, dan zie je eigenlijk hetzelfde patroon als in de huidige uitgezoomde versie. Het maakt niet uit hoe ver je inzoomt: je ziet steeds eenzelfde patroon. Dat is een kenmerk van een **fractal**. Fractals worden doorgaans getekend met behulp van recursie.

De vorm in figuur 1.55 is gemaakt met behulp van recursie.



FIGUUR 1.55

Net als bij de Cantorverzameling kunnen we onze recursieve functie interessanter maken door te zorgen dat de functie zichzelf meer dan eens aanroept.

204. Maak een schets op papier van wat je verwacht te zien als regel 24 wordt aangepast tot:

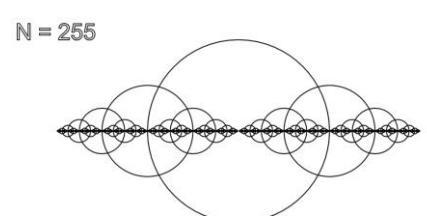
`tekenCirkel(x + 0.5*D,y,D*0.5);`

205. Controleer je verwachting.

206. Voeg in regel 25 een vergelijkbare regel toe, zodat, vanuit het midden naar links toe, hetzelfde patroon ontstaat als naar rechts.

207. In de code hierboven staat `D*0.5`. Hoeveel cirkels krijg je als je de `0.5` in `0.6` verandert voor beide regels met `tekenCirkel`?

208. En hoeveel zijn het er voor `0.75`?

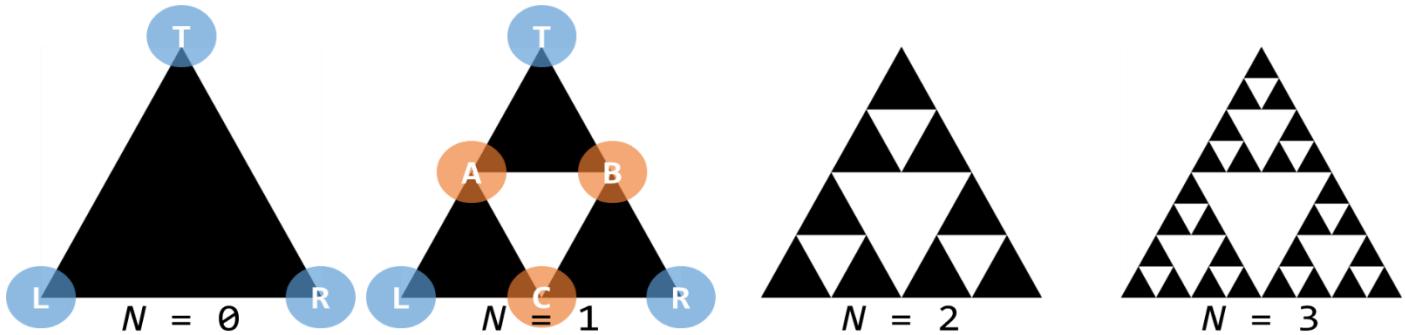


FIGUUR 1.56



Opdracht 40 Sierpinski-driehoek

Met recursie kunnen we met slechts enkele regels code complexe figuren maken. Een voorbeeld hiervan is de Sierpinski-driehoek. In figuur 1.57 zie je zijn eerste ontwikkelstappen. Herken je het recursieve patroon?



FIGUUR 1.57

Als uitgangspunt nemen we een eerste driehoek ($N = 0$) bestaande uit drie punten T (top), L (links) en R (rechts). Voor $N = 1$ zien we het patroon van een **Sierpinski-driehoek**: er zijn nu drie driehoeken, getekend vanaf de punten T , L en R en de middelpunten van de zijden A , B en C . Als we voor elk van de drie driehoeken deze stap herhalen krijgen we negen driehoeken ($3 \cdot 3 = 9$; $N = 2$) en daarna zeventig driehoeken ($3 \cdot 3 \cdot 3 = 27$; $N = 3$) enzovoorts. Dit patroon is zeer geschikt om met behulp van recursie te programmeren!

We hebben de Sierpinski-driehoek al deels voor je geprogrammeerd. In de code zijn voor de in de figuur gemaakte punten T , L en R al variabelen gemaakt voor hun x- en y-coördinaten. Datzelfde geldt voor de punten A en B .

209. Open `H1O40.js` in jouw *editor*. Merk op dat in regel 1 de waarde van N kan worden ingesteld.
210. Bekijk de regels die de x- en y-coördinaten van de punten T , L en R beschrijven. Begrijp je ze?
211. In de functie `sierpinski` zijn de punten A en B beschreven op basis van T , L en R . Voor het punt C kloppen de coderegels nog niet. Pas de regels voor Cx en Cy aan.
212. Bekijk het resultaat in de browser. Als je het goed gedaan hebt zie je de driehoek TAB.
213. Verhoog N naar 2 en vervolgens naar 3. Wat zie je?

De driehoek TAB zie je, omdat de functie `sierpinski` zich in regel 45 zelf aanroept voor de punten T , A en B . Let op: als parameter wordt nu $n - 1$ meegegeven. Dit zorgt voor $n = 0$, waardoor de functie naar het `else`-deel gaat. Daar wordt (pas) de driehoek getekend.

Om het volledige patroon (voor $N = 1$ in figuur 1.57) te krijgen moeten we de functie `sierpinski` zichzelf ook laten aanroepen voor de driehoeken ALC en BCR.

214. Voeg twee coderegels toe om dit te bereiken (in de regels 46 en 47).
215. Zet N weer terug naar 1. Voer de code uit en controleer of het resultaat klopt met figuur 1.57.
216. Verhoog N naar 2 en vervolgens naar 3. Komt dit overeen met figuur 1.57?
217. De waarde van N is beperkt tot een maximum van 10. Hoeveel driehoeken zie je dan op het scherm? (Natuurlijk ga je niet tellen, maar rekenen!)

De functie `sierpinski` roept zichzelf nu drie keer aan. Overzie jij in welke volgorde de driehoeken nu worden getekend?

218. Maak een schets op papier voor $N = 3$ en geef hierin met een volgnummer aan in welke volgorde de driehoeken volgens jou worden getekend.
219. Zorg dat N de waarde 3 heeft en haal de `//` weg voor regel 50, zodat de driehoeken in een steeds lichtere grijstint worden getekend. Klopt jouw voorspelling bij de vorige vraag?
220. Maak een functie `kiesKleur` die een willekeurige (random) kleur instelt.
221. Roep de functie `kiesKleur` aan, vlak voor de regel met `triangle`.
(De grijstinten komen hiermee te vervallen).
222. Verander N (niet te hoog!) tot het eindresultaat je bevult.

H2 OBJECTEN

2.1 Inleiding (camelCase)

In hoofdstuk 1 heb je kennis gemaakt met het programmeren in Javascript en P5 (Processing). Daarbij lag de nadruk op het toepassen van een aantal basisprincipes zoals variabelen, functies, herhalingen en voorwaarden die je waarschijnlijk al bij eerdere programmeerlessen had gezien. Misschien is het je opgevallen dat we in de code soms hoofdletters gebruiken. Vanaf nu doen we dit zoals hieronder:

```
var naam = 'JOS';           // een variabele (met tekst) bestaande uit één woord
var aantalBomen = 2;         // een variabele (nummeriek) bestaande uit twee woorden
noFill();                   // een bestaande P5-functie
createCanvas(50, 20);        // een bestaande P5-functie
function tekenStenen() { }  // een zelfgemaakte functie
class Vijand() { }         // een klasse van een object
voldemort = new Vijand();   // een instantie van de klasse Vijand
```

Deze stijl van notatie heet (*lower*) *camel case* (*camelCase*). Belangrijkste kenmerk is dat bij samengestelde woorden of zinnen (zoals *noFill* of *aantalBomen*) de afzonderlijke elementen beginnen met een hoofdletter, behalve de eerste. Deze manier van noteren ken je vast van de *iPhone*. Behalve *lower* bestaat er ook *Upper CamelCase* die je zonder dat je het wist al kent van *PlayStation* en *SpongeBob*.

De laatste twee voorbeelden met *Vijand* wijken af van de afspraak, (want het begint met een hoofdletter). Deze notatie gaan we gebruiken voor het maken van (een klasse van) **objecten**. In dit hoofdstuk leer je wat objecten zijn. We beginnen met een paar bijzondere objecten: afbeeldingen en lijsten.

2.2 afbeeldingen

In figuur 2.1 zie je een beeld uit *voorbeeld 10* waarin twee **afbeeldingen** als **sprites** worden gebruikt:

een rij bomen als achtergrond en een kever die door de lucht vliegt.

Die plaatjes laden we vooraf met de functie *preload*:

```
function preload() {
  bomen = loadImage("images/bomen.jpg");
  kever = loadImage("images/sprites/kever.png");
}
```

De afbeeldingen staan in aparte bestanden in jouw werkomgeving in de map *images*. Tussen de "" geef je aan waar het bestand staat. Het bestand *kever.png* is bijzonder, omdat hij een transparante achtergrond heeft. Hierdoor lijkt het alsof kever door de lucht vliegt. Net als een getal of een tekst, kun je een plaatje opslaan in een variabele (zoals hier *bomen* en *kever*). In het vervolg van je programma kun je verwijzen naar deze variabelen. P5 weet dan zelf dat het om een variabele van het type afbeelding gaat. Een afbeelding is een voorbeeld van een (bijzonder) **object**.

Kenmerkend voor een object is dat er **eigenschappen** aan gekoppeld zijn. Zo heeft een afbeelding een breedte en hoogte in pixels. Door eerst de naam van het object te noemen en met een punt aan de eigenschap te verbinden, kun je eigenschappen gebruiken of opvragen: *kever.width* en *kever.height*.

Voorbeeld 10 laat twee manieren zien om de afbeeldingen ook echt in het canvas te tonen:

background(bomen); en *image(kever, 40, 60);*. In de volgende opdrachten ga je hier mee werken. Bovendien leer je een aantal extra trucs die je met afbeeldingen kunt uithalen.



Het keverplaatje heeft breedte 128 px en hoogte 128 px.

FIGUUR 2.1



Opdracht 1 afbeeldingen laden en tonen

1. Open *H2001.js* in jouw *editor*. Dit is de code van *voorbeeld 10* met enkele toevoegingen. Bekijk het resultaat in de *browser*.
2. Pas regel 20 aan naar `background(kater)`; zodat we als achtergrond het beeld uit figuur 2.2 krijgen.
3. In de `preload` is de bijbehorende jpg-afbeelding geladen. De bestandsnaam is de naam van de kater. Hoe heet hij?
4. We kunnen dezelfde afbeelding ook laden met regel 21. Verwijder de `//` voor `image(kater, 0, 0)`; en bekijk het resultaat. Probeer uit te leggen waarom we niet hetzelfde zien.
5. Verander regel 20 in `background('grey')`; en verander de getallen `0, 0` in regel 21. Welke functies hebben ze?
6. Pas regel 21 aan tot `image(kater, 25, 25, 400, 400)`. Wat zie je? Wat is de betekenis van `400, 400`?
7. Haal de `//` weg bij regel 24 zodat de kever uit *voorbeeld 10* weer te zien is. Verklaar de beweging die de kever maakt.
8. Zorg dat kever met grootte 30×30 (pixels) wordt afgebeeld.

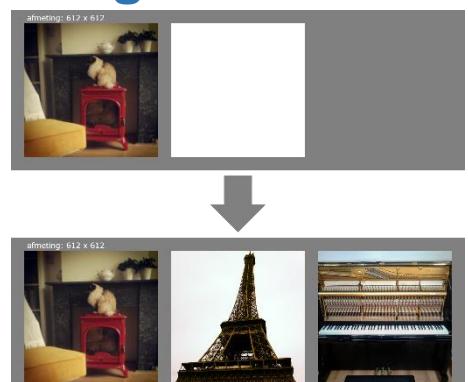


FIGUUR 2.2

Opdracht 2 meer oefenen met afbeeldingen

9. Open *H2002.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet het bovenste deel van figuur 2.3.

In de map *images* heb je behalve het bestand *brieck.jpg* ook afbeeldingen van de Eiffeltoren en een piano.



FIGUUR 2.3

10. Voeg de afbeelding van de toren toe op de wit gemarkeerde plek. Laad de afbeelding met `loadImage` in `preload`.
11. Voeg ook de foto van de piano toe. Zorg dat de onderlinge afstand tussen de foto's steeds hetzelfde is.

Boven de foto van de kater staan de afmetingen van de foto in pixels. De breedte en de hoogte zijn eigenschappen van het object *kater*. Ze worden opgevraagd met *kater.width* en *kater.height*.

12. Voeg vergelijkbare tekst toe voor de toren en de piano. Welke afbeelding heeft de meeste pixels?

Opdracht 3 bewegende foto I

13. Open *H2003.js* in jouw *editor* en je *browser*.
14. Wat is de beginwaarde van de variabele *strandX*?

We gaan nu *dezelfde foto* (figuur 2.4) twee keer achter elkaar tonen, zodanig dat de foto's precies op elkaar aansluiten. In regel 16 staat de bijbehorende coderegel:

`image(strand, strandX + strand.width, 0)`



FIGUUR 2.4

15. Wat is de beginwaarde van *strandX + strand.width*?
Ofwel: waar wordt deze foto in eerste instantie getekend?
16. Haal de `//` weg in regel 16 en bekijk het resultaat.
Zie jij waar de foto's op elkaar aansluiten?

We hebben nu een bewegende achtergrond, maar na korte tijd zijn de foto's allebei uit beeld. Moeten we nu een derde foto toevoegen?

Nee! Er zijn nooit meer dan twee foto's tegelijkertijd in beeld. Als de foto's zich over één fotobreedte (`strand.width`; dit is hier ook de breedte van het canvas!) hebben verplaatst, kunnen we de beginsituatie weer herstellen door *strandX* terug te zetten naar de beginwaarde (0).

17. Voer deze opdracht uit. HINT: gebruik een if-constructie om dit voor elkaar te krijgen.

★ Opdracht 4 obfuscator VI: bewegende foto II

18. Bekijk *OBF06*. Deze toont de bewegende foto van de vorige opdracht. Als je de pijl naar rechts indrukt, verandert de bewegingsrichting van de foto.
19. Open *H2004.js* in jouw *editor*. Bekijk het resultaat in de *browser*.

Als je nu de pijl naar rechts indrukt, zie je dat het mis gaat als *strandX* groter is dan 0 (zie figuur 2.5) of kleiner dan -600.

20. Breid de code uit zodat jouw resultaat hetzelfde is als *OBF06*.

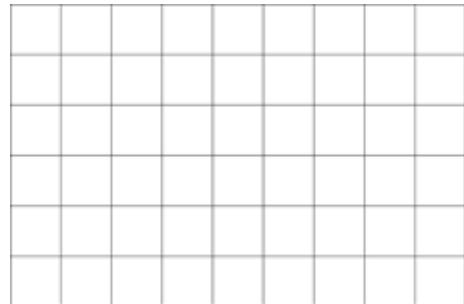


FIGUUR 2.5

Opdracht 5 overloper I: intro

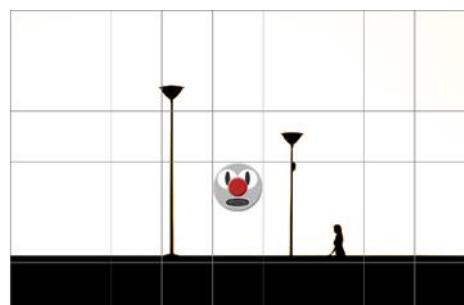
In dit hoofdstuk gaan we in stappen een spel maken dat *overloper* heet. Bij elke stap vertellen we je meer over de inhoud en de regels van het spel. Kortgezegd moet de speler proberen vanaf de linkerkant van het scherm de overkant te bereiken. Hierbij moet hij obstakels ontwijken. Als *game character* gebruiken we Jos: onze *Javascript Object Sprite* die je nog kent uit Hoofdstuk 1.

In deze opdracht maken we het speelveld en een raster (figuur 2.6) dat ons helpt bij de ontwikkeling van het spel.



FIGUUR 2.6

21. Open *H2005.js* in jouw *editor*. Bekijk het resultaat.
22. Het raster bestaat uit cellen (vierkantjes) met een lengte die hier *celGrootte* heet. In welke regel wordt deze lengte berekent? Wat is de afmeting van de cellen (in pixels)?
23. De functie *tekenRaster()* tekent op dit moment maar één cel. Maak een dubbele herhaling met een for-loop om het volledige raster te tekenen. Volg de instructies in het bestand of kijk terug naar de opdracht in hoofdstuk 1 waar je dit hebt geleerd.
24. In de *preload* is een afbeeldingsobject brug gemaakt. Voeg deze afbeelding in als achtergrond. Let op: zorg dat het raster over de foto heen wordt getekend.
25. Voeg de sprite *spriteJos* in zodat je het eindresultaat in figuur 2.7 krijgt. Gebruik de variabelen *xJos* en *yJos* om Jos op de juiste plek te krijgen.



FIGUUR 2.7

Opdracht 6 overloper II: interactie met een sprite

26. Open *H2006.js* in jouw *editor*. Bekijk het resultaat. Druk hierbij op de pijl naar rechts. Hoe groot zijn de stappen die Jos maakt?
27. Breid de besturing uit, zodat Jos ook naar links en naar boven en onder kan.

Als Jos aan de linker- of rechterkant van het canvas is, kan hij niet verder. Dit is gedaan met de regel:
xJos = constrain(xJos, 0, width - celGrootte);

28. Leg uit waarom de functie *constrain* gebruik maakt van het attribuut *width - celGrootte*.
29. Zorg dat Jos ook boven en onder op het speelveld blijft.

Van sommige stukjes code kun je voorspellen wat ze doen, ook al bevatten ze een functie die je nog niet eerder hebt gezien.

30. Probeer de code in figuur 2.8 te lezen. Voorspel wat het resultaat zal zijn van deze code.
31. Kopieer de code in figuur 2.8 naar het eind van de *draw*. Wat gebeurt er? Onder welke voorwaarden / in welke situatie?
32. Probeer nu ook eens: *spriteJos.filter(ERODE)*;

```
if (xJos == 6*celGrootte  
  && yJos == 4*celGrootte) {  
    spriteJos.resize(50,50);  
}
```

FIGUUR 2.8

2.3 Lijsten: arrays

In de vorige paragraaf hebben we leren werken met afbeeldingen. Een afbeelding is een bijzonder voorbeeld van een object. Kenmerkend voor objecten is dat ze eigenschappen hebben zoals een breedte (`foto.width`) en dat je ze kunt vragen om iets te doen, zoals van grootte veranderen (`foto.resize(50,50)`). Zie figuur 2.9.

Ook een lijst of **array** is een object. Het biedt de mogelijkheid om een reeks van gegevens die bij elkaar horen op te slaan onder één naam.

Dat zoiets handig kan zijn, laat de code in figuur 2.10 zien. Als je alle namen van je klasgenoten (of spelers van je game!) wilt opslaan, wordt het wel erg onhandig, als je die namen één voor één in aparte variabelen moet opslaan. De variabele `klas` is hier een array: een lijst met alle namen op onder één noemer.

Een array is een reeds bestaande soort object (of klasse). Een nieuwe lijst maak je met `new Array`. Alle elementen van de lijst krijgen een volgnummer. We beginnen weer te tellen bij 0, dus let goed op: het tweede element ("Alice" in figuur 2.10) heeft volgnummer 1, want bij "Bob" hoort volgnummer 0.

Omdat een array een object is, heeft het net als een afbeelding eigenschappen en kun je het object vragen om iets te doen (figuur 2.11).

Om de inhoud van het derde element (met volgnummer 2) op te vragen uit de array `klas` gebruik je `klas[2]`; en met `klas.length` krijg je het aantal elementen in de lijst terug.

Er zijn heel veel voorgeprogrammeerde handelingen die je aan een array kunt vragen. De belangrijkste voor ons staan in figuur 2.11:

- Met `push()` kun je iets toevoegen aan je lijst. Tussen haakjes vul je het element in dat je wilt toevoegen aan je lijst. Dit element komt standaard achteraan in de lijst te staan.
- Met `pop()` verwijder je het laatste element.
- Met `shift()` verwijder je juist het eerste element.
- Met `sort()` sorteert de array (alfabetisch / numeriek).

Omdat de elementen in een array een volgnummer mee krijgen, kun je ze met een for-loop (zie § 1.7) één voor één uitlezen en verwerken.

Door `klas.length` te gebruiken, bepaalt de computer zelf hoe vaak de loop moet worden herhaald:

```
for (var n = 0; n < klas.length; n++) {  
    text(klas[n], 65, 40*(n + 1));  
}
```

Bovenstaande code levert het resultaat in figuur 2.12. De teller `n` krijgt achtereenvolgens de waarde 0 t/m 3 (dus tot 4), want er wordt geteld *tot* (en dus niet *en met*) `klas.length = 4`. De namen staan 65 pixels uit de kantlijn. In dit geval komen de namen onder elkaar, omdat voor de verticale positie opnieuw de teller `n` is gebruikt. Omdat `n` steeds groter wordt, komen de namen steeds verder naar beneden.

```
// eigenschappen  
breedte =  
foto.width;  
hoogte =  
foto.height;
```

```
// handelingen  
foto.resize(50,50);
```

FIGUUR 2.9

```
var naam1 = "Bob";  
var naam2 = "Alice";  
var naam3 = "Eve";  
  
// alle namen in één lijst:  
var klas = new Array("Bob", "Alice", "Eve");
```

FIGUUR 2.10

```
// eigenschappen  
element3 = klas[2];  
  
aantalLeerlingen =  
klas.length;
```

```
// handelingen  
klas.push("Trent");  
klas.pop();  
klas.shift();  
klas.sort();
```

FIGUUR 2.11



FIGUUR 2.12



Opdracht 7 oefenen met arrays

33. Open *H2007.js* in jouw *editor*. Dit is de code van *voorbeeld 11* met enkele aanpassingen. Bekijk het resultaat in de *browser*. De array *vierkanten* bevat de lengte van een zijde van drie vierkanten.
34. Gebruik de functie `push()` om in regel 12 jouw eigen naam aan de array *namen* toe te voegen.
35. Voeg in regel 13 een coderegel toe, zodat de array *namen* alfabetisch wordt gesorteerd. Zie theorie.
36. Op regel 18 begint een for-loop. Wat is de grootste waarde die *teller* zal aannemen?
37. Met regel 21 wordt de omtrek van de vierkanten berekend en getoond. Regel 22 is voor de oppervlakte. Deze regel is nog niet af. Vul de regel aan, zodat de waarden overeenkomen met figuur 2.13.
38. Voeg in regel 14 een coderegel toe om het eerste element uit de array *vierkanten* te verwijderen. Zie theorie.

23	■	omtrek = 92	opp = 529
18	■	omtrek = 72	opp = 324
11	■	omtrek = 44	opp = 121
30	■	omtrek = 120	opp = 900

FIGUUR 2.13

Opdracht 8 huizenrij

39. Open *H2008.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet het bovenste deel van figuur 2.14.

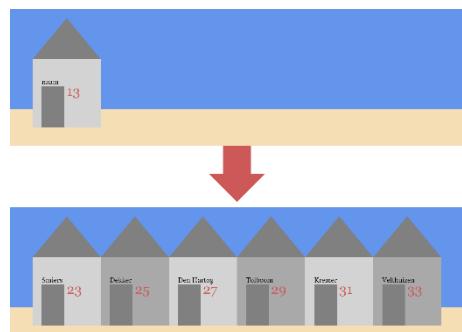
Het is de bedoeling dat we het onderste deel uit figuur 2.14 gaan programmeren. We zien zes huizen. Alle huizen hebben een nummer en een eigenaar. De bijbehorende gegevens staan in de twee arrays *huisNummers* en *huisEigenaren* (zie bovenaan in de code).

40. Maak een for-loop die de huidige regels 18 en 19 herhaalt. Gebruik de lengte van de array *huisNummers* om het juiste aantal herhalingen te krijgen.
41. Alle huizen hebben nummer 13, door het tweede argument van `tekenHuis(kleur, 13)`; Vervang het argument **13** door een stukje code, zodanig dat alle huizen een huisnummer uit de array *huisNummers* krijgen.
42. Voer de volgende stappen uit om de namen van de huiseigenaren op de huizen te krijgen:
 - Voeg een extra parameter naam toe aan de functie `tekenHuis`.
 - Pas de regel `text("naam", 20, 165)`; aan, zodat deze gebruik maakt van deze parameter
 - Pas de for-loop aan, zodat er bij het aanroepen van de functie `tekenHuis` gebruik gemaakt wordt van de array *huisEigenaren*.

De huizen in figuur 2.14 hebben afwisselend de kleur *lightgray* en *darkgrey*. Dat is geprogrammeerd door in de for-loop, na het tekenen van een huis, toe te voegen:

Als de waarde van de variabele kleur gelijk is aan "lightgray", maak hem dan "darkgrey" en andersom.

43. Gebruik een *if-else* om de grijstint van de huizen te laten afwisselen.

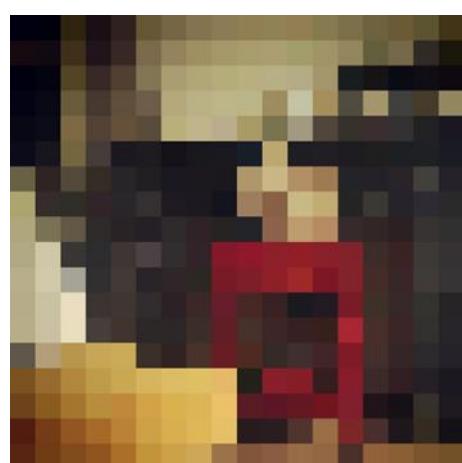


FIGUUR 2.14

★ Opdracht 9 obfuscator VII: pixels

44. Bekijk *OBF07*. Je ziet een mozaïek-versie van de kater in figuur 2.2, net als in figuur 2.15.

Als een foto-object geladen is, kun je met de functie `loadPixels()` van elke pixel van de foto informatie opvragen. *Voorbeeld 12* demonstreert hoe je met `get` kleurinformatie (in RGB-formaat) kunt opvragen. Voor elke pixel bevat het plaatje een array voor de hoeveelheid rood, groen en blauw van de pixelkleur.

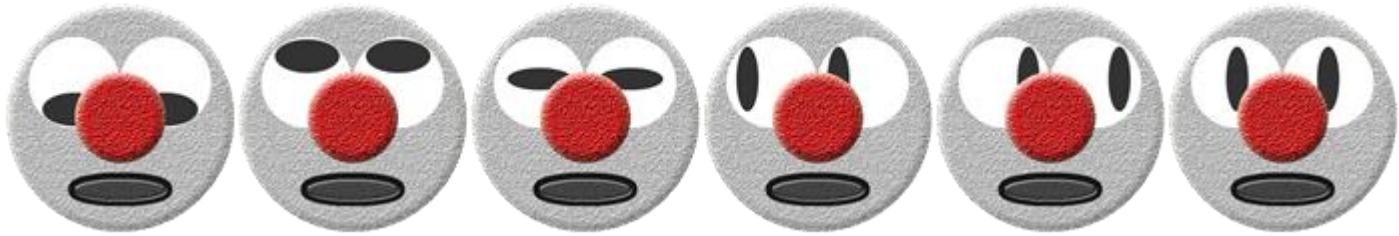


FIGUUR 2.15

45. Open *H2009.js* in jouw *editor*. Dit is *voorbeeld 12*. Bestudeer de code en in het bijzonder de manier waarop kleurinformatie wordt opgevraagd.
46. Bouw op basis van de gegeven code *OBF07* na.

2.4 animatie: een array van plaatjes

Een **sprite** kan één plaatje zijn, maar de term *sprite* wordt ook vaak gebruikt voor een bewegende plaatje of animatie. De beweging of animatie ontstaat door verschillende beeldjes of **frames** achter elkaar te tonen. Vroeger werden voor het maken van tekenfilms 24 frames getekend voor maar één seconde film. Als je de frames goed op elkaar laat aansluiten, creëer je daarmee de illusie van een vloeiende beweging.



FIGUUR 2.17

Hierboven zie je zes beeldjes van Jos, die we nummeren van 0 t/m 5. In *voorbeeld 13* zijn ze gebruikt om een array van plaatjes te maken.

De gebruikte code staat in figuur 2.16:

- o `var animatie = [];`
Met `[]` maak je een nieuwe, lege array.
- o `loadImage("Jos-" + b + ".png");`
In de for-loop wordt steeds een ander beeldje geladen, want de variabele `b` loopt van 0 t/m 5. We laden daarom achtereenvolgens de bestanden `Jos-0.png` t/m `Jos-5.png`.
- o `animatie.push(nieuw_beeldje);`
De functie `push` voegt steeds een nieuw element toe aan de array `animatie`. Na het uitvoeren van de for-loop hebben we dus een lijst met beeldjes: de array `animatie` is gevuld.

```
var animatie = [];           // maak een lege array
var aantalBeeldjes = 6;

function preload() {
    for (var b = 0; b < aantalBeeldjes; b++) {
        nieuw_beeldje =
            loadImage("Jos-" + b + ".png");
        animatie.push(nieuw_beeldje);
    }
}
```

FIGUUR 2.16

De volgende stap is om die beeldjes of frames achter elkaar te tonen. In hoofdstuk 1 hebben we kennis gemaakt met de loopfunctie `draw` die steeds opnieuw wordt uitgevoerd. Met `frameRate` stel je in hoeveel loops er per seconde zijn. Voorbeeld: `frameRate(2)` zorgt voor 2 frames per seconde; elke frame of loop duurt dus $1 / 2 = 0,5$ seconde. P5 kent een standaardvariabele `frameCount` die bijhoudt hoe vaak de loopfunctie `draw` is uitgevoerd vanaf het moment dat het programma geladen is.

In figuur 2.18 zie je hoe je dit in de `draw` kunt gebruiken om elke loop (frame) een ander beeldje kunt laden:

- o `nummer++;`
We gebruiken een variabele om bij te houden wat het juiste nummer van het frame is. Als het plaatje getoond is wordt het nummer met 1 verhoogd.
- o `image(animatie[nummer], 0, 0);`
laat het juiste plaatje uit onze array met beeldjes zien. Voorbeeld: `animatie[3]` is het 4^e (!) plaatje.
- o `if (nummer == aantalBeeldjes) {`
 `nummer = 0;`
}
Het laatste plaatje heeft nummer 5, dus nummer 6 bestaat niet. Als we bij dit nummer zijn, moet hij opnieuw beginnen bij 0.

```
var nummer = 0;

function draw() {
    image(animatie[nummer], 0, 0);
    nummer++;
    if (nummer == aantalBeeldjes) {
        nummer = 0;
    }
}
```

FIGUUR 2.18

In de volgende paragraaf leren we een andere tactiek kennen, waarbij alle beeldjes in één bestand staan.

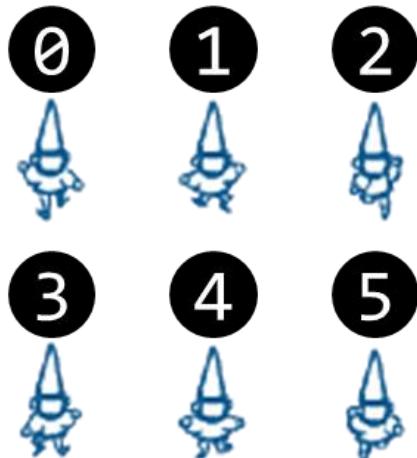


Opdracht 10 oefenen met animaties

47. Open *H2O10.js* in jouw *editor*. Dit is de code van voorbeeld 13. Bekijk het resultaat in de *browser*.
48. Voorspel wat je ziet als je regel 23 aanpast tot `image(animatie[nummer], 80, 160, 300, 300)`; Controleer je voorspelling.
49. Wat verandert er aan wat je ziet wanneer je de *framerate* (*frameRate*) ophoogt van 2 naar 5?
50. Als we alleen de eerste drie beeldjes (zie figuur 2.17) in de array laden, bewegen de ogen van Jos alleen nog van beneden naar boven (en weer terug). Verwerk dit in jouw programma.

Opdracht 11 frameCount en modulus

Op het internet zijn verschillende programma's te vinden om zelf sprites mee te tekenen. Ook zijn er talloze creatievelingen die hun werk met de wereld delen, zoals [Mitchell Vizensky](#). Van hem gebruiken we een serie beeldjes van een tovenaar ([bron](#)). De beeldjes zijn al voor je gedownload in jouw werkomgeving in de submap *images/sprites/wizard/opdracht_11A* met zes beeldjes (figuur 2.19).



FIGUUR 2.19

51. Open *H2O11.js* in jouw *editor* en bestudeer de code.
52. Leg uit wat de betekenis is van de regel:
`breedte = animatie[0].width;`
53. De gebruikte plaatjes zijn maar klein. Ze worden groter op het canvas getoond dan ze werkelijk zijn. Hoeveel keer groter?

In de theorie maken we gebruik van een variabele nummer om het juiste beeldje uit de array te halen. Met een nieuwe operator kunnen we ook gebruik maken van *frameCount*. Deze wiskundige truc heet **modulus**.

De modulus is een bovengrens. In ons geval **6** (`aantalBeeldjes = 6`). Met modulo rekenen (hiervoor wordt het procentteken gebruikt) kunnen we nu de *restwaarde van een deling* bepalen. Dat klinkt ingewikkeld maar is met een paar voorbeelden is het te begrijpen:

- 10 % 6 = 4** : Als je 10 deelt door 6, dan past de 6 er één keer in en houd je nog **4** over.
12 % 6 = 0 : Als je 12 deelt door 6, dan past de 6 precies twee keer. Je houdt geen (**0**) restant over.
5 % 6 = 5 : Als je 5 deelt door 6, dan past de 6 er nul keer in en houd je dus **5** over.

De *frameCount* bevat het aantal keren dat de loopfunctie *draw* is uitgevoerd en wordt dus steeds met 1 verhoogd. Als we als volgnummer voor de array met beeldjes *frameCount % aantalBeeldjes* invullen, dan is de uitkomst achtereenvolgens 0, 1, 2, 3, 4, 5 en dan opnieuw 0, 1, 2, 3, 4, 5, etc.

54. Vervang jouw *draw* door de code in figuur 2.20. Bekijk het resultaat. (De regels voor de achtergrond en de teksten zijn niet veranderd.)

```
function draw() {  
    background('lavender');  
    nummer = frameCount % aantalBeeldjes;  
    image(animatie[nummer], 150, 0, breedte, hoogte);  
    text("frameCount=" + frameCount, 5, 40);  
    text("nummer=" + nummer, 5, 70);  
}
```

FIGUUR 2.20

Als je afbeeldingen in je programma laadt, moet je goed letten op de bestandslocatie (de map of in het Engels: *directory*) en de precieze bestandsnaam. Als de computer de afbeelding niet kan vinden omdat je een klein foutje hebt gemaakt, laat hij *Loading...* zien. Onthoud dit: het zal je vast eens gebeuren!

In jouw werkomgeving staat in de map *wizard* in de submap *opdracht_11B*. Deze bevat een andere serie sprites met andere bestandsnamen en een ander aantal.

55. Bekijk de inhoud van de submap *opdracht_11B*. Hoeveel beeldjes zijn er? Hoe is de bestandsnaam opgebouwd?
56. Pas de *preload* aan zodat de beeldjes uit deze nieuwe map worden geladen. Vergeet niet om voor de variabele *aantalBeeldjes* het juiste aantal frames in te vullen! Bekijk het resultaat.

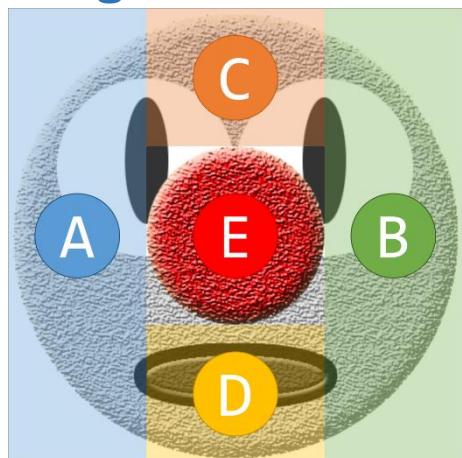
★ Opdracht 12 obfuscator VIII: kijkrichting

In *OBF08* reageren de ogen van Jos op de positie van de muis. De programmeur heeft dit gedaan door de afbeelding van Jos in gedachten op te delen in vijf delen A, B, C, D en E (figuur 2.21). Als de muis in gebied A is kijkt Jos naar links, in B naar rechts, in C naar boven en in D naar beneden. Als de muis zich in E bevindt kijkt Jos recht vooruit (net als wanneer de pagina wordt geladen).

57. Bekijk *OBF08* en beweeg daarbij je muis over de figuur. Stel vast welk plaatje in welke situatie wordt geladen en welk volgnummer van de array animatie hoort bij dit plaatje.

Ter herinnering: voor de positie van de muis heeft P5 de standaard variabelen `mouseX` en `mouseY`.

58. Open *H2O12.js* in jouw *editor* en bestudeer de code.
59. Bouw op basis van de gegeven code *OBF08* na. HINT: Gebruik *if* en *else* om de variabele nummer de juiste waarde te geven.



FIGUUR 2.21

Opdracht 13 overloper III: sprites invoegen

We gaan weer een stukje aan ons spel *overloper* (figuur 2.22) toevoegen. We gebruiken de bestanden in de map *images/sprites/Jos100px* om Jos meer tot leven te laten komen.

60. Open *H2O13.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van *overloper II*. Wel is alvast een lege array animatie gemaakt in regel 5 en is regel 14 toegevoegd om het eerste beeldje uit de map te laden.
61. Gebruik een for-loop om alle zes beeldjes uit de genoemde map te laden in de array animatie.
62. In regel 7 is de variabele nummer gedeclareerd met de waarde 3. Gebruik deze variabele om te zorgen dat het plaatje met volgnummer 3 wordt getoond als het programma wordt geladen.

In de vorige *overloper*-opdracht hebben we er al voor gezorgd dat Jos te besturen is met de pijltjestoetsen. Het doel van deze opdracht is dat aan Jos te zien is naar welke kant hij als laatste gelopen is, zoals in figuur 2.22.

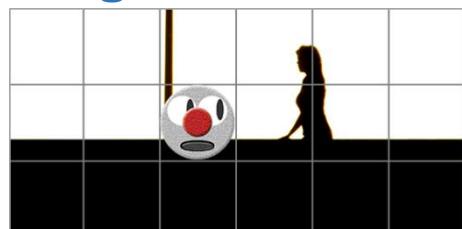
Voorbeeld: Als op de linker pijl is gedrukt, willen we dat Jos vanaf dat moment naar links kijkt (tot er op een andere pijl wordt gedrukt). In figuur 2.24 zie je de code die nodig is om dit te bereiken.

63. Pas het programma aan zodat aan bovenstaande eisen wordt voldaan (voor alle bewegingsrichtingen).

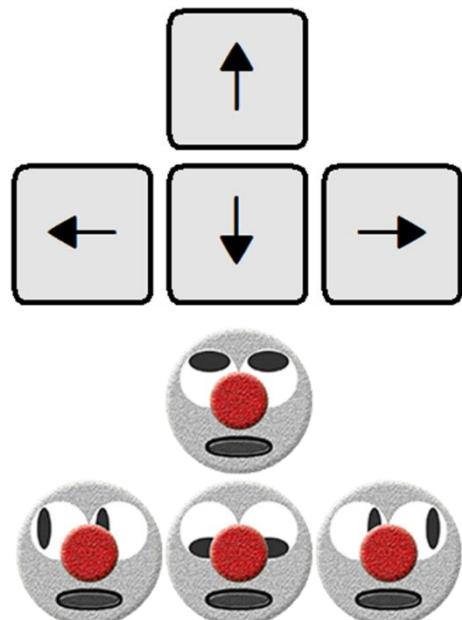
De plaatjes die we nu gebruiken zijn allemaal precies 100×100 pixels. Die afmetingen passen precies bij onze keuzes voor de grootte van ons raster en de afmetingen van het canvas. Want: de grootte van één cel is nu exact gelijk aan de grootte van één afbeelding.

Wat nu als dit niet zo is?

64. Verdubbel het aantal rijen en kolommen (zie regel 1 en 2). Bekijk het resultaat: hoe groot is Jos t.o.v. het raster? Hoe beweegt hij nu?
65. Pas de regel met `image` aan, zodat Jos weer precies in één cel past.



FIGUUR 2.22



FIGUUR 2.23

```
if (keyIsDown(LEFT_ARROW)) {  
    xJos -= celGrootte;  
    nummer = 2;  
}
```

FIGUUR 2.24

★ 2.5 VERDIEPING: spritesheets

In de vorige paragraaf hebben we geleerd om animaties te maken met een array van plaatjes. In de praktijk wordt vaak gewerkt met maar één grote afbeelding waarin alle losse beeldjes achter elkaar zijn geplaatst, zoals in figuur 2.25. Deze **spritesheet** heeft frames van 460×460 pixels, dus de hele afbeelding is één frame is 4140×920 pixels groot ($9 \times 460 = 4140$ en $2 \times 460 = 920$).



FIGUUR 2.25

Om een afbeelding in het canvas te tonen gebruiken we de functie `image`. We hebben al gezien dat je aan deze functie niet altijd hetzelfde aantal parameters hoeft mee te geven. Ter herinnering:

- `image(kater, 10, 25);`
Plaats het object (of: de afbeelding) kater en toon het (op ware grootte) op positie $x = 10$ en $y = 25$
- `image(kater, 10, 25, 50, 40);`
Toon kater op positie $x = 10$ en $y = 25$, maar nu met *breedte* 50 en *hoogte* 40

Ons doel is om maar een klein stukje van onze spritesheet te tonen. Hiervoor is extra informatie nodig. Om het **groen gemaakteerde deel** uit de spritesheet van figuur 2.25 te selecteren, gebruiken we:

- `image(spriteSheet, 10, 25, 50, 50, 1380, 0, 460, 460);`
Toon spriteSheet op positie $x = 10$ en $y = 25$, met *breedte* en *hoogte* 50.
Ga binnen de gebruikte afbeelding eerst naar de linkerbovenhoek van het groene vlak:
 1380 (3×460) opzij en 0 naar beneden. Selecteer vanaf dat punt een gebied van 460×460 pixels.

Dit zijn wel erg veel attributen. We zetten het daarom nog eens rustig op een rij voor het **geel gemaakteerde frame** dat je selecteert met `image(spriteSheet, 10, 25, 50, 50, 2300, 460, 60, 460)`
De betekenis van deze waarden wordt toegelicht in onderstaande tabel waarin we vier delen zien:

Waar op het canvas moet de afbeelding worden geplaatst?		Hoe groot moet de afbeelding worden weergegeven?		Waar zit de linkerbovenhoek van het (gele) vlak?		Welk gedeelte wil je vanaf de linkerbovenhoek selecteren?	
x canvas	y canvas	breedte	hoogte	x sprite	y sprite	breedte	hoogte
10	25	50	50	2300	460	460	460
x	y	br	ho	5×460	1×460	sBr	sHo

De tabel laat zien dat je kunt werken met de volgnummers voor *rij* en *kolom*. In *voorbeeld 14* zijn de variabelen zo gemaakt dat de volgende twee regels dezelfde uitkomst hebben:

```
image(spriteSheet, 10, 25, 50, 50, 2300, 460, 460, 460);
image(spriteSheet, x, y, br, ho, kolom*sBr, rij*sHo, sBr, sHo);
```

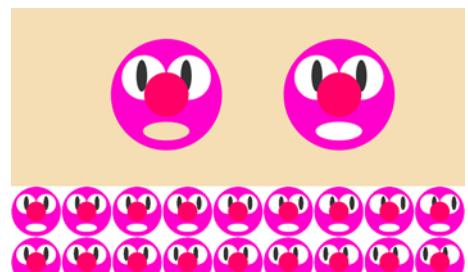
Het gebruik van al die variabelen is in eerste instantie ingewikkelder, maar zorgt wel dat we, als we een andere sprite willen gebruiken, niets meer aan onze code hoeven te veranderen om het te laten werken! Je hoeft alleen nog zelf het aantal rijen en kolommen te tellen en daarna doet de computer de rest.

Als we ook nog de truc met het modulo rekenen uit de vorige paragraaf toepassen, kunnen we het kolomnummer laten oplopen met `frameCount % aantalSpriteKolommen`. Als we de sprite naar links willen laten kijken zoals in de onderste rij (*rij* = 1) van figuur 2.25 dan komen we tot:

```
image(spriteSheet, x, y, br, ho, (frameCount % aantalSpriteKolommen)*sBr, 1*sHo, sBr, sHo);
```

★ Opdracht 14 oefenen met spritesheets

66. Open *H2O14.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet een variant op voorbeeld 14 met links een animatie en rechts één frame uit de animatie (figuur 2.26).
67. Bestudeer de code. Welk frame wordt op dit moment rechts getoond in figuur 2.26?
68. Er is één frame waarbij ons *game character* met de naam Alice de inkleuring van de mond mist (zie figuur 2.26 links). Pas regel 38 aan, zodat dit frame steeds aan de rechterkant wordt getoond. Welke getallen moet je invullen?
69. Verander de waarde van de variabele *schaal* in 0.33
Waarom verandert het rechterplaatje niet mee?
70. Hoe snel kun jij wisselen tussen van links naar rechts kijken?
Pas de framerate aan, zodat Alice net zo snel van kijkrichting wisselt als jij.



FIGUUR 2.26

★ Opdracht 15 Pony

Online zijn vele spritesheets te vinden. Kijk vooral zelf rond of maak je eigen spritesheet. Hiervoor zijn gratis programma's beschikbaar. In deze opdracht gebruiken we de spritesheet *Pony* ([bron](#)) van TrueFrenzy ([Markus Boch](#)). De spritesheet (figuur 2.27) is al voor je gedownload in jouw werkomgeving in de map *images/sprites*.



FIGUUR 2.27

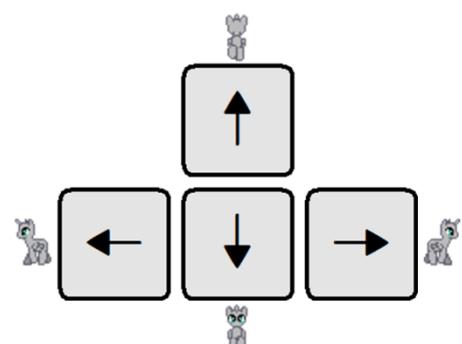
71. Open *H2O15.js* in jouw *editor*. Als je kijkt naar het resultaat in de *browser* dan zie je *Pony* die alleen vooruit loopt.
Oftewel: alleen de beeldjes uit de bovenste rij worden doorlopen.
72. Zorg dat de animatie één voor één alle rijen doorloopt en weer terugkeert naar de bovenste rij als alle frames zijn geweest.

★ Opdracht 16 obfuscator IX: Pony besturen

De spritesheet in figuur 2.27 bevat vier rijen van animaties die je zou kunnen opvatten als vier looprichtingen. Eerder hebben we al geleerd hoe we een object kunnen laten reageren op de pijltjestoetsen om het over het canvas te laten bewegen.

73. Bekijk *OBF09* en druk op de pijltjestoetsen. Je kunt *Pony* over het canvas laten lopen.

We gaan *OBF09* nabouwen. Denk hierbij aan de volgende aandachtspunten en hints:



FIGUUR 2.28

- De besturing gaat met de pijltjestoetsen. In figuur 2.28 kun je zien welke richting bij welke rij van de spritesheet hoort.
- Als *Pony* van looprichting wisselt, betekent dit voor de animatie dat de waarde van de variabele *rij* moet veranderen.
- Als *Pony* een rand van het canvas bereikt, kan hij niet verder lopen. Dit is bereikt met de functie `constrain`. Gebruik hierbij de breedte van het frame, zoals die op het canvas verschijnt.

74. Bouw *OBF09* na volgens bovenstaande aanwijzingen.

OBF09 bevat wel iets vreemds: de animatie gaat door, ook als we geen enkele pijltjestoets indrukken. We willen eigenlijk dat *Pony* helemaal stil blijft staan (op hetzelfde frame) als er niet op een toets wordt gedrukt.

75. Pas jouw code aan zodat ook aan deze extra eis wordt voldaan.

2.6 zelf objecten maken

In de vorige paragrafen heb je kennis gemaakt met de bijzondere objecten afbeelding en lijst (array). Ze zijn bijzonder, omdat hun werking afwijkt van andere objecten, maar belangrijk voor nu is dat we hebben gezien dat ze eigenschappen hebben en dat je ze kunt vragen om een handeling uit te voeren (figuur 2.29).

AFBEELDINGEN	LIJSTEN (ARRAYS)	OBJECT kever
// attributen: eigenschappen spriteJos.width; spriteJos.height; // methodes: handelingen spriteJos.hide();	// attributen: eigenschappen klas.length; klas[2]; // methodes: handelingen klas.sort(); klas.push("Trent");	// attributen: eigenschappen kever.x; kever.y; // methodes: handelingen kever.beweeg();

FIGUUR 2.29

Het maken van programma's met objecten wordt **object-georiënteerd** programmeren genoemd. Het is een bepaalde tactiek van programmeren, ook wel **programmeer-paradigma** genoemd, waarbij je eigenschappen en handelingen koppelt onder één noemer: het **object**. Een voorbeeld:

In *voorbeeld 10* (zie ook § 2.2) laten we een kever over het canvas bewegen. Er is een variabele kever voor de bijbehorende sprite en er zijn variabelen keverX en keverY om zijn positie mee te bepalen. Daarnaast is er code om de kever te laten bewegen.

Eigenlijk is dit onhandig, want dit zijn allemaal *losse* stukjes code en gegevens, terwijl ze eigenlijk *bij elkaar* horen. Net zoals bij de afbeeldingen en lijsten, willen we de kever vragen om informatie over zichzelf te geven of om iets voor ons te doen, zoals in het rechterblok van figuur 2.29. We willen van de kever een **object** maken.

In *voorbeeld 15* is de kever van *voorbeeld 10* nogmaals te zien, maar nu object-georiënteerd. De declaratie van ons object kever zie je in figuur 2.30.

Een eigenschap van een object heet een **attribuut**. In dit voorbeeld zijn er drie attributen: x, y en sprite. De eerste twee hebben we meteen een waarde gegeven, maar sprite blijft nog even *leeg* (omdat we daar straks in de *preload* een afbeelding inladen). Als je een attribuut nog niet meteen een waarde meegeeft, kun je dat aangeven met **null**. Let goed op het gebruik van { }, : en , bij het maken van een object. Dat komt heel precies!

Een handeling van een object heet een **methode**. Als we de kever vragen om te bewegen via de methode **beweeg()**, dan worden drie regels uitgevoerd. Deze regels gebruiken de attributen x, y en sprite, maar wel steeds voorafgegaan door **this**.

Het gebruik van **this** is even wennen. Het is een verwijzing naar een attribuut of methode van het object zelf. In ons voorbeeld is dat (de) kever. Je schrijft dan **this.x** (en niet **kever.x**) omdat het gaat om eigenschappen en handelingen van *dit* object. Dat gebruiken van **this** doe je alleen bij het maken van een object. In het hoofdprogramma gebruik je de naam die je aan het object hebt gegeven; voor een attribuut:

```
text("De kever bevindt zich op x-positie" + kever.x, 0, 0);  
en voor een methode:  
kever.beweeg();
```

```
var kever = {  
    // attribuut  
    x: 100,  
    y: 150,  
    sprite: null,  
  
    // methode  
    beweeg() {  
        this.x += random(-5, 5);  
        this.y += random(-5, 5);  
        image(this.sprite, this.x, this.y);  
    }  
};
```

FIGUUR 2.30



Opdracht 17 werken met objecten

76. Open *H2O17.js* in jouw *editor*. Dit is de code van *voorbeeld 15* met enkele kleine aanpassingen. Bekijk het resultaat in de *browser*.
77. De tekst onderaan verandert op dit moment niet mee als de positie van de kever verandert. Pas de code aan, zodat de actuele waarden van *x* en *y* worden getoond.
78. Voeg een attribuut naam toe aan het object kever en geef als waarde een zelfbedachte naam mee. LET OP: omdat een naam een tekst is, moet deze tussen aanhalingstekens.
79. Pas de getoonde tekst aan zodat in plaats van “*Het object kever*” de door jou gekozen naam verschijnt. Gebruik de code *kever.naam* voor het tonen van de naam.

Opdracht 18 Jos als object

80. Open *H2O18.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet de getekende versie van Jos uit hoofdstuk 1.

In hoofdstuk 1 werd Jos getekend met een zelfgemaakte functie. De coderegels staan nu als methode teken binnen het object jos.

81. Hoeveel attributen heeft het object jos?
82. Wat is op dit moment de waarde van het attribuut *jos.x*?
83. De methode teken heeft nu als parameter *muispositieX*. De bedoeling is dat Jos ook echt gaat meebewegen met de muis. Pas de regel *jos.teken(500);* aan, zodat Jos reageert op de x-positie van de muis.

We willen dat Jos groter wordt als hij naar rechts beweegt en kleiner als hij naar links beweegt.

84. Voeg de volgende regel toe aan teken in regel 9: *this.schaal = this.x / (0.25*width);*
85. Zorg dat in de tekst bovenaan behalve de x-positie ook de schaal van Jos wordt getoond.
86. Pas de code aan, zodat Jos ook reageert op de y-positie van de muis. Doe de volgende stappen:
 - Zorg dat de methode teken een extra parameter *muispositieY* krijgt.
 - Gebruik de functie *constrain* om te zorgen dat *y* (van Jos) tussen de 100 en 150 blijft.
 - Zorg dat bij het aanroepen van de methode teken de y-positie van de muis als parameter wordt meegegeven.



FIGUUR 2.31

Opdracht 19 overloper IV: het raster als object

In deze opdracht gaan we het paradigma van object-georiënteerd programmeren toepassen op het spel *overloper* dat als rode draad door dit hoofdstuk loopt. We beginnen met het raster.

87. Open *H2O19.js* in jouw *editor* en bestudeer de code. Dit is qua werking het eindresultaat van *overloper III*.
88. Maak een object raster met de attributen aantalRijen, aantalKolommen en celGrootte en geef ze achtereenvolgens de waarden *6, 6* en *null* mee.
89. Voeg de methode *berekenCelGrootte()* toe aan het object raster volgens het voorbeeld in figuur 2.32.
90. Om de celGrootte te berekenen, moet deze methode nog wel worden aangeroepen. Vraag binnen de *setup* het object raster om de methode *berekenCelGrootte()* uit te voeren.
91. De oude variabele celGrootte willen we niet meer gebruiken. Pas alle coderegels die celGrootte gebruiken aan, zodat ze het attribuut *raster.celGrootte* gebruiken.
92. Voeg de methode *teken()* toe aan het object raster, zodat de regel *raster.teken();* zorgt voor het tekenen van het raster. Gebruik de coderegels uit de *oude* functie *tekenRaster()* als basis.
93. Welke *oude coderegels* kun je nu allemaal verwijderen?
94. Voeg de regel *raster.teken();* toe aan de *draw*. Wordt het raster nog steeds getekend?

```
berekenCelGrootte() {  
    this.celGrootte =  
        width/this.aantalKolommen;  
}
```

FIGUUR 2.32

Opdracht 20 overloper V: het bijzondere object jos

In de vorige opdracht heb je zelf van het raster een object gemaakt. In dit geval hebben we alvast het object jos voor je gemaakt. Doel van deze opdracht is dat je begrijpt wat ze betekenen.

95. Open *H2O20.js* in jouw *editor*. Hoeveel attributen heeft object jos? En welke methodes heeft jos?

Eén van de attributen van jos is de `stapGrootte`. In eerste instantie heeft deze geen waarde (`null`).

96. Door welke coderegel krijgt `stapGrootte` wel een waarde? Welke waarde is dat?

97. Verklaar door naar de code te kijken waarom jos bij een beweging naar links een andere afstand aflegt dan in de overige richtingen.

98. Pas de code aan, zodat jos bij een beweging naar links ook een afstand `jos.stapGrootte` aflegt.

Eén van de attributen van jos is zelf een object, namelijk de array animatie. In de `preload` worden alle afbeeldingen (frames) die we gebruiken in de array animatie geladen.

99. Leg in je eigen woorden uit wat de betekenis is van `jos.animatie.push(frame);` (regel 61).

100. Doe een voorspelling over wat je op het scherm zal zien als de coderegel

```
text(jos.animatie[3].width, 5, 15);
```

aan het eind van `draw` wordt toegevoegd. Controleer daarna of je voorspelling klopt.

101. Maak het aantal rijen en kolommen van het raster drie keer zo groot (18 respectievelijk 27).

102. De `draw` is ons hoofdprogramma. Uit hoeveel coderegels bestaat `draw` nog, nu we werken met objecten?

Opdracht 21 zoekspelletje I

Zie je de donkergele cirkel op de foto in figuur 2.33? We gaan een spel maken waarbij de speler de cirkel moet vinden en aanklikken. Te makkelijk? Elke keer dat je goed klikt, verschijnt de cirkel op een andere plek, maar dan kleiner en iets meer doorzichtig.

103. Open *H2O21.js* in jouw *editor* en bestudeer de code. Voer de code uit. Wat gebeurt er als je op een toets drukt?

Het zwarte scherm dat verschijnt als je op een toets drukt, is bedoeld als hulp voor de programmeur (of als *cheat* voor de speler!):

Als je de cirkel kwijt bent, kan het zijn dat je een programmeefout hebt gemaakt of dat je hem zelf niet kunt vinden. Door op een knop te drukken, kun je vaststellen wat er aan de hand is.

104. Verklaar dat de cirkel op een andere plek verschijnt als je de pagina opnieuw laadt.



FIGUUR 2.33

Het object `cirkel` heeft een methode `controleerRaak()` die de volgende stappen moet gaan uitvoeren:

- Bepalen van de afstand van de muis tot het middelpunt van de cirkel. Hiervoor is al een coderegel: `afstandMuisCirkel = dist(mouseX, mouseY, this.x, this.y);` (Zie de uitleg van `dist` in opdracht 22 van H1 of de *reference*: <https://p5js.org/reference/#/p5/dist>)
- Als het raak is, moet het attribuut `alpha` (die de doorzichtigheid van `cirkel` bepaalt) 20% kleiner worden, ofwel de vorige waarde maal 80%. Bovendien moet de cirkel een nieuwe plek kiezen.

105. Breid de methode `controleerRaak()` uit zodat aan alle eisen is voldaan. Gebruik een if-structuur en het feit dat het *raak* is, als de onderlinge afstand kleiner of gelijk is aan de straal van de cirkel. Hint: in hoofdstuk 1 hebben we `if (mouseIsPressed == true)` gebruikt, om te controleren of de speler met de muis klikt.

106. Het object `cirkel` heeft een attribuut `aantalRaak`. Zorg dat deze met één wordt verhoogd, als iemand raak klikt. Maak vervolgens een coderegel in de `draw` die zorgt dat je op het scherm kunt zien hoe vaak je raak hebt geklikt. Gebruik hierbij de `text`-functie.

★ Opdracht 22 natuurlijk bewegen

In voorbeeld 16 zie je een bal die tegen de wanden van het canvas terugkaatst. De bal is hier een object. De beweging van de bal lijkt misschien wel op een beweging die je vaker in spellen ziet, maar het lijkt niet erg op de beweging van een echte bal. Die stuitert meer zoals in figuur 2.34.

In deze opdracht gaan we de bal natuurlijker laten vallen en stuiteren.



FIGUUR 2.34

107. Open H2O22.js in jouw editor en bestudeer de code.
108. Hoeveel attributen heeft object bal? En welke methodes heeft bal?
109. Wat is de waarde van bal.y na het uitvoeren van de `setup`?

In de methode `beweeg` staan de coderegels:

```
if (this.x <= this.straal || this.x >= canvas.width - this.straal) {  
    this.snelheidX *= -this.demping; }
```

Als een bal ergens tegenaan botst, verliest het doorgaans een deel van zijn snelheid. In dit geval verliest hij nog geen snelheid, omdat `this.demping` de waarde **1.0** heeft. Als dit wordt veranderd naar **0.95** verliest de bal 5% van zijn snelheid, want de nieuwe waarde is nog 95% (**0.95**) van de vorige waarde. Merk op: door `canvas.width` te gebruiken, maken we hier gebruik van het feit dat `canvas` zelf ook een object is!

110. Voor welke (getals-) waarden van `this.x` weerkaatst de bal hier?

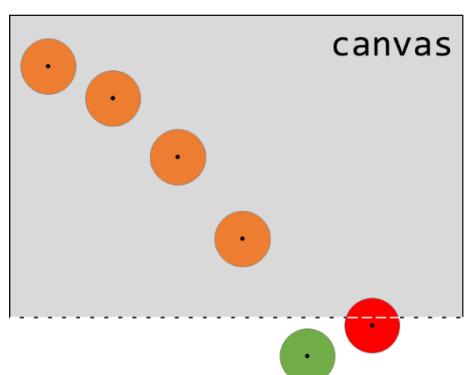
111. Stel `this.demping` in op **0.95** en bekijk het resultaat.

Als je voldoende geduld hebt bij de laatste opdracht, dan heb je misschien gezien dat de bal op een gegeven moment tegen de rand van het canvas blijft *plakken*. Dat probleem lossen we straks in deze opdracht op. Eerst gaan we de beweging van de bal natuurlijker maken.

De bal beweegt op dit moment in een rechte lijn. In het echt valt een bal naar beneden, omdat de aarde eraan trekt. Tijdens het vallen gaat de bal steeds sneller: er is een *versnelling*. Als de bal weer omhoog beweegt, remt hij juist af, tot het hoogste punt waar de snelheid 0 is.

112. Geef het attribuut `snelheidY` de beginwaarde **0** en `demping` de waarde **0.9**.
113. Voeg een nieuw attribuut `versnelling` toe met beginwaarde **0.2**.
114. Voeg als eerste regel van de methode `beweeg` de volgende coderegel toe:
`this.snelheidY += this.versnelling`
Bij elke stap in de beweging wordt nu **0.2** opgeteld bij de snelheid van dat moment.
115. Bekijk het resultaat. Hoe zorgt deze regel ervoor dat de bal afremt als hij naar boven gaat?

We zien opnieuw dat de bal op een zeker moment aan de rand van het canvas blijft *plakken*. Hoe kan dat? De tekening in figuur 2.35 toont schematisch wat er gebeurt. De hoogte van de bal wordt na elke loop van `draw` opnieuw berekend. De bal verplaatst zich hierdoor in stapjes. Op een bepaald moment komt de bal onder de rand van het canvas (groen gekleurd). De verticale snelheid moet dan omkeren, want de bal moet weer omhoog: `this.snelheidY=-0.95;`



FIGUUR 2.35

Nu kan het gebeuren (het hoeft niet!), dat het middelpunt van de bal bij de eerstvolgende stap nog steeds onder de rand van het canvas zit (rood gekleurd). Het programma zal hierop reageren door opnieuw de snelheid om te keren, waardoor de bal weer naar beneden gaat. Dat was niet de bedoeling! Een manier om dit op te lossen is om de bal precies aan de onderkant van het canvas te plaatsen, op het moment dat deze op de groen gemarkeerde plek belandt: `this.y = canvas.height - this.straal;`

116. Leg uit waarom `this.straal` hier nog van de hoogte van het canvas wordt afgetrokken.
117. Pas de 2^e if-structuur aan zodat er komt te staan: `if (this.y>=canvas.height-this.straal) {`
zodat deze alleen nog reageert als de bal onderaan het canvas is.
118. Voeg daarna aan de regel `this.y = canvas.height - this.straal;` toe aan deze if-structuur.
119. Ziet het resultaat er *echt* uit? Pas desnoods de demping aan totdat je tevreden bent.

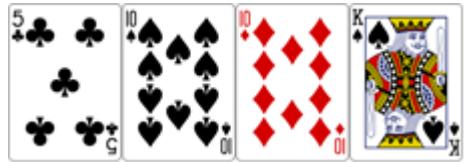


Opdracht 23 obfuscator X: speelkaarten kiezen

Een pak met speelkaarten bestaat uit 52 verschillende kaarten: 13 verschillende kaarten in de soorten klaver, schoppen, ruiten en harten. Als je *OBF10* bekijkt, kun je als speler willekeurig vier kaarten uit het pak van 52 kiezen door met de muis te klikken. Het resultaat ziet eruit zoals in figuur 2.36.

120. Bekijk *OBF10* en klik een aantal keren met je muis.
121. Open *H2O23.js* in jouw editor en bestudeer de code. Bekijk het resultaat in de *browser*. Het scherm toont telkens opnieuw een willekeurig gekozen speelkaart.

De code bevat al drie objecten:



FIGUUR 2.36

- Het object *kaartSoorten* is een array met de vier soorten kaarten in het spel, afgekort met letters. Deze lijst wordt gebruikt voor het laden van de plaatjes.
- Het object *kaartSpel* is een array met daarin alle 52 afbeeldingen behorende bij de verschillende kaarten. (Die afbeeldingen zijn eigenlijk ook nog objecten!)
- Het object *speler* dat een kaart uit *kaartSpel* kan trekken (*trekKaart()*) en zijn getrokken kaarten (attribuut *getrokkenKaarten*) op het scherm kan laten zien (*methode toonKaarten*). Hierbij wordt de wiskundefunctie *floor* gebruikt, die alle kommagetallen (die door *random* worden gekozen) naar beneden afrondt. Zie ook de *reference*: <https://p5js.org/reference/#/p5/floor>

De code van *H2O23.js* is nog niet af. Een aantal stappen die nog moeten worden gezet zijn:

- De methode *toonKaarten* toont op dit moment de laatst getrokken kaart, maar moet alle getrokken kaarten netjes op een rij tonen.
- Op dit moment wordt automatisch steeds een nieuwe kaart getrokken, maar dit moet alleen als er wordt geklikt en er minder dan vier kaarten zijn getrokken.
- Als er al vier kaarten zijn getrokken en er toch weer wordt geklikt, moet er een tekst verschijnen.

122. Pas de code aan, zodat het gedrag van jouw programma gelijk is aan *OBF10*.

In theorie is het nu nog mogelijk dat je twee keer dezelfde kaart trekt, zoals in figuur 2.37 (tenzij je daar al aan hebt gedacht!).



FIGUUR 2.37

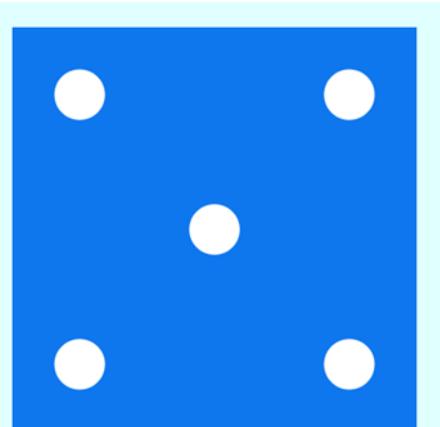
123. Breid de methode *trekKaart* uit, zodat deze voorkomt dat er twee keer dezelfde kaart wordt getrokken. Een mogelijke oplossing is: controleer voordat de getrokken kaart aan *getrokkenKaarten* wordt toegevoegd of deze al in de lijst zit.

Opdracht 24 dobbelstenen I

124. Open *H2O24.js* in jouw editor en bestudeer de code. Voer de code uit. Wat gebeurt er als je met je muis klikt?

De code om het object *dobbelSteen* te maken is best uitgebreid. Dit komt vooral door de methode *teken()* die regelt dat de witte stippen op de steen bij elk aantal ogen op de juiste plaats worden getekend. Deze methode is helemaal klaar.

In de methode *gooi* staat *this.ogen = floor(random(0, 6)) + 1;*
De wiskundefunctie *floor* rondt alle kommagetallen (die door *random* worden gekozen) naar beneden af.



FIGUUR 2.38

125. Wat is het verschil in mogelijke uitkomsten tussen *round(random(0, 6))* en *floor(random(0, 6))*?
126. Geef de dobbelsteen zijden van 200 pixels en zwarte stippen.
127. Maak een attribuut *totaal* dat het totaal van alle worpen bijhoudt en zorg dat dit totaal steeds zichtbaar is in het canvas.

2.7 een object dat ja of nee antwoordt

Objecten hebben attributen en methodes. Met een methode kan een handeling worden verricht. Dit kan ook betekenen het programma (of een ander object) aan een object iets vraagt en dat het antwoord geeft.

In hoofdstuk 1 heb je misschien *jager* en *prooi* gemaakt, waarin je een vierkantje (de *jager*) met de pijltjestoetsen kan besturen om een rechthoek (de *prooi*) te raken. Als het raak is, verandert de prooi van kleur. Hierbij moet het programma voortdurend de vraag stellen: *raakt de jager de prooi of niet?*

In voorbeeld 17 is *jager* en *prooi* opnieuw gemaakt, maar nu in het object-georiënteerde paradigma. Het programma stelt steeds twee vragen:

- *Jager, zit je vlakbij de rand?*
- *Prooi, wordt je geraakt door jager?*

Deze hebben slechts twee mogelijke uitkomsten: het is *waar* of *niet waar*, in het Engels **true** of **false**.

In figuur 2.39 zie de methode `vlakbijRand()` van het object *jager* dat een antwoord teruggeeft met `return`. Dat antwoord is **true** of **false** op basis van de x-positie `this.x` van het object *jager*.

De methode doet zijn werk pas als je hem aanroeft. Dat zie je in figuur 2.40: als de *jager* zich te dicht bij de rand van het canvas bevindt, wordt de achtergrond van het canvas rood.

Merk op: tussen de `()` van de `if` staat nu geen vergelijking meer (zoals `hoogte > 150`). In plaats daarvan roepen we de methode `vlakbijRand()` aan. Die geeft antwoord met *ja* of *nee*.

De vraag aan de *prooi* of hij geraakt wordt door de *jager* is anders dan de vraag aan de *jager* of hij *vlakbij de rand zit*, omdat het object *prooi* hiervoor informatie nodig heeft van het object *jager* (waar ben je?).

figuur 2.41 toont het voor nu belangrijkste deel van de code waarmee het object *prooi* is gemaakt. Het attribuut `benGeraakt` is hier geen getal of tekst, maar bevat de waarde **false**. Een attribuut of variabele die waar of niet waar is, heet een **boolean**.

De methode `wordJeGeraakt` heeft een zekere *vijand* als parameter. Dat is een object! Door eigenschappen van die *vijand* te vergelijken met zijn eigen eigenschappen, weet *prooi* of hij geraakt wordt. Is dit het geval, dan krijgt het attribuut `benGeraakt` de waarde **true**. De boolean `benGeraakt` wordt daarna in de methode `teken()` gebruikt om de vulkleur te bepalen.

We vragen de *prooi* of hij wordt geraakt door de *jager* met de coderegel:

```
prooi.wordJeGeraakt(jager);
```

Merk op dat hier als *vijand* *jager* wordt ingevuld.

Voorbeeld 17 is nog niet helemaal af. Dat doen we in de volgende opdracht.

```
vlakbijRand() {
    if (this.x < 10 || this.x > 990) {
        return true;
    }
    else {
        return false;
    }
}
```

FIGUUR 2.39

```
if (jager.vlakbijRand()) {
    background('red');
}
else {
    background('orange');
}
```

FIGUUR 2.40

```
var prooi = {
    x: 800,
    y: 175,
    benGeraakt: false,
    wordJeGeraakt(vijand) {
        if (vijand.x >= this.x - vijand.zijde) {
            this.benGeraakt = true;
        }
    },
    teken() {
        if(this.benGeraakt) {
            fill('white');
        }
        else {
            fill('green');
        }
    }
}
```

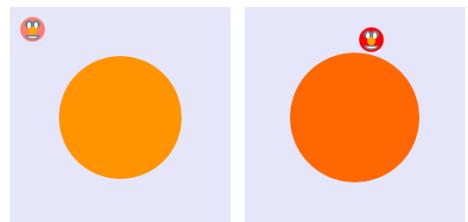
FIGUUR 2.41

✓ Opdracht 25 jager en prooi

128. Open *H2O25.js* in jouw *editor*. Dit is de code van voorbeeld 17. Bekijk het resultaat in de *browser*.
129. Bekijk de methode `vlakbijRand()` van het object *jager*. Hoeveel pixels mag de jager maximaal van de rand van het canvas zitten als hij wil voorkomen dat de achtergrond rood kleurt?
130. De methode `vlakbijRand()` werkt nu alleen voor de linker- en rechterrand van het canvas. Pas de code aan, zodat deze ook voor de boven- en onderrand van het canvas werkt.
131. Ook de methode `wordJeGeraakt` van het object *prooi* is nog niet af. Wat gaat er mis?
132. Vul de methode `wordJeGeraakt` aan, zodat de prooi wit kleurt als hij geraakt is door de jager.
133. We willen dat de prooi weer tot leven komt, als de jager te dicht bij de rand komt. Pas het programma aan, zodat de boolean *benGeraakt* weer de waarde `true` krijgt, als het object *jager* zich vlakbij (of tegen) de rand bevindt.

Opdracht 26 heet!

In deze opdracht gebruiken we opnieuw het object *jos* dat we in opdracht 18 voor het eerst hebben gebruikt. Jos (het object *jos*) bevindt zich in de buurt van een gevaarlijk vuur. Als hij te dichtbij komt krijgt hij een rood (*red*; zie figuur 2.42 rechts) hoofd. Gaat hij weer verder weg staan dan kleurt zijn hoofd weer zalmroze (*salmon*; zie figuur 2.42 links).



FIGUUR 2.42

134. Open *H2O26.js* in jouw *editor* en bestudeer de code. Voer de code uit.
135. Zoek uit hoe het object vuur van kleur en grootte wisselt.

Op dit moment kleurt het hoofd van Jos nog niet als hij te dichtbij het vuur komt. Hiervoor is binnen het object *jos* al wel een methode `isVlakbij` gemaakt. Deze maakt gebruik van de `dist`-functie.

136. Bestudeer de methode `isVlakbij`. Beschrijf in je eigen woorden wanneer deze methode de waarde `true` als antwoord teruggeeft.
137. In figuur 2.43 staat een if-else-structuur die op de met `//` gemarkeerde plek in de `draw` moet worden geplaatst. Tussen de haakjes (`.....`) moet gebruik gemaakt worden van de methode `isVlakbij`.
Pas de code aan zodat het gezicht van Jos rood kleurt als hij te dicht in de buurt van het vuur komt.

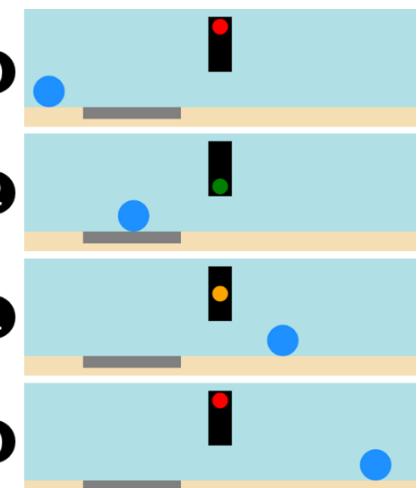
```
if ( ..... ) {  
    jos.kleur = 'red';  
}  
  
else {  
    jos.kleur = 'salmon';  
}
```

FIGUUR 2.43

★ Opdracht 27 obfuscator XI: stoplicht

In figuur 2.44 zie je een stoplicht dat verbonden is met een detectielus. Als er iets over de grijze rechthoek rolt, gaat het stoplicht op groen (2). Als de lus niets meer detecteert gaat de tijd lopen. Op een bepaald moment gaat het stoplicht op oranje (1) en even later naar rood (0).

Het object *stoplicht* gebruikt de methode `bepaalStand` om het stoplicht op rood, oranje of groen te zetten. Als parameter gebruikt deze methode de boolean *ikVoelIets* van het object *detectielus*. Het is de bedoeling dat *ikVoelIets* van waarde kan veranderen met behulp van de methode `detecteer` van het object *detectielus*.



FIGUUR 2.44

138. Bekijk *OBF11*.
139. Open *H2O27.js* in jouw *editor* en bestudeer de code.
140. Pas de methode `detecteer` aan zodanig dat jouw programma hetzelfde gedrag vertoont als *OBF11*.

Opdracht 28 overloper VI: een vijand voor jos

Bij de introductie van het spel *overloper* is verteld dat het de bedoeling is dat Jos de overkant bereikt en dat hij daarbij obstakels moet ontwijken. In figuur 2.45 zie je een eerste obstakel: Alice.

141. Open *H2O28.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van opdracht 20, aangevuld met een nieuw object Alice. Kijk in het bijzonder naar de code van dit object.

Alice staat nu stil, maar ook voor haar is er een methode om te bewegen. Hierin wordt de wiskundefunctie `floor` gebruikt, die alle kommagetallen (die door `random` worden gekozen) naar beneden afrondt. Zie ook de *reference*: <https://p5js.org/reference/#/p5/floor>

142. Bestudeer de methode `beweeg()` van Alice en voorspel welk resultaat deze methode zal hebben.
143. Zorg dat Alice gaat bewegen (in de `draw`). Klopt de manier van bewegen met jouw voorspelling?
144. Is het mogelijk dat Alice gedurende een frame stil blijft staan? Leg uit.

Het spel moet stoppen als Jos door Alice wordt geraakt. Om dat te bereiken is in de `draw` de code uit figuur 2.46 toegevoegd die gebruik maakt van de methode `wordtGeraakt` van Jos. We stellen hiermee de vraag of Jos geraakt wordt door Alice. Zo ja, dan stopt het spel.

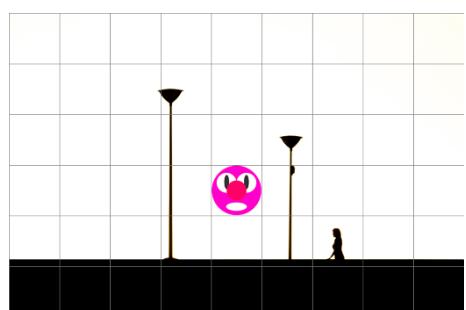
145. Pas de methode `wordtGeraakt` aan, zodat deze waarde `true` teruggeeft, als Jos en Alice zich op dezelfde plaats bevinden.



FIGUUR 2.45

```
if (jos.wordtGeraakt(alice))  
{  
    noLoop();  
}
```

FIGUUR 2.46



FIGUUR 2.47

Opdracht 29 overloper VII: gehaald

We hebben al veel gewerkt aan het spel *overloper*, maar op dit moment kun je het spel nog niet winnen. Er zijn vele manieren waarop we dit zouden kunnen programmeren. De vraag *heeft Jos de overkant gehaald* is ook te beantwoorden met ja of nee. Gezien het thema van deze paragraaf kiezen we daarom bij de oplossing voor een boolean *gehaald* als attribuut van het object *jos*.

146. Open *H2O29.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van opdracht 28.
147. Voeg een attribuut *gehaald* toe aan het object *jos* en geef deze de beginwaarde `false`.

Op dit moment kan Jos niet buiten het canvas omdat zowel *jos.x* als *jos.y* wordt beperkt door een `constrain`-functie. Voor *jos.x* is dit:

```
this.x = constrain(this.x, 0, canvas.width - raster.celGrootte);
```

148. Verander de laatste parameter van deze coderegel naar *canvas.width*. Welk resultaat verwacht je? Controleer het resultaat.
149. Wat is de waarde van *jos.x* als Jos rechts uit beeld is verdwenen?

Als Jos rechts uit beeld is verdwenen (zoals in figuur 2.47), moet de boolean *gehaald* de waarde `true` krijgen.

150. Voeg onderaan de methode `beweeg` (van *jos*) een if-structuur toe die dit regelt. Gebruik het antwoord op de vorige vraag.

Als de overkant is gehaald, willen we dat het spel stopt en in beeld komt te staan dat je hebt gewonnen, zoals in figuur 2.48.

151. Voeg een if-structuur toe aan de `draw` zodat er een eindscherm verschijnt als de overkant is gehaald.



FIGUUR 2.48

2.8 een klasse van objecten

Het spel *overloper* loopt als rode draad door dit hoofdstuk. In de vorige paragraaf is een vijand (Alice) toegevoegd. Stel nu dat we een tweede vijand willen toevoegen om het spel lastiger te maken. Dat kan door alle code van het object Alice te kopiëren en daarna een nieuwe naam voor dit object te bedenken. Erg logisch is dat niet, want we krijgen hierdoor allemaal ‘dubbele code’, bijvoorbeeld voor de methode *beweeg*. En wat als we een kleine aanpassing willen doen? Moet dat dan op twee plaatsen in de code? Veel liever zouden we één basiscode schrijven waar we steeds nieuwe versies van kunnen maken. Zo’n basis heet een **klasse**. Een nieuw exemplaar ervan – een nieuw object - heet een **instantie** van de klasse.

Instantie, klasse en object zijn belangrijke begrippen. Vergelijk het met het maken van voorwerpen met een 3D-printer. Eerst maak je een basis op de computer met een 3D-tekenprogramma. Dit is de klasse. Vervolgens kun je met de 3D-printer zoveel instanties printen als je wilt. Die instanties zijn de objecten.

In voorbeeld 18 zie je de bomen uit figuur 2.49. Het zijn drie objecten van dezelfde klasse. Daarom hebben ze veel overeenkomsten:

- De bomen hebben allemaal dezelfde eigenschappen. Dit zijn de attributen *leeftijd*, kleur en *x*. Hun waarde verschilt wel per instantie van de klasse *Boom*!
- De bomen worden allemaal volgens dezelfde tactiek getekend.
- De bomen kunnen allemaal groeien (zie boven en onder).

In figuur 2.50 zie je de code waarmee de algemene basis voor alle bomen, de **klasse**, is gemaakt. Let goed op het gebruik van `{}` en merk op dat er hier niet, zoals bij een object, gebruik gemaakt wordt van komma’s. Een klassenaam begint met een hoofdletter.

Een instantie van de klasse *Boom* maak je met een regel zoals:

```
boom1 = new Boom(1, 'olive', 130);
```

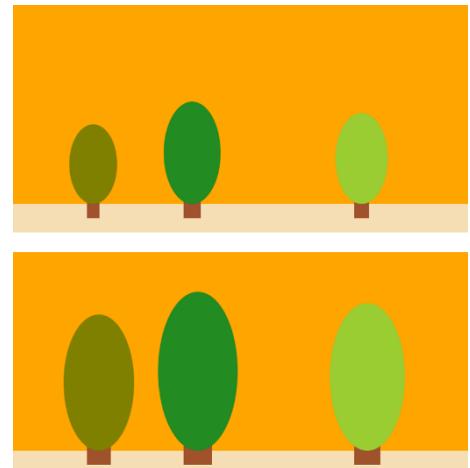
Dit kun je lezen als: maak een nieuwe versie (instantie) van de klasse *Boom* en geef dit object de naam *boom1*. Zorg dat de leeftijd van deze boom *1* is, de kleur van de bladeren '*olive*' en de *x*-positie *130*. Deze drie argumenten verwijzen naar de drie parameters van (de) *constructor* waarmee een nieuwe instantie wordt geconstrueerd. De klasse *Boom* heeft hiermee drie attributen *leeftijd*, *kleur* en *x* gekregen.

Met de klasse kunnen we nu eenvoudig nieuwe instanties maken:

```
boom2 = new Boom(5, 'forestgreen', 300);
boom3 = new Boom(3, 'yellowgreen', 600);
```

De objecten bestaan op dit moment alleen in het geheugen van de computer. Net als in de vorige paragraaf moeten we methodes aanroepen om de objecten op het scherm te tekenen en in dit geval de boom te laten groeien. In figuur 2.51 zie je dat dit nog steeds gaat zoals je in de vorige paragraaf hebt geleerd.

Wanneer gebruik je een klasse? Daar zijn niet alle informatici het over eens. Sommigen vinden dat je alleen een klasse hoeft te maken als je meer *dezelfde* (vergelijkbare) objecten in je programma gebruikt, dus: meerdere instanties van dezelfde klasse. Dat scheelt veel dubbel werk en zorgt ervoor dat je in één handeling alle instanties kunt aanpassen. Anderen stellen dat je altijd een klasse moet maken, ook als je er uiteindelijk maar één object mee maakt. Dit heeft in ieder geval als voordeel dat je werkt met één codestijl voor alle objecten. Mede daarom volgen we in deze module vanaf nu die laatste tactiek.



FIGUUR 2.49

```
class Boom {
    constructor(td, kl, x) {
        this.leeftijd = td;
        this.kleur = kl;
        this.x = x;
    }
    groei() {
        if (this.leeftijd < 20) {
            this.leeftijd++;
        }
    }
    teken() {
        // zie bestand
    }
}
```

FIGUUR 2.50

```
function draw() {
    boom1.teken();
    boom1.groei();
}
```

FIGUUR 2.51

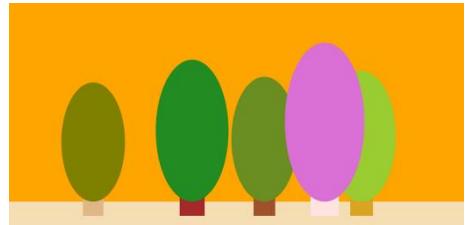


Opdracht 30 de klasse Boom

152. Open *H2O30.js* in jouw *editor*. Dit is de code van voorbeeld 18. Bekijk het resultaat in de *browser*.
153. De bomen stoppen op een zeker moment met groeien. Welke leeftijd hebben ze dan?
154. Voeg twee objecten toe met de namen *boom4* en *boom5*. Kies zelf een geschikte leeftijd, kleur en plek voor de bomen. Natuurlijk zorg je er ook voor dat de bomen op het scherm verschijnen en groeien.
155. Alle stammen hebben nu dezelfde kleur. Welke kleur is dat? Pas de kleur van de stam aan naar *burlywood*.

We willen zorgen dat alle bomen een eigen kleur van de stam kunnen krijgen, zoals in figuur 2.52. Dat kan met de volgende drie stappen:

- Aan de constructor moet een attribuut *kleurStam* worden toegevoegd.
- Bij het aanmaken van instanties van *Boom* moet worden meegegeven wat de kleur van de stam van die (instantie van) *Boom* is.
- Bij het tekenen van de bomen moet gebruik gemaakt worden van het nieuwe attribuut *kleurStam*.

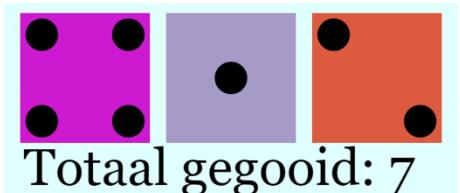


FIGUUR 2.52

156. Voer bovenstaande stappen uit. Kies zelf kleuren voor de stammen van de verschillende bomen.

Opdracht 31 dobbelstenen II

157. Open *H2O31.js* in jouw *editor* en bestudeer de code. Deze bevat het object *dobbelsteen* uit opdracht 24.
158. Pas de code van het object *dobbelsteen* aan naar een klasse *Dobbelsteen*. Zorg er hierbij voor dat bij het aanmaken van een instantie van de klasse alleen het attribuut *x* (de *x*-positie van de dobbelsteen) ingesteld hoeft te worden.
159. Maak een instantie van de klasse *Dobbelsteen* door aan de *setup* de volgende coderegel toe te voegen:
`dob1 = new Dobbelsteen(25);`
160. Pas de coderegels in de *draw* aan zodat *dob1* wordt gegooid en getekend als er met de muis wordt geklikt.
161. Voeg twee nieuwe instanties *dob2* en *dob3* toe en zorg dat alle drie de dobbelstenen tegelijkertijd gegooid en getoond worden.
162. Breid het programma uit, zodat het totaal aantal ogen van de worp op het scherm wordt getoond, zoals in figuur 2.53.



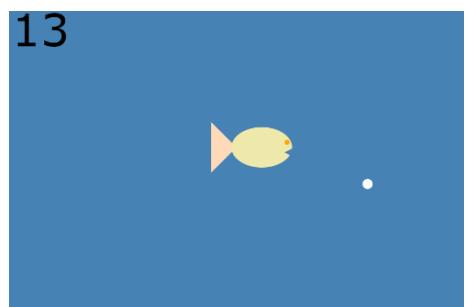
FIGUUR 2.53



Opdracht 32 obfuscator XII: (b)-eet

Bij het spel *(b-)eet* bestuur je een vis die een prooi wil opeten. Lukt dat niet, dan ben je af; lukt het wel, dan kun je een tweede prooi vangen.

163. Bekijk *OBF12* en probeer een aantal prooien te vangen door de pijltoetsen voor naar boven en beneden te gebruiken. Als dat lukt gaat de vis sneller zwemmen en wordt de prooi op een willekeurige (*random*) verticale plek geplaatst.
164. Open *H2O32.js* in jouw *editor* en bestudeer de code. Bekijk het resultaat in de *browser*.
165. Maak het spel af door de volgende stappen te doen:
 - Pas de methode *zwem()* van de klasse *Vis* aan, zodat een vis omhoog en omlaag kan worden bestuurd.
 - Zorg dat garnalen een nieuwe willekeurig gekozen verticale positie krijgt, als hij is opgegeten.
 - Zorg dat de snelheid van gup met 3 toeneemt als hij een garnal opgegeten heeft.



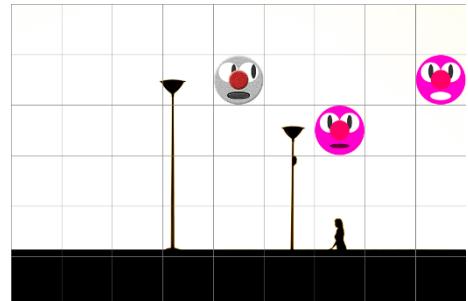
FIGUUR 2.54

Opdracht 33 overloper VIII: werken met een klasse

Vanaf nu willen we voor alle objecten werken vanuit een klasse, dus ook voor de objecten in *overloper*. In deze opdracht hebben we alvast een begin voor je gemaakt. Er is een klasse *Jos* gemaakt. Het object eve is een instantie van de klasse *Jos*. Vanaf nu heet het object waarmee we spelen dus Eve (object eve)! Daarnaast is er een klasse *Vijand* gemaakt. Het object *alice* is een instantie van *Vijand*.

166. Open *H2O33.js* in jouw *editor* en bestudeer de code. Bekijk het resultaat in de browser.
167. Voor het object *raster* is nog geen klasse gemaakt. Schrijf de code om tot een klasse *Raster* en maak hiermee een instantie *raster* (let op het verschil: de klasse is met een hoofdletter, het object is met kleine letter) die wordt gebruikt.
168. Zorg dat het raster weer verschijnt. Bedenk zelf welke aanpassingen hiervoor nog nodig zijn in de *setup* en de *draw*.

Om het spel moeilijker te maken, voegen we een tweede vijand toe met de objectnaam *bob*. Bob lijkt op Alice, maar heeft een andere mond (zie figuur 2.55). De bijbehorende afbeelding vindt je in de map: *images/sprites/Bob100px/Bob.png*



FIGUUR 2.55

169. Voeg het object *bob* toe aan het programma. Zorg er hierbij voor dat hij zijn eigen sprite en startpositie krijgt.
170. Heb je er al aan gedacht dat het spel nu is afgelopen als Jos door Alice of door Bob wordt geraakt? Zo niet, breid dan je programma uit zodat ook aan deze eis is voldaan.

Opdracht 34 overloper IX: een eigen plek

Misschien heb je al gezien dat het in de vorige opdracht mogelijk is dat Alice en Bob zich op dezelfde plek bevinden. Hun afbeeldingen staan dan over elkaar, waardoor het lijkt alsof er nog maar één vijand is. Hoe kunnen we dit voorkomen?

Een mogelijk tactiek is de volgende:

- Laat zowel Alice als Bob bewegen.
- Controleer daarna of Alice en Bob zich op dezelfde plek bevinden.
- Is dat het geval? Vraag dan aan Bob om zich nogmaals te bewegen.

171. Open *H2O34.js* in jouw *editor* en bestudeer de code. Dit is de eindversie van de vorige opdracht.
172. Voeg een if-constructie in de *draw* toe die controleert of Alice en Bob zich op dezelfde plek bevinden. Als dit het geval is, moet *bob.beweeg()* nogmaals worden uitgevoerd.
173. Bedenk minimaal twee bezwaren tegen de gekozen aanpak.

Opdracht 35 overloper X: wie is aan de beurt?

In het spel *overloper* kunnen alle objecten nu tegelijkertijd bewegen. We kunnen er een tactisch spel van maken door om en om Eve (de speler) en haar vijanden Alice en Bob (de computer) een stap te laten zetten en te zorgen dat Eve niet meer naar links (achteruit) kan.

174. Open *H2O35.js* in jouw *editor* en bestudeer de code. Hoe zie je in de code dat Eve niet meer naar links kan bewegen?
175. De klasse *Jos* heeft een nieuw attribuut. Het is een boolean. Wat is de naam van deze boolean? Wat is de beginwaarde?
176. Voeg de code in figuur 2.56 toe aan de *draw* onder *raster.teken()*. Hoe bewegen de drie objecten nu?
177. Welke regel moet op de plaats van *//* worden toegevoegd zodat de objecten om en om bewegen?

```
if (eve.aanDeBeurt) {  
    eve.beweeg();  
}  
else {  
    alice.beweeg();  
    bob.beweeg();  
    //  
}
```

FIGUUR 2.56

2.9 een array van objecten

In de vorige paragraaf heb je kennis gemaakt met klassen. Als je één keer een **klasse** hebt gemaakt, is het heel eenvoudig om een nieuwe **instantie** van die klasse te maken met `new`, zoals met de drie bomen in *voorbeeld 18*.

Door het gebruik van een klasse hoeven we methodes zoals `groei()` en `teken()` slechts één keer te programmeren. Daarna roepen we dezelfde methode voor elke instantie van de klasse `Boom` opnieuw aan, zoals in figuur 2.57.

Toch ziet de code er een beetje onhandig uit. En stel nu eens dat je niet drie maar tien bomen hebt zoals in figuur 2.58? Of zelfs honderd bomen? In dat geval wil je alle bomen in één handeling vragen om te groeien en niet elke boom apart.

Eerder in dit hoofdstuk hebben we gezien dat je een **array** kunt gebruiken om gegevens die bij elkaar horen onder één objectnaam te bewaren. Voor het maken van de animaties (sprites) hebben we al een lijst van objecten (namelijk afbeeldingen) gemaakt. De code om een lijst van objecten van de klasse `Boom` te maken is vergelijkbaar:

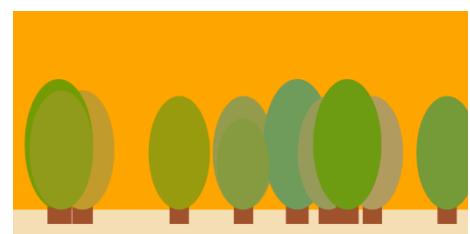
```
var bomen = []; // maakt een lege array

function setup() {
  for (var b = 0; b < 10; b++) {
    bomen.push(new Boom());
  }
}
```

```
function draw() {
  boom1.teken();
  boom2.teken();
  boom3.teken();

  boom1.groei();
  boom2.groei();
  boom3.groei();
}
```

FIGUUR 2.57



FIGUUR 2.58

Met de methode `push` voeg je een element toe aan een lijst. Om het extra beknopt te houden staat tussen de `()` van de methode niet een object, maar de opdracht om een object te maken: `new Boom()`. Met de herhaling (`for`) vraag je nu tien keer om een nieuwe boom aan de lijst met bomen toe te voegen. In *voorbeeld 19* is het hele programma uitgewerkt.

In figuur 2.57 zie je hoe in *voorbeeld 18* één voor één aan alle bomen werd gevraagd om te groeien. Hoe doen we dat, nu we een hele lijst met bomen hebben? We gebruiken opnieuw een `for`-loop om de lijst (met de naam) `bomen` te doorlopen; figuur 2.59 toont de exakte code.

Ter herinnering: met `bomen[3]` duiden we het (vierde!) element uit de lijst aan. In dit geval is dit element een object van de klasse `Boom` met de methodes `groei()` en `teken()`. Omdat de teller `n` loopt van 0 tot en met 9, vragen we hier met slechts een paar regels code aan tien bomen om te groeien en zichzelf te tekenen. En willen we honderd of duizend bomen, dan kan dat nog steeds met deze regels!

```
for(n=0;n<bomen.length;n++)
{
  bomen[n].teken();
  bomen[n].groei();
}
```

FIGUUR 2.59

Opdracht 36 een array van bomen

178. Open `H2O36.js` in jouw *editor*. Dit is de code van *voorbeeld 19*. Bekijk het resultaat in de *browser*.
179. Pas de code aan zodat er niet tien maar twintig bomen worden getekend.
180. Wat verwacht je te zien als aan het begin van de `draw` de coderegel `bomen[10].G = 0;` wordt toegevoegd? Controleer je voorspelling.
181. De bomen kunnen op dit moment 20 jaar oud worden. Pas dit aan naar tien jaar.

Als bomen tien jaar oud zijn, gaan ze dood. Dit betekent voor het programma dat ze niet meer moeten worden getekend.

182. Gebruik een if-constructie in de for-loop van de `draw` om dit te programmeren. Als je het programma uitvoert heb je als het goed is aan het eind geen bomen meer over!

Opdracht 37 knikkerbak I

In voorbeeld 16 is een object bal gemaakt dat tegen de wanden van het canvas weerkaatst. Op basis van de code is een klasse `Knikker` gemaakt waarmee we een grote bak met knikkers gaan maken.

183. Open `H2O37.js` in jouw *editor* en bestudeer de code. Bekijk het resultaat in de *browser*.
184. Op regel 29 is een lege array `knikkerVerzameling` gemaakt. Vul deze lijst met tien instanties van de klasse `Knikker` op de manier die in de theorie wordt beschreven.
185. Gebruik een for-loop in de `draw` om alle knikkers te laten bewegen en te tekenen, zoals in figuur 2.60.



FIGUUR 2.60

Opdracht 38 knikkerbak II: interactie met de muis

We gaan de knikkerbak uit de vorige opdracht uitbreiden zodat het programma reageert op de muis, door gebruik te maken van de boolean `mouseIsPressed` die standaard in Processing zit en waar je al eerder kennis mee hebt gemaakt.

186. Open `H2O38.js` in jouw *editor* en bestudeer de code. Voorspel op basis van de code wat je zult zien als er met de muis wordt geklikt.
187. Bekijk het resultaat in de *browser*. Klopt je voorspelling? Begrijp je wat er gebeurt?

Ten opzichte van de vorige opdracht is het vullen van de lijst met knikkers verandert. In de `setup` staat nu: `knikkerVerzameling.push(new Knikker(random(20, 980), random(20, 280), 'white'));`

188. Welke betekenis hebben de drie parameters die aan `Knikker` worden meegegeven?

Als op de muis wordt gedrukt, willen we bereiken dat er een nieuwe knikker wordt toegevoegd aan de lijst. Deze knikker moet verschijnen op de plek waar de muis zich bevindt als er wordt geknikt.

189. Pas de code aan, zodat dit doel wordt bereikt. Maak de nieuwe knikkers rood, zodat je goed kunt zien welke de nieuwe zijn.

In figuur 2.61 is zojuist één keer met de muis geklikt. Toch zie je meerdere rode knikkers.



FIGUUR 2.61

190. Geef hiervoor de verklaring.

Opdracht 39 knikkerbak III: beweeg naar de muis

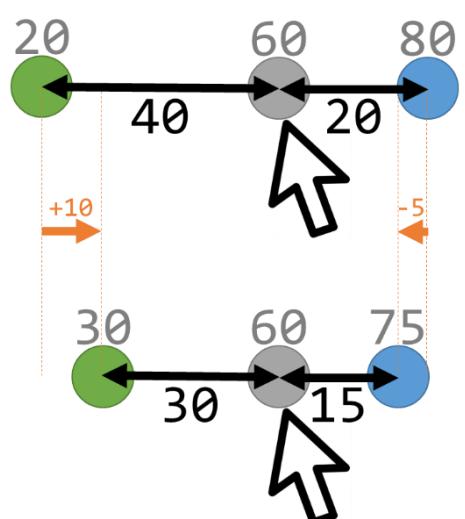
In figuur 2.62 zie je een veel gebruikt principe bij het maken van games: **lineaire interpolatie**. Het wordt onder andere gebruikt om objecten naar elkaar toe te laten bewegen.

In de tekening bevindt de groene knikker zich in eerst op $x = 20$ en de muis op $x = 60$. De onderlinge afstand is dus $60 - 40 = 20$. We willen programmeren dat de groene knikker een stap zet die 25% (kwart) van de onderlinge afstand is. Die stap is dus $40 \cdot 0,25 = 10$.

In een wiskundige formule met variabelen is de grootte van de stap:

$$\text{stap} = (x_{\text{muis}} - x_{\text{knikker}}) \cdot 0,25$$

191. De groene knikker beweegt naar rechts, maar de blauwe knikker beweegt naar links. Klopt de formule ook voor blauw?
192. Open `H2O39.js` in jouw *editor* en bestudeer de code. De klasse `Knikker` bevat een methode `gaNaarMuis` die nu nog leeg is.
193. Vul de methode `gaNaarMuis` zodanig aan dat de knikker zich bij elke aanroep zowel in de x-richting als de y-richting 25% van de onderlinge afstand richting de muis beweegt.
194. Verander het percentage naar 5% en bekijk het eindresultaat.



FIGUUR 2.62

Opdracht 40 overloper XI: een array met bommen

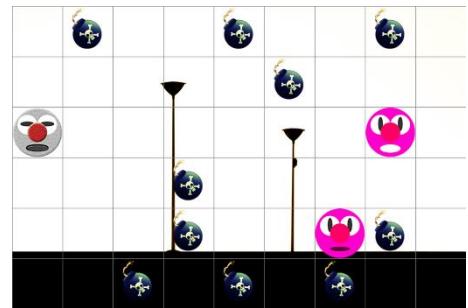
Om het spel *overloper* uitdagender te maken gaan we het raster vullen met een aantal bommen. Als je Eve op een bom laat stappen, ben je af. Het eindresultaat ziet eruit zoals in figuur 2.63, maar we beginnen eerst met één bom.

Voor Eve is een nieuw attribuut `staOpBom` gemaakt. Het is de bedoeling dat deze boolean straks de waarde is `true` krijgt, als Eve zich op de plek van een bom bevindt.

195. Open `H2O40.js` in jouw *editor* en bestudeer de code. Hierin is een nieuwe klasse `Bom` opgenomen, waarmee één instantie is gemaakt.

De `constructor` van de klasse `Bom` bevat de volgende regels:

```
this.x = floor(random(1, raster.aantalKolommen))*raster.celGrootte;  
this.y = floor(random(0, raster.aantalRijen))*raster.celGrootte;
```



FIGUUR 2.63

196. Welke mogelijke waarden hebben `this.x` en `this.y`?
197. Bedenk een mogelijke reden waarom de programmeur ervoor gekozen heeft om voor `this.x` als argument van de functie `random` een `1` in te vullen in plaats van `0`.
198. Op regel 73 is een lege array `bommenArray` gemaakt. Vul deze lijst met tien instanties van `Bom`.
199. Gebruik een for-loop in de `draw` om alle bommen te tonen. Als je nu nog coderegels met `bom1` hebt, moet je die verwijderen.
200. Controleer het resultaat. Hoeveel bommen zie je?
201. Geef **twee** redenen waarom je, als je het spel laadt, niet altijd tien bommen ziet.

Alice en Bob kunnen gewoon over bommen heenlopen, maar als Eve op een bom staat, moet het spel afgelopen zijn. Om dit te bereiken is voor Eve (in de klasse `Jos`) de methode `staatOp` gemaakt, met als parameter de array met bommen. Deze wordt al aangeroepen (met `Eve.staatOp(bommenArray)`), maar bevat nog niet de juiste code. Om dit te bereiken moeten de volgende stappen worden gezet:

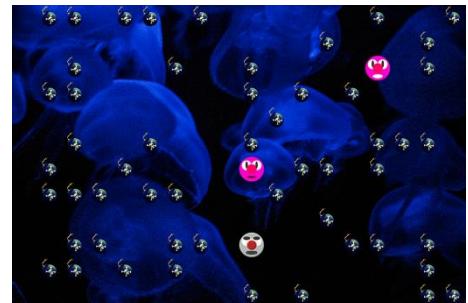
- Maak een for-loop die één voor één de bommen uit de array langsloopt
- Controleer voor elke bom of de positie van Eve overeenkomt met die van de bom
- Is dat zo? Verander dan `staOpBom` in `true`.
- Klaar met de lijst? Geef dan de waarde van `staOpBom` terug.

202. Pas de methode `staatOp` aan op basis van bovenstaande stappen en controleer het eindresultaat.

203. Leg uit dat het in theorie mogelijk is dat er een spel ontstaat dat niet te winnen is.

204. Voer de volgende stappen uit om tot een eindversie van *overloper* te komen, vergelijkbaar met figuur 2.64:

- Pas het raster aan zodat het 18×12 cellen bevat
- Kies voor dit grotere speelveld (216 cellen!) zelf een geschikt aantal bommen.
- Kies zelf een mooie afbeelding als achtergrond
- Zet het tekenen van het raster uit



FIGUUR 2.64

Opdracht 41 obfuscator XIII: bouncer

Een laatste uitdaging in dit hoofdstuk: bij het spel *bouncer* kun je door met je muis te klikken nieuwe ballen laten stuiteren, totdat één van de ballen de grond heeft geraakt. Maar pas op: vervolgens moet jij alle gestuiterde ballen wegklikken, voordat ze voor een tweede keer de grond raakt. Hoeveel risico neem jij?

205. Bekijk *OBF13*. Probeer het spel een paar keer uit.
206. Open `H2O41.js` in jouw *editor* en bestudeer de code. Hierin is al geprogrammeerd dat je nieuwe ballen kunt maken, totdat één van de ballen gestuiterd is.
207. Vul `H2O41.js` aan zodat deze zich net zo gedraagt als *OBF13*.

★ Opdracht 42 overloper XII: overerving

Een belangrijke eigenschap van object-georiënteerd programmeren is **overerving** dat je misschien al kent van het maken van websites met HTML & CSS.

In figuur 2.65 zie je een tekening met vier blokken. De bovenste drie zijn gemaakt met de klasse *Blok* en de onderste is een instantie van de klasse *Spiegel*. Als je de attributen en methodes van beide klassen bekijkt, zie je dat er een grote overlap is tussen deze twee klassen.

Je zou kunnen zeggen dat de klasse *Spiegel* grotendeels een uitbreiding is van de klasse *Blok* met één extra attribuut *randKleur* en één extra methode *keerOm*. Kunnen we *Spiegel* programmeren als uitbreiding van *Blok*?

208. Open *H2O42.js* in jouw *editor* en bestudeer de code. Let in het bijzonder op de drie klassen *Bom*, *Vijand* en *Jos*.

209. Neem figuur 2.66 over en geef (in het linker deel) aan voor welke attributen en methodes er overlap is tussen de drie klassen.

210. Vul het schema verder aan door aan te geven welke attributen en methodes uniek zijn voor de verschillende klassen.

Zowel *Bom*, *Vijand* als *Jos* is een object met een zekere positie en afbeelding in het raster dat met de methode *teken* op het verschijnt. Hiervoor worden drie keer dezelfde coderegels gebruikt. Als oplossing maken we een **superklasse** *rasterObject*. Op basis van de superklasse maken we vervolgens drie nieuwe **subklassen** *Bom*, *Vijand* en *Jos* die daarna hun eigen unieke attributen en methodes krijgen maar de overlappende attributen en methodes erven van de superklasse. Dit heet overerving.

211. De klasse *Bom* overlapt volledig met de andere klassen. Verander daarom de naam *Bom* naar *rasterObject*. Vergeet niet om ook regel 112 (waarin bommen worden gemaakt) aan te passen.

Ten opzichte van *rasterObject* heeft *Vijand* alleen één extra methode *beweeg*. We kunnen daarom een uitbreiding van de superklasse maken met *extends*.

212. Verander regel 14 naar `class Vijand extends rasterObject {`

213. Verwijder alle methodes uit de klasse *Vijand* die ook al in de *rasterObject* staan. Als het goed is houd je alleen code voor de methode *beweeg* over. Stel de juiste werking van het spel vast.

Ten opzichte van *rasterObject* heeft *Jos* niet alleen andere methodes, maar ook een aantal attributen. Hiervoor is, in tegenstelling tot de klasse *Vijand* daarom nog een constructor nodig.

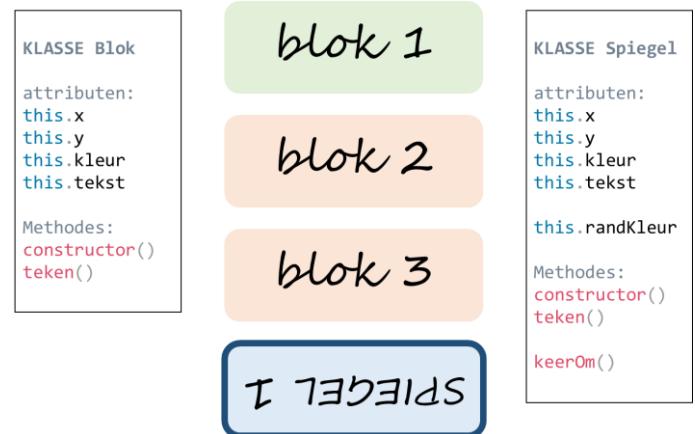
214. Gebruik *extends* om te zorgen dat de klasse *Jos* een subklasse van *rasterObject* wordt.

215. Vervang de constructor door de code in figuur 2.67.

216. Probeer te bedenken wat de regel *super(sprite, stap)* doet.

217. Verwijder alle methodes uit de klasse *Jos* die ook al in de *rasterObject* staan (behalve de constructor). Controleer de juiste werking van het spel.

218. Bij jouw controle heb je gemerkt dat *Jos* niet meer netjes aan de linkerkant van het scherm begint. Breid de constructor uit met regels voor *this.x* uit om dit op te lossen.



FIGUUR 2.65

	Bom, Vijand & Jos	Bom	Vijand	Jos
overlappende attributen		eigen attributen		
overlappende methodes		eigen methodes		

FIGUUR 2.66

```
constructor(sprite,stap) {
    super(sprite,stap);
    this.gehaald = false;
    this.aanDeBeurt = true;
    this.staOpBom = false;
}
```

FIGUUR 2.67

H3 GAME DESIGN

3.1 Inleiding (Wat is een spel?)

Wat is een game? Bij het nadenken over het antwoord op die vraag denk je misschien aan games die je hebt gespeeld zoals *Minecraft*, *Fortnite*, *Overwatch*, *FIFA 19*, *The Legend of Zelda* etc. Dit zijn voorbeelden van uitgebreide, complexe spellen waar grote teams van programmeurs jarenlang aan hebben gewerkt.

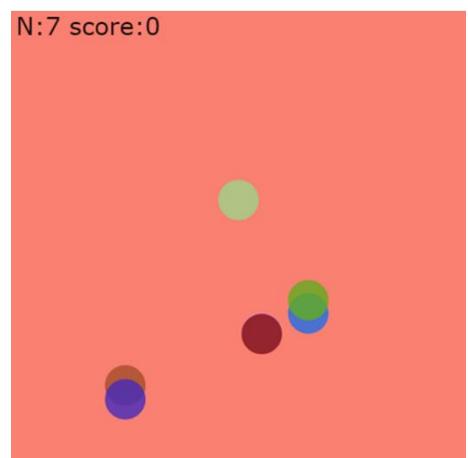
Voorbeeld: aan *The Legend of Zelda: Breath of the Wild* (2017) is vijf jaar lang gewerkt door meer dan 300 ontwikkelaars!

Het kan veel eenvoudiger: ook *Zoekspelletje* (figuur 3.1 boven) en *Bouncer* (onder) uit het vorige hoofdstuk zijn voorbeelden van spellen. Het belangrijkste element dat een game onderscheidt van b.v. een film of een boek is dat je zelf invloed kunt uitoefenen op de uitkomst (interactiviteit). Een paar algemene kenmerken van games:

- speler (-s)
- interactie / keuze (spelers kunnen het spel beïnvloeden)
- spelregels
- uitdaging / doel / competitie
- resultaat / score / uitkomst
- voortgang
- levels / niveau
- terugkoppeling (tekst, beeld, geluid)
- abstractie (versimpeling t.o.v. de echte wereld)
- emotie van de speler (plezier, stress, frustratie, focus)

Spellen zijn er in vele soorten. In dit hoofdstuk concentreren we ons vooral op eenvoudige spellen waarbij behendigheid, strategie / slimheid en geluk (denk ook aan *random*) een rol spelen.

Aan het eind van dit hoofdstuk heb je hopelijk voldoende geleerd om zelf een spel te kunnen bedenken, ontwerpen en programmeren.



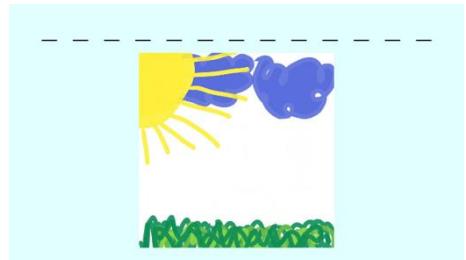
FIGUUR 3.1

Opdracht 1 Nadenken over Bouncer

1. Open *H3O01.js* in de browser (het is niet nodig om de code in jouw *editor* te openen) en probeer het spel te spelen.
2. De spelregels staan niet vermeld. Lukt het je om erachter te komen hoe het spel werkt? Vraag eventueel een medeleerling of je leraar. Geef een beschrijving van de spelregels.
3. In de theorie hierboven staan kenmerken van games opgesomd. Geef voor elk kenmerk aan of, en zo ja op welke manier, dit in het spel naar voren komt.
4. Geef voor onderstaande spelelementen aan of, en zo ja hoe, ze terugkomen in *Bouncer*.
 - a. behendigheid
 - b. strategie / slimheid
 - c. geluk
5. Er is een slimme strategie om tot een hele hoge score te komen. Heb je die manier ontdekt?
6. De *slimme strategie* maakt het spel minder uitdagend dan de spelmaker had bedoeld. Beschrijf een aanpassing (niet in programmeercode, maar in woorden) die deze *slimme strategie* blokkeert.
7. Het spel is eigenlijk niet zo heel goed gemaakt. Er zijn veel verbeteringen mogelijk, zowel qua programmeercode als op het gebied van spelbeleving. Bedenk minimaal vijf aanpassingen om dit spel te verbeteren (zonder dat je de aard van het spel te veel te verandert).
8. Wanneer vind jij een game leuk? Gebruik de kenmerken uit de theorie en vraag 4 in je antwoord.
9. Games worden niet alleen gespeeld omdat ze leuk zijn. Bedenk drie mogelijke doelen van games.

3.2 Het object spel & flowcharts

In voorbeeld 20 kun je het spel *Galgje* spelen. Bij dit spel kiest de computer een woord dat getoond wordt als een reeks streepjes (zie figuur 3.2 boven). De speler moet dit woord vinden door letters te raden. Is een letter goed, dan verschijnt het op de juiste positie (-s) binnen het woord tot het geraden is. Raad je een letter die niet in het woord zit, dan wordt een stukje van een galg getekend. Is de tekening van de galg af, dan heb je verloren (zie figuur 3.2 onder).

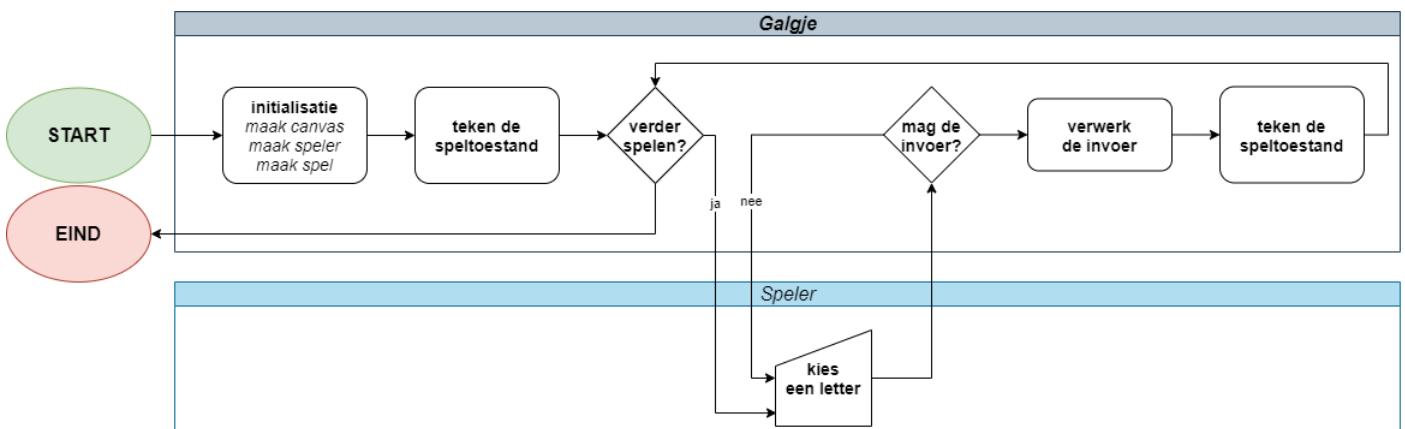


In figuur 3.3 zie je een **flowchart** van *Galgje* waarin de hoofdstructuur van het spel te zien is. Dit diagram bevat de volgende elementen:

- *ellips* geeft het begin of eind van een programma (of proces)
- *afgeronde rechthoek* geeft een proces weer met één of meerdere stappen
- *ruit* geeft aan dat er een beslissing of keuze is
- *trapezium* geeft aan dat er sprake is van een invoer van een gebruiker
- *pijlen* geven aan in welke volgorde stappen worden gedaan



FIGUUR 3.2



FIGUUR 3.3

In de flowchart van figuur 3.3 zijn de stappen onderverdeeld in twee blokken, behorende bij de klassen *Galgje* (of algemener *Spel*) en *Speler*. Verreweg de meeste processen worden door het spel afgehandeld.

Wanneer je een groter spel wilt maken, is het meestal een goed idee om een flowchart te tekenen. Het dwingt je om goed na te denken over vragen als:

- Welke handelingen moeten worden verricht?
- In welke volgorde moet dat gebeuren?
- Wie is verantwoordelijk voor welke handeling?
- Welke objecten zijn (dus) nodig?

De flowchart van figuur 3.3 is gemaakt voor het spel *Galgje*, maar als je er over nadenkt is het diagram toepasbaar op heel veel spellen met één speler:

- **initialisatie** Elk spel (of zelfs: programma) begint met het vastleggen van de begininstellingen (*preload* en *setup*). Denk aan het aanmaken van variabelen, objecten, het canvas en plaatjes.
- **teken speltoestand** Je speelt het spel op een beeldscherm. Er moet dus altijd iets te zien zijn. Wat je ziet is bij een spel afhankelijk van de keuze van de speler. Na elke (geldige) keuze krijgt de speltoestand een update. Daarna moet de nieuwe speltoestand dus weer worden getekend.
- **mag de invoer?** Een spel heeft spelregels die bepalen wat een speler wel en niet mag. Vaak is er een vorm van controle nodig op **geef invoer** (hier: **kies een letter**). Bij een juiste invoer volgt **verwerk de invoer**. Wat wel en niet mag hangt natuurlijk af van de spelregels.
- **verder spelen?** Een spel heeft een begin en een eind. Na elke verandering moet worden vastgesteld of bijvoorbeeld het speldoel is bereikt of dat de speler *af* is.

Wat levert deze structuur op voor de code van het spel? We gaan hier niet de hele code van *Galgje* bespreken, maar benoemen wel een aantal belangrijke en nieuwe punten.

In figuur 3.4 zie je het **hoofdprogramma** van *Galgje*. Het gedeelte met de grijze achtergrond bevat wat extra code, om de afbeeldingen van het spel te laden. De overige code van het hoofdprogramma bestaat verder alleen nog uit de `setup` en `keyTyped`.

Merk op: de loopfunctie `draw` is hier helemaal verdwenen! Het voortdurend *lopen* is hier niet nodig. Kijk nog maar eens naar de flowchart: na elke invoer (dus: `keyTyped`) van de speler verwerkt het spel deze invoer en wacht het simpelweg tot de volgende invoer.

In de `setup` wordt eerst een speler gemaakt (met `new Speler`). Daarna wordt een instantie van *Galgje* gemaakt: `spel = new Galgje(speler, beeldjes)`

Aan deze coderegel kun je zien dat het object `spel` het object `speler` mee krijgt. Binnen `keyTyped` staat dan ook: `spel.speler.kiesLetter();`: de speler **van het spel** wordt gevraagd om de methode `kiesLetter` uit te voeren, als er op een knop van het toetsenbord wordt gedrukt. Als dit een letter uit het alfabet is, vraagt de speler vervolgens aan het object `spel` om `controleerInvoer` uit te voeren, waarin wordt gekeken of het woord niet al volledig is geraden.

Dit is onze eerste kennismaking met een object `spel`. Vanaf nu zullen we alle spellen maken met een object `spel` dat één of meerdere andere objecten (zoals `speler`) zal bevatten.

```
var beeldjes = [];
var aantalBeeldjes = 11;

function preload() {
    for (b = 0; b < aantalBeeldjes; b++) {
        bld = loadImage("galgje(" + b + ").jpg");
        beeldjes.push(bld);
    }
}

function setup() {
    var myCanvas = createCanvas(700, 400);
    myCanvas.parent('processing');
    speler = new Speler('Vincent');
    spel = new Galgje(speler, beeldjes);
    spel.teken();
}

function keyTyped() {
    spel.speler.kiesLetter();
}
```

FIGUUR 3.4

```
class Speler {
    constructor(n) {
        this.naam = n;
        this.resterendeBeurten = null;
    }

    kiesLetter() {
        if (key >= 'a' && key <= 'z') {
            spel.controleerInvoer();
        }
    }
}
```

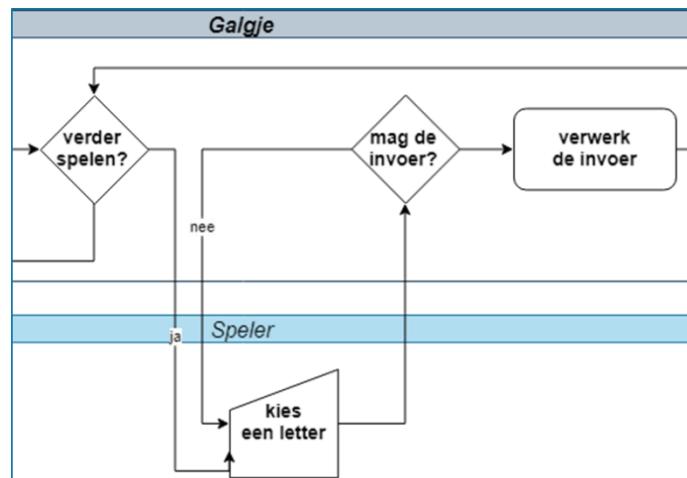
FIGUUR 3.5

Opdracht 2 Galgje I

10. Open *H3O02.js* en speel *Galgje*.
11. Open *H3O02.js* in jouw *editor* en bestudeer de code. Dit is de code van voorbeeld 20. De constructor van de klasse `Speler` bevat: `this.resterendeBeurten = null;` Waar in de code krijgt dit attribuut een waarde? Welke waarde is dat? Hoeveel foute letters mag je dus raden voor je af bent?

In figuur 3.6 zie je een deel van de flowchart van het spel met het keuze-element *mag de invoer*.

12. Hoe heet de methode van de klasse `Galgje` die deze taak uitvoert?
13. Onder welke twee (!) voorwaarden wordt de gekozen letter verder verwerkt?
14. Als je de klasse `Speler` bekijkt, kun je zien dat deze ook al een controle op de invoer uitvoert, voordat het spel zelf de invoer controleert. Welke controle voert het object `speler` zelf al uit?



FIGUUR 3.6

Opdracht 3 Galgje II: inzet van booleans

15. Open *H3O03.js* in de browser en speel het spel *Galgje* (figuur 3.7) dat wordt beschreven in de theorie van de paragraaf.
16. *Galgje* is een spel dat erg bekend is en door heel veel mensen is gespeeld. Bedenk drie kenmerken van het spel die de populariteit zou kunnen verklaren.
17. *Galgje* wordt meestal gespeeld op papier (of het digibord ☺). Benoem een aantal verschillen tussen die manier van spelen en deze Javascript-versie.
18. Bedenk vijf verbeterpunten voor de huidige versie van ons spel. Wat zou er volgens jou veranderd kunnen worden om het spel nog aantrekkelijker te maken? Gebruik eventueel jouw antwoorden van de vorige vragen.
19. Open *H3O03.js* in jouw *editor* en bestudeer de code. Dit is de code van *voorbeeld 20*.



FIGUUR 3.7

Het proces *verwerk de invoer* in de flowchart van het spel (zie figuur 3.3 of figuur 3.6) komt in de code terug als de methode *verwerkInvoer* van het object *spel* (regel 55). De methode sluit af met de code in:

```
if (!letterZitInWoord) {  
    this.speler.resterendeBeurten--;  
}
```

20. Beschrijf deze Javascript-code in een Nederlandse zin.

De variabele *letterZitInWoord* is van het type **boolean**. De methode *woordIsGeraden* bevat eveneens een boolean (geraden), maar geeft bovendien een boolean als antwoord terug met *return*.

21. Op welke twee plaatsen in de code wordt de methode *woordIsGeraden* aangeroepen?
22. Als de speler het woord geraden heeft, verschijnt op dit moment de tekst **GEWONNEN :)**. Pas het programma aan, zodat bij een overwinning de tekst **Goed gedaan, Vincent!** verschijnt.
Maak hierbij gebruik van het attribuut naam van het object *speler*.
23. Pas de *setup* aan zodat de speler jouw eigen naam krijgt en controleer de werking van jouw code.



Opdracht 4 Galgje III: verbeteringen

Bij de vorige opdracht heb je verbeterpunten voor de huidige versie van *Galgje* bedacht. In deze opdracht voeren we de volgende verbeterpunten door:

- De letters die je al hebt geprobeerd, staan in beeld
 - Als je twee keer dezelfde verkeerde letter raadt, kost je dat niet twee keer een beurt
24. Open *H3O04.js* in jouw *editor* en bestudeer de code. Hoe wordt het te raden woord opgesplitst in losse letters die in de array *letters* worden bewaard?
 25. De letters die door de speler worden geprobeerd, worden bewaard in de array *pogingen*. Gebruik deze array om alle letters die zijn geprobeerd op het canvas te tonen onder de tekening, vergelijkbaar met figuur 3.8.



FIGUUR 3.8

De speler van het spel in figuur 3.8 heeft achter elkaar de letters l, e, t, t, e en r geprobeerd. De array *pogingen* bevat daarom zes waarden. De letters t en e komen hierin twee keer voor.

De methode *verwerkInvoer* van het object *spel* controleert al of de letter die wordt geraden in het woord zit. Op een vergelijkbare manier kan worden gecontroleerd of de gekozen letter al in de array *pogingen* zit.

26. Breid de methode *verwerkInvoer* uit zodat aan de tweede eis van deze opdracht wordt voldaan.



Opdracht 5 obfuscator: XIV Galgje IV

In deze opdracht voegen we nog meer verbeteringen toe aan het spel *Galgje*.

- Een teller toont het aantal resterende beurten tenzij je het woord geraden hebt.
 - Als het spel afgelopen is, kun je door op de spatiebalk te drukken een nieuw spel beginnen. Gedurende het spel mag het drukken op de spatiebalk geen effect hebben.
 - Het spel bevat een array met woorden waaruit willekeurig een opdracht wordt gekozen.
27. Bekijk *OBF14*. Probeer het spel uit en stel vast of het aan bovenstaande eisen voldoet.
28. Open *H3O05.js* in jouw *editor* en bestudeer de code. Breid de code uit, zodat aan de eerste eis (resterende beurten) wordt voldaan.
29. De invoer van de speler wordt in eerste instantie verwerkt in *keyTyped*. Breid deze uit met een voorwaarde, zodanig dat *setup* (voor een nieuw spel) opnieuw wordt uitgevoerd op het moment dat op de spatiebalk wordt gedrukt. Natuurlijk mag dat alleen aan het eind van en niet tijdens een spel!
30. Pas de constructor van *Galgje* aan zodat het te raden woord willekeurig wordt gekozen uit een array met woorden.

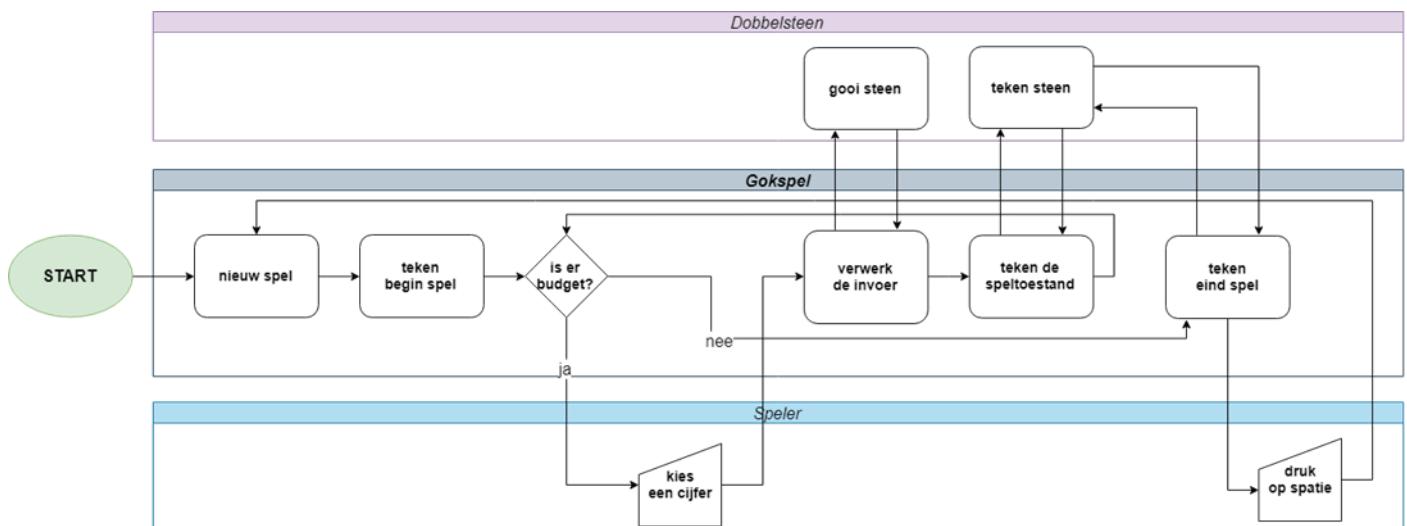
Opdracht 6 Gokspel

In het vorige hoofdstuk hebben we in opdracht 24 een klasse *Dobbelsteen* gemaakt die we nu opnieuw gaan gebruiken voor een spel met maar een paar eenvoudige spelregels:

- De speler begint met een 'budget' van 5.
- De speler raadt hoeveel ogen er zal worden gegooid. Vervolgens gooit het spel de dobbelsteen.
- Als de speler het goed geraden heeft, krijgt hij er 5 punten bij. Is het fout, dan gaat er 1 punt af.
- Je bent af als er geen budget meer over is. In dat geval start je met de spatiebalk een nieuw spel.

31. Open *H3O06.js* in jouw *editor* en speel een aantal keren *Gokspel*.
32. Welke aspecten van dit spel zouden het aantrekkelijk kunnen maken om te spelen?
33. Reageerde je anders op het spel op het moment dat je een worp goed had voorspeld? Waarom?
34. Welk ontwerpelement van het spel zorgt ervoor dat een speler blijft doorspelen?

De programmeur heeft vooraf onderstaande flowchart bedacht met daarin drie (klassen van) objecten:



FIGUUR 3.9

35. Welke overeenkomsten zie je tussen deze flowchart en die van het spel *Galgje* in figuur 3.3?
36. Open *H3O06.js* in jouw *editor*. Op welke punten wijkt de code af van de flowchart in figuur 3.9?
37. Een nieuw spel beginnen mag alleen als het vorige spel is afgelopen. Met welke code wordt voorkomen dat een speler middenin een spel een nieuw spel kan beginnen met de spatiebalk?
38. Regel 101 luidt: `this.speler.budget -= this.strafFout;` Onder welke voorwaarde (-n) wordt deze coderegel uitgevoerd? Beschrijf de betekenis van deze regel in spreektaal.

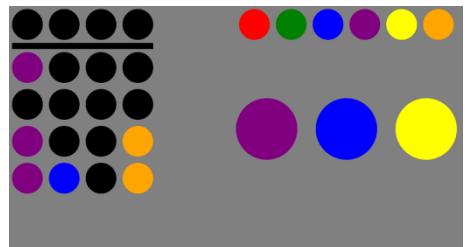
Opdracht 7 Codekraker I

In figuur 3.10 zie je een screenshot uit het spel *Codekraker*.

39. Open *H3O07.js* in de browser (het is niet nodig om de code in jouw *editor* te openen) en probeer het spel te spelen. Probeer de precieze spelregels te ontdekken door het spel te spelen.

40. Beschrijf de spelregels stapsgewijs. De bedoeling is dat je het zo opschrijft, dat een goede programmeur op basis van jouw omschrijving het spel zelf kan programmeren.

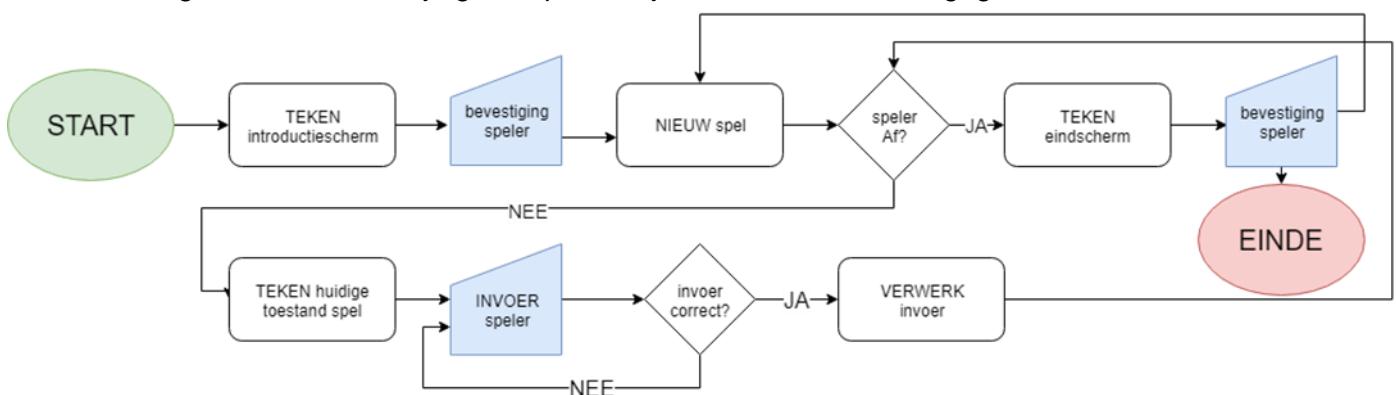
Zie de vorige opdracht voor een voorbeeld van spelregels.



FIGUUR 3.10

Het spel *Codekraker* heeft qua structuur grote overeenkomsten met *Gokspiel*:

Er is een startscherm dat door een actie van de speler verdwijnt. Daarna moet je iets raden. Elke keer dat je iets raadt, controleert het spel of je het goed hebt gedaan. Als je af bent, verschijnt er een eindscherm en kun je ervoor kiezen om nogmaals te spelen. De twee spellen zijn te beschrijven met de algemene flowchart in figuur 3.11. Hierin zijn geen aparte objecten of klassen weergegeven.



FIGUUR 3.11

41. Noem nog twee bestaande spellen die te programmeren zijn volgens het schema van figuur 3.11.

Qua speelervaring lijkt *Codekraker* meer op *Galgje* dan op *Gokspiel*, omdat er niet alleen een factor **geluk** is maar je ook na kunt (moet!) denken over een slimme **strategie**.

42. Vind je dat *het moeten nadenken over een strategie* een spel aantrekkelijker maakt? Leg uit.
43. Welke strategieën kun je bij het spel *Codekraker* inzetten?
44. Welke strategieën kun je bij het spel *Galgje* inzetten?
45. Beschrijf de factor geluk bij zowel het spel *Codekraker* als bij het spel *Galgje*?

Opdracht 8 Nim

46. Open *H3O08.js* in de browser en speel Nim (figuur 3.12).
De spelregels worden uitgelegd op het openingsscherm.

Deze versie van Nim volgt de flowchart van figuur 3.11.

47. Open *H3O08.js* in jouw *editor* en zoek de methode die het proces *invoer correct?* uit de flowchart afhandelt.
Hoe heet de bijbehorende methode? Wat wordt er allemaal gecontroleerd om vast te stellen of de invoer correct is?
48. Heeft de programmeur de munten hier als object geprogrammeerd? Hoe zie je dat?



FIGUUR 3.12

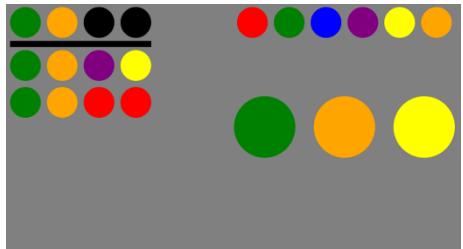
Als je het spel een paar keer hebt gespeeld, kom je erachter dat je altijd verliest van de computer. Dat komt omdat er een **optimale strategie** voor dit spel is: als je de truc kent, wint de tweede speler altijd.

49. Bij welk proces in de flowchart kiest de computer hoeveel munten hij zelf pakt?
50. Zoek de bijbehorende methode in de Javascript-code. Wat is de optimale strategie bij dit spel?



Opdracht 9 Codekraker II: verbeterde feedback

Hoe kunnen we de **speelervaring** of algemener **user experience** van Codekraker verbeteren? In figuur 3.13 zie je een versie van Codekraker die aangepast is ten opzichte van opdracht 7.



FIGUUR 3.13

51. Vergelijk figuur 3.13 met figuur 3.10. Wat is het verschil in feedback op een poging tussen beide spelvarianten?
52. Welke extra informatie levert het spel in figuur 3.13 ten opzichte van figuur 3.10?
53. Open *H3O09.js* in jouw *editor*. Dit is de versie van *Codekraker* van opdracht 7.
54. Navigeer binnen de code naar de methode `controleerPoging` van de klasse `codeKraker` en bestudeer de code van de methode.

De regel `this.pogingen[this.pogingen.length-1].geraden[p] = true;` zorgt er nu voor dat de goed geraden kleuren van een poging op het scherm worden getoond. Bij `true` wordt de echte kleur getekend, bij `false` verschijnt een zwarte cirkel.

55. Zorg dat na een volledige poging alle kleuren van de poging (en dus niet alleen de goed geraden kleuren) op het scherm verschijnen.
56. Zorg dat de zwarte bollen van de geheime code (boven de streep) kleuren, op het moment dat een kleur goed geraden is. HINT: de opdracht is het eerste element van de array met pogingen.

3.3 Acties en gebeurtenissen

Een kenmerk van spellen is **interactiviteit**: de speler kan het verloop van het spel beïnvloeden door een **actie** (*input*) met bijvoorbeeld een toetsenbord, muis of touchscreen. Moderne spelcomputers en telefoons maken ook gebruik van beweging of spraak. In *voorbeeld 21* (toetsenbord), *voorbeeld 22* (muis) en *voorbeeld 23* (touchscreen; zie figuur 3.14) zie je uitwerkingen voor verschillende acties.

De spelbeleving, of algemener de **user experience**, is erg afhankelijk van de manier waarop de speler het spel kan besturen. Besturing gaat niet alleen over de vraag welke hardware (toetsenbord, muis, etc.) er wordt gebruikt. Besturing is ook het aantal handelingen dat tegelijk moet / kan worden gedaan, of de snelheid waarmee je iets moet doen. Zowel het aantal verschillende handelingen als de snelheid van handelen hebben te maken met de **moeilijkheidsgraad** van het spel en de **behendigheid** van de speler. In veel spellen zit een opbouw in moeilijkheidsgraad, bijvoorbeeld door het gebruik van levels. Hier komen we later op terug.



FIGUUR 3.14

```
function touchStarted() {  
    cirkels.push(new  
    Cirkel(mouseX, mouseY));  
}
```

FIGUUR 3.15

Een game moet continu opletten of er een actie van de speler is. Aan alle mogelijke acties is in Javascript een **event listener** gekoppeld, die een **event handler** vraagt om de invoer van de speler *af te handelen*. Deze bevat een stuk code waarin geprogrammeerd is wat er moet gebeuren bij een bepaalde gebeurtenis. Wat dat precies is, is afhankelijk van de **spelregels**. De functie `touchStarted` in figuur 3.15 is een voorbeeld van een *event handler*. In veel bronnen worden de *event listener* en *event handler* niet apart benoemd, maar wordt voor de combinatie van beide de term *event handler* gebruikt. Een actie van een speler is slechts één categorie van gebeurtenissen in een spel. Het kaatsen van een bal tegen een muur is een voorbeeld van een gebeurtenis die het gevolg is van spelregels. In dit voorbeeld moet het spel voortdurend opletten of de bal de muur raakt (de ‘echte’ gebeurtenis: een *event*). Als dit het geval is, moet de bal weerkaatsen (het gevolg van de gebeurtenis), waarbij hij van richting en snelheid verandert, op basis van de spelregels. Het weerkaatsen kun je in één methode programmeren, maar je kunt dit ook opsplitsen in twee methodes: het vaststellen (*Raakt de bal de muur?*) en het verwerken (*Verander van richting*). In deze paragraaf oefenen we met voorbeelden van acties en gebeurtenissen.



Opdracht 10 Acties met het toetsenbord

57. Bekijk voorbeeld 21 in de browser.

Processing bevat de standaard ingebouwde toetsenbord-variabelen key en keyCode die gebruikt kunnen worden bij een event.

58. Druk op verschillende knoppen van je toetsenbord en bekijk de inhoud van de variabelen key en keyCode.
59. Welke verschillen zijn er tussen een ingedrukte letter-knop en een daarna weer losgelaten letter-knop?
60. Hoe heeft de programmeur ervoor gezorgd dat er bij sommige letters een * verschijnt, terwijl bij andere letters de letter zelf op het scherm komt? Welke letters verschijnen wèl in het canvas?
61. Open H3O10.js in de browser en jouw editor. Dit is de code van voorbeeld 20, maar nu alleen met de twee cirkels en zonder teksten.
62. De twee objecten c1 en c2 bewegen met de methode `beweeg1` en `beweeg2`. Leg op basis van de code en je waarneming uit wat het verschil is tussen het gebruik van
`if (keyCode == LEFT_ARROW) en`
`if (keyIsDown(LEFT_ARROW)).`

De code bevat de *event handler function* `keyTyped()` die we al vaker gebruikt hebben binnen deze module. Deze functie wordt bij elke toetsaanslag aangeroepen. Tussen de {} kun je vervolgens aangeven hoe de toetsaanslag moet worden afgehandeld.

63. Beschrijf hoe een toetsaanslag op dit moment wordt afgehandeld in voorbeeld 21. Wat gebeurt er op het niveau van code en variabelen? En wat is het resultaat voor de speler?

Met deze code als basis kunnen we een eenvoudig spelletje maken. Als je één keer de cirkels in beweging hebt gebracht met de pijltjestoetsen is het moeilijk ze weer precies op dezelfde plaats (x-positie) te krijgen. We maken een spel met de volgende eisen:

- De cirkels bewegen met de pijltjestoetsen (zoals dat nu al geprogrammeerd is)
 - Als een letter (of b.v. spatie) wordt ingedrukt, verschijnt een tekst vergelijkbaar met figuur 3.16: Als de x-positie gelijk is staat er *gelukt!* Als dat niet zo is zie je de x-posities van c1 en c2.
64. Pas de code aan, zodat aan bovenstaande eisen is voldaan en speel het spel. (Best lastig, toch?)

Opdracht 11 Tennis I

65. Open H3O11.js in de browser en jouw editor en probeer *Tennis* uit.

Bij het spelen van het spel kun je het volgende waarnemen:

- Als je naar links of naar rechts wilt, moet je steeds opnieuw op de *a* of de *d* drukken: je kunt de knoppen niet ingedrukt houden. Dit komt omdat de *event handler* `keyTyped()` slechts één keer reageert als er op een knop wordt gedrukt en dus ook maar één keer `spel.r.beweeg()` uitvoert.
- Het maakt helemaal niet uit of je de bal raakt of mist: het spel gaat gewoon verder.

66. Verplaats de regel `spel.r.beweeg()` naar het begin van de methode `update` van de klasse *Tennis*. Vergeet niet dat je nu `spel` moet vervangen door `this`.

67. Pas de methode `botsTegenWand` van de klasse *Tennisbal* aan, zodat de bal niet meer stuitert tegen de onderkant van het canvas maar het canvas verlaat wanneer je de bal hebt gemist.

68. Breid de klasse *Tennis* uit met een methode `eindScherm` die een tekst laat zien, als de bal het canvas via de onderkant van het scherm heeft verlaten. Welke extra aanpassingen moet je nog doen om dit te laten werken? HINT: Kijk eventueel terug bij opdracht 6, 7 of 8.



FIGUUR 3.16



Opdracht 12 Tennis II

Hoe kunnen we de spelbeleving van het spel *Tennis I* verbeteren? Als de bal het racket raakt, beweegt deze heel voorspelbaar verder op dezelfde manier als bij het raken van de rand van het canvas. Het zou mooi zijn als de speler dit kan beïnvloeden.

69. Open *H3O 12.js* in de browser en jouw *editor*. Dit is de eindversie van *Tennis I*, met enkele kleine aanpassingen. Je ziet nu bovenaan een *botsingsfactor* (zie figuur 3.17).

We willen bereiken dat de beweging van de bal afhangt van de plaats waar de bal het racket raakt.

70. Navigeer naar de methode *reageerOpRacket* van de klasse *Tennisbal*. Deze bevat drie regels die zijn uitgezet met */* */*. Haal deze weg en speel het spel om te ervaren wat er veranderd is.

De waarde van het attribuut *this.factor* wordt berekend met:

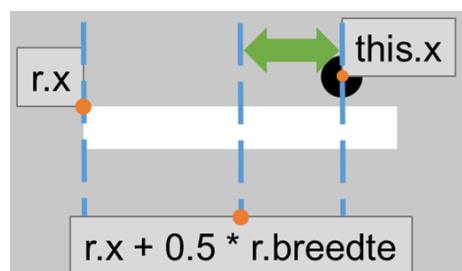
$-4 * ((r.x + 0.5 * r.breedte) - this.x) / r.breedte;$

De groene pijl in figuur 3.18 laat zien wat de betekenis is van:
 $(r.x + 0.5 * r.breedte) - this.x$

71. Wat zijn in theorie de grootste en kleinste waarde van factor?
72. Beschrijf in woorden hoe het raakpunt van de bal op het racket de snelheid van de tennisbal beïnvloedt.



FIGUUR 3.17



FIGUUR 3.18

Het kan handig zijn om op deze manier een bal *met effect* te spelen, maar soms wil je ook gewoon dat de bal op een voorspelbare manier verder beweegt (zoals bij *Tennis I*). Het effect moet in- en uitgeschakeld kunnen worden met ENTER. Als het effect *aan* staat, moet het racket oranje zijn (en anders wit).

73. Breid het spel uit, zodat aan deze eisen wordt voldaan. Gebruik hierbij: *keyCode == ENTER*.

De speler kan nu het horizontale deel van de snelheid veranderen. Deze extra optie heeft het spel iets complexer gemaakt, maar de moeilijkheidsgraad blijft daarna steeds hetzelfde. Daarom willen we dat de *verticale* snelheid van de bal in eerste instantie gelijk is aan *this.basisnelheid* en daarna telkens met 10% toeneemt als deze het racket verlaat. De maximale *verticale* snelheid is 40.

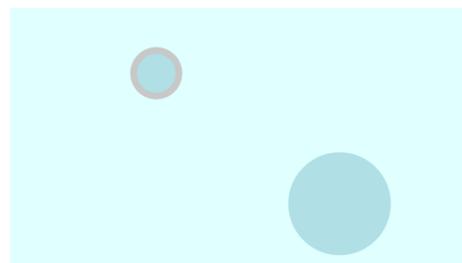
74. Breid het spel uit, zodat ook aan deze twee eisen wordt voldaan.



Opdracht 13 Acties met de muis

75. Bekijk *voorbeeld 22* in de browser. Probeer alle functies van je muis uit en bekijk hun invloed.

Processing bevat de standaard ingebouwde muis-variabelen *mouseX* en *mouseY* die de muispositie bijhouden en de boolean *mouseIsPressed* die alleen *true* is als een muisknop ingedrukt is (en dus weer *false* wordt als je de knop loslaat). Je kunt *mouseIsPressed* gebruiken om de gebeurtenis *er wordt geklikt* af te handelen (b.v. met een *if*-constructie). Verder zijn er voor de muis een aantal *event handlers* beschikbaar: zie de *reference* van P5.



FIGUUR 3.19

76. Open *H3O 13.js* in de browser en jouw *editor*. Dit is de code van *voorbeeld 22*, maar nu alleen met de twee cirkels en zonder teksten. Welke *event handlers* zijn er in de code gebruikt?

We maken een spel met deze code. Doel is om de twee cirkels dezelfde kant op te laten bewegen met dezelfde snelheid, x-positie en diameter. Als dat is gelukt, heb je gewonnen.

77. Maak dit spel. Zorg dat de tekst *gelukt!* verschijnt op een groene achtergrond als je gewonnen hebt.

Opdracht 14 Tennis III

78. Open *H3O14.js* in de browser en jouw *editor*. Dit is de eindversie van *Tennis II* van opdracht 12.

In voorbeeld 22 wordt een aantal voorbeelden getoond van muis-events. Gebruik dit voorbeeld en opdracht 13 voor de volgende opdrachten.

79. Het spel begint nu als je op de spatiebalk drukt. Zorg dat het spel begint bij een muisclick.

80. Zorg dat het racket alleen kan bewegen door met de muis te slepen. Hierbij moet de x-positie van de muis overeenkomen met het midden van het racket.

Waarschijnlijk heb je jouw oplossing nu zo gemaakt, dat je het racket vanuit elke positie kunt slepen. Het maakt niet uit waar de muis is ten opzichte van het racket. Als je een sleepbeweging maakt aan de linkerkant van het scherm, terwijl het racket helemaal rechts staat, dan schiet het racket ineens naar links. We willen dat voorkomen met een extra spelregel of eis:

Alleen als de cursor van de muis het racket raakt, mag je het racket slepen.

81. Breid de klasse *Racket* uit met een methode *raaktMuis* die *true* antwoordt als de muis het racket raakt (en anders *false* terug geeft). Gebruik deze methode om aan de beschreven eis te voldoen.

Het spel bevat een instelling om de bal *met effect* te kunnen slaan die je met ENTER in- en uit kunt schakelen. We willen bereiken dat *metEffect* aan en uit kan gaan door het wielje van de muis te draaien.

82. Pas jouw programma aan zodat aan deze eis wordt voldaan.

Om het spel netjes af te maken verrichten we nog twee stappen:

83. Pas de tekst van het beginscherm aan, zodat deze weer klopt met de doorgevoerde aanpassingen.

84. Breid de klasse *Spel* uit met een boolean-attribuut *afgelopen*. Gebruik dit attribuut om te zorgen dat je, als je af bent (dus als *afgelopen waar is*), een nieuw spel kunt beginnen door met de muis te klikken. In dat geval moet *setup* opnieuw worden uitgevoerd.

Opdracht 15 Acties met een touchscreen

85. Bekijk voorbeeld 23 op een device met een touchscreen (digibord, smartphone, tablet). Raak het scherm een aantal keren aan.

86. Wat gebeurt er als je een vinger over het scherm sleept? Welke *event handler* reageert dan?

87. Wat gebeurt er als je meerdere vingers tegelijk op het scherm zet?

Processing heeft een ingebouwde variabele *touches*. Dit is een array waarin *live* wordt bijgehouden op welke plaatsen het scherm wordt aangeraakt. Omdat dit op meer dan één plek tegelijk kan zijn, is *touches* een array.

De coördinaten van een aanraakpunt vind je met *touches[0].x* en *touches[0].y* voor het eerste aanraakpunt. Voor het tweede aanraakpunt vervang je de 0 door 1, etc.



FIGUUR 3.20

88. Wat bepaalt in voorbeeld 23 de kleur van de cirkels?

89. Als je het scherm loslaat, blijft er altijd maar één cirkel staan (ook als je met meerdere vingers het scherm hebt aangeraakt). Welke cirkel blijft staan? Waar wordt dit door bepaald?

90. Open *H3O15.js* in de browser en jouw *editor*. Dit is de code van voorbeeld 23.

91. Welke *event handlers* zijn er in de code gebruikt?

Als je het scherm een flink aantal keren aanraakt, zijn er volgens de tekst veel cirkels (zie figuur 3.20).

92. Hoe wordt het op het scherm getoonde aantal berekend?

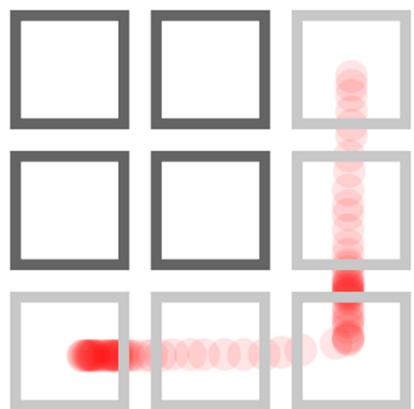
93. In de *draw* staan drie regels die zijn uitgezet met */* */*. Haal dit weg zodat de regels weer worden uitgevoerd en stel vast wat het resultaat is.

94. Eén van de drie regels luidt: *cirkels.splice(c, 1);*. Zoek uit wat deze regel betekent. Gebruik b.v. : <https://javascript.info/array-methods> of https://www.w3schools.com/js/js_array_methods.asp.

Opdracht 16 Login-schermpuzzel

Op veel mobiele telefoons kun je inloggen door een patroon op je scherm te *swipen*. Dit heeft als doel om jou (-w telefoon) te beschermen, maar je kunt het ook zien als een puzzelspel. In deze opdracht bekijken we een eenvoudige versie, gemaakt met de touchscreen-functies van Processing.

95. Open *H3O16.js* op een device met een touchscreen (digibord, smartphone, tablet) en probeer de inlogcode te kraken.
96. Open *H3O16.js* in jouw *editor* en navigeer naar de globale variabele patroon. Had je de code al gekraakt? Zo niet, dan lukt het vast met deze informatie.
97. Maakt het uit in welke volgorde je het patroon tekent?
98. Welke *event handler* geeft de opdracht om te checken of de ingevoerde code juist is?
99. Leg uit waarom het veiliger is om de juistheid van de ingevoerde code niet te checken binnen `touchMoved()`.
- 100.★ Breid het programma uit, zodat er een code van 4×4 moet worden gekraakt.



FIGUUR 3.21

Opdracht 17 obfuscator: XV Racer

In figuur 3.22 zie je een screenshot van *Racer*: een touchscreen-spel waarbij je een auto over een circuit heen moet loodsen, zonder de zwarte randen te raken.

101. Bekijk *OBF15* op een device met touchscreen en speel *Racer*.
102. Open *H3O17.js* in jouw *editor* en bestudeer de code. Bekijk het resultaat in de *browser*. Het scherm toont telkens opnieuw een willekeurig gekozen speelkaart.



FIGUUR 3.22

De klasse `Auto` bevat een methode `aangeraakt` en de klasse `Cel` een methode `wordtGeraakt`.

103. Wat is het doel van de methode `aangeraakt`? Onder welke voorwaarden antwoordt hij `true`?

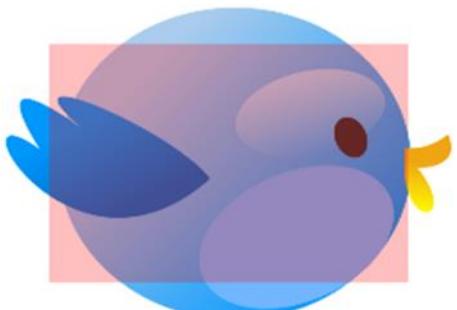
Op dit moment kun je met de auto straffeloos over alle cellen rijden. De methode `wordtGeraakt` moet vaststellen of speler een zwart stuk van het parcours raakt en `true` antwoorden als dit het geval is.

104. Pas `wordtGeraakt` aan zodat aan deze eis wordt voldaan.
105. Pas het spel verder aan, zodanig dat het parcours eruit ziet zoals in figuur 3.22 en speel *Racer*.

Opdracht 18 Bluebird I

Je hebt vast weleens *Flappy Bird* gespeeld. In deze opdracht maken we de variant *Bluebird*.

106. Open *H3O18.js* in de browser en jouw *editor* en speel *Bluebird*.



FIGUUR 3.23

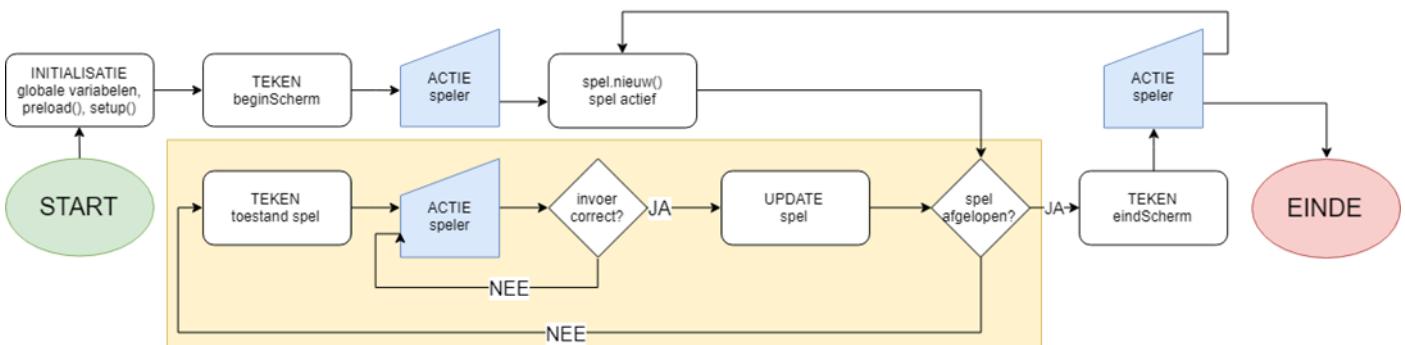
Dit spel maakt gebruik van afbeeldingen. Dat maakt het lastiger om vast te stellen of de vogel een obstakel raakt dan met een cirkel of rechthoek. De programmeur heeft dit opgelost met een **hitbox**: een rechthoekig gebied (figuur 3.23) waarmee wordt bepaald of het 'raak' is. Vanaf de rand van de afbeelding wordt een marge aangehouden, zodat je alleen af bent als je een obstakel met het gekleurde deel raakt

107. Bestudeer de methode `raakt(vogel)` van de klasse `Obstakel`. Klopt de reeks met voorwaarden?
108. Zorg dat het spel de marge nog steeds gebruikt, maar de hitbox (het rode vlak) niet meer laat zien.
109. Pas het spel aan zodat het met het toetsenbord kan worden gespeeld. Gebruik spatie om te vliegen en Enter om het spel te beginnen.

3.4 Vaste patronen: design patterns

In paragraaf 3.2 hebben we **flowcharts** gebruikt bij het ontwerpen van spellen. Je hebt daar kunnen zien dat spellen die op het eerste gezicht heel verschillend zijn, qua flowchart erg op elkaar kunnen lijken. We kunnen spellen groeperen op basis van hun ontwerppatroon. Zo'n algemene structuur heet in het Engels een **design pattern**. Dit begrip wordt heel breed ingezet binnen de informatica. Binnen het vakgebied *software engineering* (het maken van programma's) wordt een *design pattern* omschreven als: "een algemene oplossing voor een veelvoorkomend ontwerpprobleem."

Hiermee wordt niet een oplossing in programmeercode bedoeld. Het is een algemene beschrijving van de oplossing van het probleem die je in meerdere situaties (voor ons: meerdere spellen) kunt gebruiken.



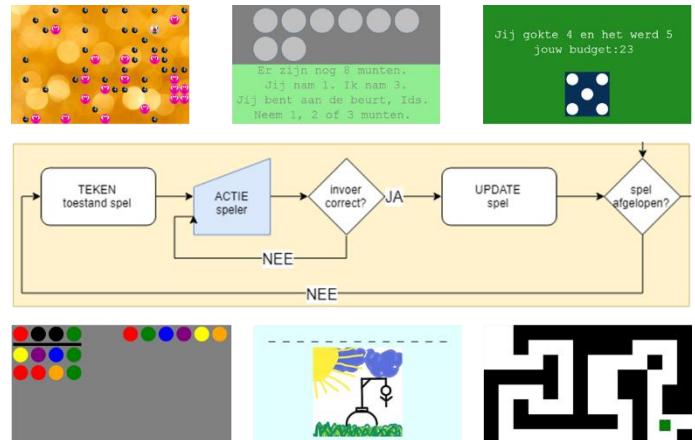
FIGUUR 3.24

Als je eerst nadenkt over de vraag welk ontwerppatroon bij jouw spel past, dan kun je dit patroon als basis gebruiken voor het ontwerp van je spel. In figuur 3.24 zie je een flowchart van een vaste structuur die veel elementen bevat die we al eerder hebben toegepast in spellen. Zo biedt dit patroon de mogelijkheid om een nieuw spel te beginnen als het spel is afgelopen. In deze paragraaf gaan we vooral kijken naar het geel gemarkeerde deel waar het spel 'echt' wordt gespeeld.

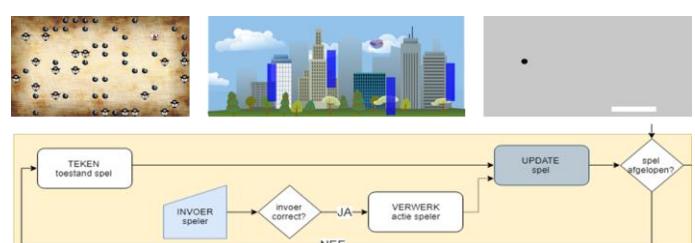
In figuur 3.25 zie je dit gele deel nogmaals met zes screenshots van spellen die gemaakt kunnen worden met dit ontwerp. Voor deze spellen geldt: De speler doet een actie, het spel doet een *update* en toont daarna de nieuwe speltoestand. Toch is er een subtiel verschil tussen de drie spellen boven de flowchart en de drie spellen onder de flowchart:

Bij de onderste drie spellen betekent een *update* alleen dat de invoer wordt verwerkt en er wordt gekeken of het spel gewonnen of verloren is. Bij de bovenste drie spellen voert het spel tijdens de *update* zelf ook een actie uit, die de **speltoestand** beïnvloedt. Tijdens het spelen krijg je te maken met een nieuwe situatie of opdracht, terwijl bij de onderste drie spellen de opdracht steeds hetzelfde blijft. Dat is van grote invloed op de **speelervaring**.

Bij de spellen in figuur 3.26 verandert de speltoestand ook als de speler niets doet. Dat heeft grote invloed op de **user experience**. Zo valt de vogel in het spel *Bluebird* naar beneden als de speler niets doet. De speltoestand verandert dus voortdurend. Hiervoor gebruik je bij dit type spellen de loopfunctie **draw** als **game-loop**.



FIGUUR 3.25



FIGUUR 3.26

In deze paragraaf bekijken we deze en andere vaste spelstructuren of *design patterns*.



Opdracht 19 Varianten van Overloper

Het spel *Overloper* liep als een rode draad door hoofdstuk 2. In voorbeeld 24 en voorbeeld 25 kun je het spel spelen.

110. Speel beide varianten van *Overloper*.
111. Noem een voordeel dat je als speler hebt bij voorbeeld 24 ten opzichte van voorbeeld 25.
112. Andersom: noem een voordeel dat je als speler hebt bij voorbeeld 25 t.o.v. voorbeeld 24.
113. Omschrijf het verschil in *user experience* tussen de twee spellen, zoals jij dat ervaart.
114. In de variant van voorbeeld 24 volgt alleen een *update* als de speler een actie uitvoert. Hierbij wordt twee keer gekeken of de speler een bom of een vijand raakt. Leg uit waarom dit noodzakelijk is.

We gaan wat dieper in op de variant van voorbeeld 25.

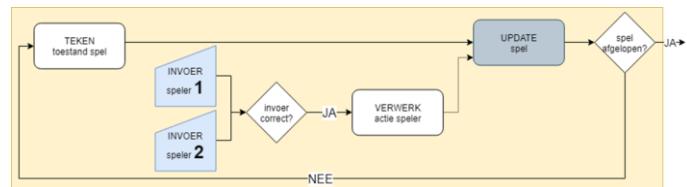
115. Open *H3O19.js* in jouw *editor* en bestudeer de code. Dit is de code van voorbeeld 25 (figuur 3.27).
116. Navigeer naar het hoofdprogramma. Hoe kun je aan de code zien dat het bij het begin van het spel niet uitmaakt op welke toets je drukt?
117. Hoe heeft de programmeur het tempo waarmee de vijanden bewegen geprogrammeerd?
118. In de procedure *update* van de klasse *Overloper* worden drie hoofdstappen uitgevoerd. Welke?
119. Maakt het uit in welke volgorde deze drie stappen worden uitgevoerd?



FIGUUR 3.27

Opdracht 20 Multi-player simultaan: Tennis IV

Tot nu toe zijn alle spellen in deze module van het type **single-player**: je speelt het spel alleen. Veel grote spellen zijn **multi-player** games. Dit zijn spellen die je samen kunt spelen of online met grote groepen. Wij beperken ons hier tot spellen die je met z'n tweeën op één computer kunt spelen. Het onderwerp netwerk-games behandelen we niet.



FIGUUR 3.28

In deze opdracht kijken we opnieuw naar het spel *Tennis*, maar nu voor twee gelijkwaardige spelers. Het is gemaakt volgens de flowchart in figuur 3.28.

120. Bestudeer de flowchart. Wat is het verschil ten opzichte van figuur 3.26?
121. Open *H3O20.js* in de browser en jouw *editor* en speel *Tennis* met je buurman of buurvrouw.
122. Je speelt nu tegen elkaar. Beschrijf hoe dit de manier waarop je het spel beleeft verandert ten opzichte van *alleen spelen*.
123. Wat betekent het woord *simultaan*? Bedenk waarom dit spel **simultaan** kan worden genoemd.

Als je de code van dit spel bestudeert, zie je dat deze grotendeels overeen komt met *Tennis III*. De belangrijkste verandering zit hem in het feit dat er nu twee spelers zijn.

124. Bekijk de constructor van de klasse *Speler*. Hoe wordt onderscheid gemaakt tussen de twee spelers? Hoe komen ze aan hun eigen kleur en plek?
125. Leg uit dat het ook voor de klasse *Tennisbal* uitmaakt met welke speler hij te maken heeft. Beschrijf hoe de bal anders moet reageren op de linker speler vergeleken met de rechter speler.
126. In welke methode van welke klasse wordt bepaald welke speler er gewonnen heeft?
127. Als het spel is afgelopen, kun je met een muisklik een nieuw spel beginnen. Wie heeft de bal uit? Hoe wordt dat bepaald?
128. Hoe vaak is de bal overgespeeld? Breid de klasse *Tennis* uit met het attribuut aantalSlagen dat het aantal slagen telt. Zorg dat dit aantal op het eindscherm wordt getoond.

Opdracht 21 Multi-player sequentieel: Vier op 'n rij

129. Open *H3O21.js* in de browser en jouw editor en speel *Vier op 'n rij* met je buurman of buurvrouw.

130. Wat betekent het woord **sequentieel**? Bedenk waarom dit spel **sequentieel** kan worden genoemd.

Je speelt nu tegen elkaar, net als bij *Tennis* (IV) in de vorige opdracht. Toch is de *user experience* bij dit spel heel anders dan bij *Tennis*.

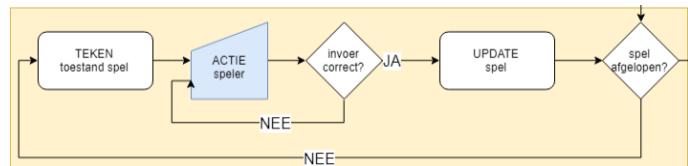
131. Beschrijf overeenkomsten en verschillen in de manier van spelen tussen beide spellen.

132. Noem een voorbeeld van een ander (bord-) spel dat sequentieel kan worden genoemd.

Een sequentieel spel speel je om de beurt. Daarom worden deze spellen in het Engels aangeduid als **turn-based** (*turn* is Engels voor beurt). Een spel als *Tennis* is van de categorie **simultaan** (tegelijkertijd) of **real-time**. Beide spelers kunnen dan voortdurend tegelijkertijd het verloop van het spel beïnvloeden.

133. Noem een voorbeeld van een ander spel dat *simultaan* kan worden genoemd.

In figuur 3.29 zie je dezelfde flowchart als in figuur 3.25. *Vier op 'n rij* kan met deze flowchart worden beschreven als we **ACTIE speler** zien als de actie van de speler die aan de beurt is: de actieve speler.



FIGUUR 3.29

Het spel kan niet controleren of het wisselen van de beurt goed gaat: beide spelers gebruiken de muis voor hun invoer. Een speler zou 2x kunnen klikken.

134. Bestudeer de code van *Vier op 'n rij*: wat controleert het spel wèl met **controleerZet**?

135. hoe komt het wisselen van de beurt in de code terug?

136. De spellen in figuur 3.25 speel je in je eentje. Toch zou je bij die spellen kunnen spreken van een tweede speler. Wie is die tweede speler (bij voorbeeld *Over/oper*)?

In de huidige versie van het spel staat wel een **draw**, maar deze heeft eigenlijk geen functie, omdat de enige regel met **//** in commentaar gezet is.

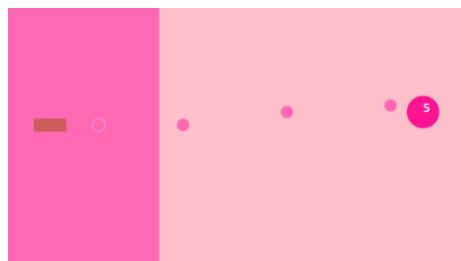
137. Beschrijf wat er verandert aan de *user experience*, als de **//** in de **draw** worden weggehaald.

Het spel stelt op dit moment vast of er een winnaar is, maar dit spel heeft ook de mogelijkheid van gelijk spel.

138. Pas de code aan, zodanig dat ook een gelijk spel kan worden afgehandeld.

Opdracht 22 Multi-player: Hit me!

Bij het spel *Tennis* spraken we in opdracht 20 van **gelijkwaardige spelers**. Daarmee bedoelen we hier niet dat de spelers hetzelfde niveau van vaardigheid hebben, maar dat ze dezelfde **rol** hebben: ze hebben dezelfde opdracht en dezelfde middelen om die opdracht uit te voeren. Zulke spellen worden **symmetrische spellen** genoemd.



FIGUUR 3.30

In deze opdracht kijken we naar het spel *Hit me!* uit de categorie **asymmetrische spellen**. Hier zijn **ongelijkwaardige spelers**. Er zijn hier twee rollen die vergelijkbaar zijn met *jager* en *prooi*.

139. Open *H3O22.js* in de browser en jouw editor en speel *Hit me!* met je buurman of buurvrouw.

140. Bedenk een aanpassing die het voor de schutter (klasse **Kanon**) makkelijker maakt om te winnen.

141. Bedenk een aanpassing die het voor de vijand (klasse **Vijand**) makkelijker maakt om te winnen.

142. Bestudeer de code. Probeer de attributen van de aanwezige klassen zodanig aan te passen dat het ongeveer één minuut duurt om de vijand uit te schakelen als jij met je buurman of buurvrouw speelt.

Op dit moment kan de vijand zien hoeveel levens hij nog heeft.

143. Breid het spel uit, zodat aan de kant van het kanon ook te zien is hoeveel kogels er zijn afgevuurd.

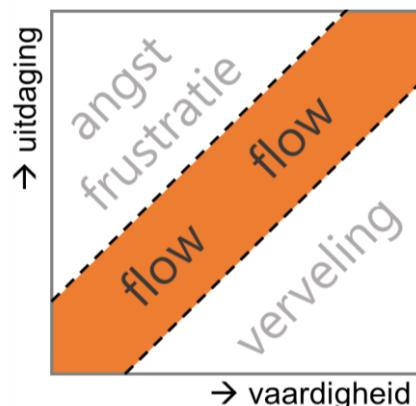
3.5 Uitdaging en beloning

Er zijn spellen die je slechts een paar keer (of zelfs maar één keer) speelt en spellen die je met plezier vaker speelt. Wat maakt die spellen aantrekkelijk? Wat zorgt ervoor dat je niet wilt stoppen met spelen of dat je het meteen opnieuw wilt proberen als je af bent?

In de eerste plaats moet een spel voldoende **uitdaging** bieden. Als een spel te makkelijk is, vind je het al snel saai. Is het te moeilijk dan kom je *niet op gang* met het spel en haak je gefrustreerd af. Het is belangrijk om snel de juiste **moeilijkheidsgraad** te vinden: een balans tussen die uitdaging en de vaardigheden van de speler (Engels: *balancing*).

Grote games zijn ontworpen om je in een **flow** te krijgen. Als je in een flow zit, ben je maximaal gemotiveerd en betrokken en word je volledig in het spel gezogen, doordat het spel voor jou precies de juiste uitdaging biedt. In figuur 3.31 is dit weergegeven in een grafiek.

Naar verloop van tijd zal je vaardigheid toenemen. De moeilijkheidsgraad of uitdaging van het spel moet dan in de juiste mate toenemen om jou in een flow te houden. Een veelgebruikte manier hiervoor is het gebruik van **levels**. Realiseer je dat spelers niet allemaal hetzelfde talent hebben; *het juiste uitdagingsniveau* verschilt!



FIGUUR 3.31

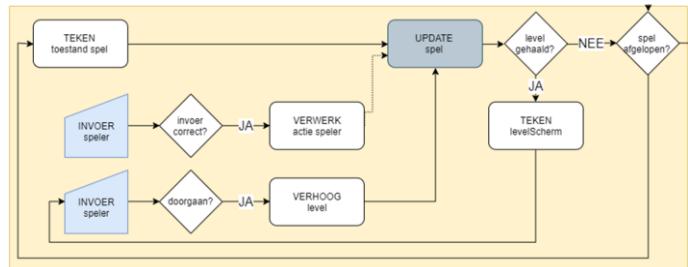
In de tweede plaats moet een spel zorgen dat de speler regelmatig een **beloning** krijgt. Een spel heeft een **doel** of een **opdracht**. Het halen van een doel of level geeft een gevoel van voldoening. Bij veel spellen neemt gedurende het spel (ook tijdens een level) de **score** toe. Het verhogen van de score zorgt met regelmaat voor een succeservaring. Varianten hierop zijn een *aantal levens* of een afnemende *health*. En hoe voelt het als jouw klasgenoten allen een score van onder de 100 halen en jij als enige 101 scoort?

Er zijn vele manieren om met uitdaging en beloning een flow te creëren. De basis hiervoor ligt vast in de **spelregels**. Verandering en verrassing zorgen voor nieuwsgierigheid bij een speler, evenals een verhaal. Een gamewereld met aantrekkelijk beeld en geluid inspireert en zorgt voor emoties en (positieve) prikkels. Probeer je maar eens in te beelden welk effect het op jou heeft als je bij het halen van een score een simpel geluidje hoort. Misschien schrik je er wel van hoe makkelijk je jezelf laat beïnvloeden!

De *user experience* wordt bepaald door wat het spel teruggeeft: dit heet terugkoppeling of **feedback**. Natuurlijk moet de speler hierbij het spel in voldoende mate kunnen beïnvloeden door interactie met b.v. muis, toetsenbord, controller of stem. In deze paragraaf stellen we ons de vraag hoe we de mate van uitdaging – de moeilijkheidsgraad dus – kunnen aanpassen door gebruik te maken van levels. Daarnaast experimenteren we met het ontwerpen van beloning door het inbouwen van feedback via levels en score. We gaan niet uitgebreid in op zaken als vormgeving en verhaallijnen, omdat ze erg veel tijd kosten.

In *voorbeeld 26* zie je een voorbeeld van het gebruik van levels. In dit voorbeeld verschijnt er een tussenscherm als het level is gehaald. In figuur 3.32 zie je een uitbreiding op de flowchart van figuur 3.26 waarin de levels zijn opgenomen.

Na elke spelupdate van de **game-loop** wordt nu niet alleen gecheckt of het spel afgelopen is, maar ook of een level gehaald is. Als een level gehaald is, verschijnt eerst een scherm, waarna er wordt gewacht op de invoer van de speler. Merk op: dit onderscheidt de onderste **INVOER SPELER** van de bovenste **INVOER speler**, waarbij het spel ook verder gaat, als er geen invoer is.



FIGUUR 3.32

Er zijn ook spellen waarbij er geen tussenscherm verschijnt als je een level haalt. Deze spellen gaan continu verder, maar verhogen als een doel gehaald is wel de moeilijkheidsgraad (en soms expliciet het level). *Voorbeeld 27* is hiervan een voorbeeld, maar bijvoorbeeld ook *(b)-eet* (opdracht 32 van hoofdstuk 2) en het dinosaurus-spel dat je in Google Chrome kunt spelen (als je offline bent) vallen in die categorie.

Opdracht 23 Jumper I: speelervaring

In figuur 3.33 zie je een screenshot van het spel *Jumper*. We gaan nadenken over de de *user experience* van dit spel.

144. Open *H3O23.js* in de browser en speel *Jumper* minstens 5x.
145. Is vooraf duidelijk wat het doel is van het spel? En is duidelijk wanneer je gewonnen hebt?
146. Bespreek met een klasgenoot de mate van uitdaging m.b.t.:
 - a. de besturing van het object
 - b. het landen en blijven staan op een platform
 - c. het halen van een level
147. Welke rol speelt willekeur of kans (*random*) in dit spel?

Als je een level hebt gehaald, ga je naar een volgend level.



FIGUUR 3.33

148. Op welke manier is gezorgd dat de moeilijkheidsgraad toeneemt?
149. Bedenk drie andere manieren waarop de moeilijkheidsgraad bij een volgend level zou kunnen toenemen.

In figuur 3.31 zie je een diagram met uitdaging versus vaardigheid.

150. Lukte het om het spel bij de eerste poging meteen uit te spelen? Of andersom: had je veel pogingen nodig om voor het eerst level 1 te halen? Waar zat jij in dit diagram op het moment dat...
 - a. ... je voor het eerst *Jumper* speelde?
 - b. ... je voor het eerst level 3 haalde?
 - c. ... je het spel vijf keer had geprobeerd?

Als je een spel vaker speelt, wordt je vaardiger. Er is sprake dan van een **leereffect**.

151. Heb je in dit geval een leereffect ervaren?
152. Bij deze vraag gaan we ervan uit dat je het spel niet bij de eerste poging wist uit te spelen.
 - a. Hoe voelde het om na een aantal pogingen voor het eerst een hoger level te halen?
 - b. Leverde dat ook een ander gevoel op tijdens het spelen van dat hogere level?
 - c. Heb je de neiging om het graag opnieuw te willen proberen als je af bent?
 - d. Hangt je gevoel af van hoe ver je gekomen bent?

Het spel is zodanig geprogrammeerd dat je, als je af bent, weer helemaal overnieuw moet beginnen bij level 1. Dat kan er voor zorgen dat je in figuur 3.31 in het gebied van de *verveling* terecht komt. Dit is te voorkomen door het spel verder te laten bij het level dat niet is gehaald. Als je dus af was in level 4, begint het spel daarna opnieuw in level 4.

153. Vind je dit een betere of slechtere keuze dan steeds opnieuw bij level 1 beginnen. Leg uit.
154. Verwacht je dat dit de manier van spelen van een speler zal veranderen?
155. Open *H3O23.js* in jouw editor en pas de code aan, zodanig dat aan deze eis is voldaan.
156. Bij veel spelers zorgt deze aanpassing ervoor dat ze tijdens het spel meer risico nemen.
 - a. Verklaar dit.
 - b. Heb je dit zelf ook zo ervaren?

De programmeur heeft een aantal bewuste keuzes gemaakt bij de ontwikkeling van dit spel. Zo kan de speler b.v. niet links uit het canvas verdwijnen, maar verdwijnen je wel boven uit beeld als je te hoog springt.

157. Pas het spel aan, zodanig dat de speler bovenaan niet meer uit beeld kan verdwijnen.
158. Vergelijk de speelervaring van de spelvarianten waarbij je wel en niet boven uit beeld kunt verdwijnen. Welke variant vind je...
 - a. ... aantrekkelijker?
 - b. ... uitdagender?
 - c. ... eerlijker?

Motiveer steeds je antwoord.

Opdracht 24 Jumper II: game settings en levels

Bij de vorige opdracht is je gevraagd om na te denken over manieren waarop je de moeilijkheidsgraad van *Jumper* zou kunnen vergroten. In deze opdracht bekijken we een aantal opties. Misschien had jij ze ook al bedacht.

159. Open *H3O24.js* in de browser en probeer *Jumper* uit te spelen.

HINT: Lees verder als het niet lukt!

De game developer heeft een inschatting gemaakt van het instapniveau van spelers. Deze keuzes kun je vinden in de game **settings** (instellingen) van het spel.



FIGUUR 3.34

160. Open *H3O24.js* in jouw editor en bekijk de game settings (bovenaan) en de bijbehorende beschrijving. Leg per instelling uit of een verhoging de moeilijkheidsgraad verhoogt of verlaagt.
161. Pas de settings aan zodanig dat het je niet lukt om het spel in één keer uit te spelen.
162. Zijn de instellingen die je nu hebt gekozen ook de beste settings als je dit spel online zou plaatsen voor anderen? Zo niet: wat zou je veranderen? Motiveer je antwoord.
163. Beschrijf op welke manier de moeilijkheidsgraad per level toeneemt.
164. In figuur 3.34 zie je een screenshot van *Jumper*. Het spel geeft feedback met kleuren. Beschrijf de betekenis van de kleuren oranje, groen, rood, grijs en zwart.

Opdracht 25 Jumper III: beloning

Bij de vorige versies van *Jumper* werd de speler beloond door het halen van een level. In deze opdracht bekijken we of we het spel aantrekkelijker kunnen door de speler op andere manieren te belonen.

165. Open *H3O25.js* in de browser en speel deze versie van *Jumper*. Merk op dat de game settings zijn aangepast.
166. Op welke manier speelt **timing** een rol bij dit spel?

Het spel vertelt niet op welke manier er punten gehaald en verloren kunnen worden. Je moet dat als speler zelf ontdekken.



FIGUUR 3.35

167. Beschrijf nauwkeurig hoe de score tot stand komt.
 - a. Wanneer krijg je er punten bij?
 - b. Op welke twee manieren verlies je punten?
168. Beschrijf op welke manier de manier waarop jij speelt wordt beïnvloed door de wijze waarop punten gewonnen en verloren kunnen worden. Is jouw **strategie** nu veranderd?

Je wordt als speler beloond als je het spel uitspeelt, maar behalve een score krijg je ook een ranking. Afhankelijk van het niveau waarop je speelt ben je *beginner*, *gevorderd* of *ninja*.

169. Hoe voelde het om *beginner* genoemd te worden?
170. Veranderde je gevoel op het moment dat je *gevorderd* werd?
171. Kreeg je de neiging om het spel opnieuw te proberen op het moment dat je zag dat iemand anders de status van *ninja* kreeg? Of zie je om je heen dat er sprake is van competitie?



Opdracht 26 Memory

172. Open *H3O26.js* in de browser en speel *Memory*. De spelregels ken je vast al.

Het spel geeft op dit moment weinig **feedback**. Ook is er geen scoresysteem: het maakt niet hoeveel pogingen je doet om de puzzel op te lossen.

173. Bedenk minimaal twee manieren om de feedback van dit spel te verbeteren.
174. Bedenk een manier waarop de score zou kunnen afhangen van de vaardigheid van de speler.
175. Open *H3O26.js* in jouw editor en bouw de door jou aangehaalde punten in het spel in.

Opdracht 27 Zure Regen: uitdaging zonder levels

Tot nu toe hebben we naar spelletjes gekeken waarbij je een level kon halen. Het spel werd steeds onderbroken op het moment dat dit was gelukt. Er zijn ook spellen waar je zonder onderbreking doorspeelt. Soms loopt hierbij het nummer van het level nog steeds op, maar regelmatig zijn levels niet zichtbaar en loopt alleen de moeilijkheidsgraad op, zoals in het spel *Zure Regen* (figuur 3.36).

176. Open *H3O27.js* in de browser en speel *Zure Regen*.

In figuur 3.31 zie je een diagram met uitdaging versus vaardigheid.

177. Waar zat jij in dit diagram op het moment dat...

- a. ... je voor het eerst *Zure Regen* speelde?
- b. ... je het drie keer had geprobeerd?
- c. ... je voor het eerst een niveau boven de 200 haalde?

178. Op welke manier is gezorgd dat de moeilijkheidsgraad toeneemt?

179. Wat vind je van de mate van uitdaging...

- a. ... aan het begin van het spel?
- b. ... tijdens het spel: neemt de moeilijkheidsgraad in de juiste mate toe?

Geef een toelichting bij je antwoorden.

180. Kwam je in een *flow*? Waarom wel of niet, denk je?

Zure Regen is een type spel dat je nooit kunt winnen: je gaat altijd af.

181. Geef je mening over dergelijke spellen.

182. Noem twee andere spellen waarbij je uiteindelijk altijd *af* bent.

183. Hoe kun je de spelduur verlengen?

- a. Hoe krijg je er levens bij?
- b. En hoe verlies je levens?

 Het spel geeft ondermeer **feedback** door het aantal levens en je niveau te vermelden. Op dit moment neemt je niveau ook toe als je druppels mist (omdat het gelijk is aan het aantal gemaakte druppels).

184. Open *H3O27.js* in jouw editor en bestudeer de code.

185. Geef de klasse **Speler** een nieuw attribuut niveau.

186. Zorg dat in het attribuut niveau wordt opgeslagen hoeveel druppels de speler opgevangen heeft.

Natuurlijk zorg je ervoor dat dit tijdens het spel wordt weergegeven.

Afhankelijk van je prestatie word je als Roodkapje, padvinder of ninja gekwalificeerd.

187. Speel het spel met de zojuist gedane aanpassingen. Vanaf welke waarde van niveau moet een speler wat jou betreft het niveau padvinder krijgen? En ninja?

188. Zorg dat het eindscherm de waarde van niveau toont, inclusief bijbehorende afbeelding op basis van jouw antwoord op de vorig vraag.

Opdracht 28 Bluebird II

In deze opdracht kijken we naar een verbeterde versie van *Bluebird* (figuur 3.37; zie opdracht 18).

188. Open *H3O28.js* in de browser en speel *Bluebird*.

189. Open *H3O28.js* in jouw editor en bekijk de game **settings**. Overleg met een klasgenoot over de beste instellingen.

Dit spel heeft nog geen vermelding van levels of score.



FIGUUR 3.36



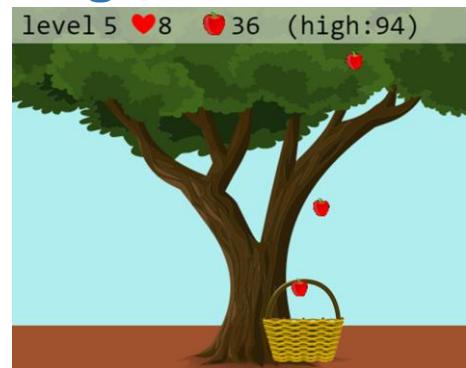
FIGUUR 3.37

190. Gebruik de methode **tekenScorebord** in de klasse **Speler** om je eigen scorebord te maken op basis van een zelfbedacht scoresysteem.

Opdracht 29 Appelvanger: levens en highscore

Bij het spel *Galgje* (zie de opdrachten in § 3.2) is het tekenen van de galg opgedeeld in een aantal stappen. Dit aantal geeft aan hoeveel fouten je nog mag maken, zonder af te zijn. Vaak wordt dit aantal fouten in spellen aangegeven met het aantal **levens** dat je nog hebt.

Er zijn spellen waar het aantal levens niet alleen afneemt als je een fout maakt, maar ook toeneemt als je iets goeds doet: een **beloning!** Het spel *Appelvanger* is zo'n spel.



FIGUUR 3.38

191. Maak drietalen en ga met elkaar naar één computer. Open *H3O29.js* in de browser en speel één voor één twee maal achter elkaar het spel *Appelvanger*.

De twee mensen die niet spelen, observeren de speler.

Noteer (zonder iets hardop te zeggen over wat je ziet):

- Welk gedrag vertoont de speler? Hoe reageert hij op het spel? Wat doet en zegt hij?
- Hoe verandert het gedrag van de speler als het spel vordert? Wat is het verband met de speltoestand van dat moment?
- Hoe reageert de speler als hij bijna af is?
- En als hij helemaal af is? Reageert hij bijvoorbeeld op het vermelde eindniveau?
- Hoe reageert iemand die zelf niet aan het spelen is op het spel van de ander?

192. Bespreek met jouw drietal jullie waarnemingen. Kunnen jullie ze verklaren? Zijn er verschillen tussen de groepsleden?

193. Bespreek met elkaar elementen van een goede **strategie** voor dit spel.

194. Verandert de beste strategie als het spel vordert? Leg uit.

195. Welke spelelementen zorgen ervoor dat je meer stress gaat ervaren als het spel vordert?

196. Welke invloed heeft het vermelden van een **highscore** op de spelbeleving?

197. Heeft het vermelden van het level volgens jou invloed op de spelbeleving? Leg uit.

We zoomen nu in op de ontwikkeling van de moeilijkheidsgraad van dit spel. We gebruiken weer het algemene diagram in figuur 3.31.

189. Waar zat *jij* in dit diagram op het moment dat...

- d. ... je voor het eerst *Appelvanger* speelde?
- e. ... je het drie keer had geprobeerd?
- f. ... je voor het eerst padvinder-niveau haalde?

190. Welke rol speelt willekeur of kans (*random*) in dit spel?

191. Op welke manier is gezorgd dat de moeilijkheidsgraad tijdens het spel toeneemt?

192. Wat vind je van de mate van uitdaging...

- a. ... aan het begin van het spel?
- b. ... tijdens het spel: neemt de moeilijkheidsgraad in de juiste mate toe?

Geef een toelichting bij je antwoorden.

193. Kwam je in een *flow*? Waarom wel of niet, denk je?

194. De spelregels van het spel staan nergens vermeld. Hoe zou je de spelregels formuleren, zodanig dat deze voor elke buitenstaander zowel duidelijk als volledig zijn?

195. Bedenk met elkaar welke aanpassingen je zou willen doen, om het verloop van het spel aantrekkelijker te maken.

Om te zorgen dat zoveel mogelijk spelers in een *flow* komen tijdens het spel, probeert men de **settings** van de game zo goed mogelijk in te stellen.

196. Open *H3O29.js* in jouw editor en bekijk de game **settings** (bovenaan). Zitten er instellingen bij die je bij de vorige vraag had benoemd?

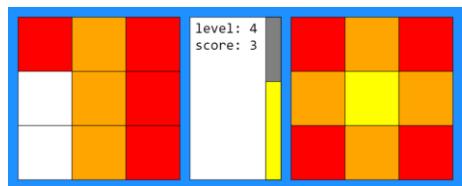
197. Beschrijf per instelling van de game settings of een verhoging leidt tot een vereenvoudiging van het spel, of niet. (Probeer het desnoods uit).

198. Pas de settings aan zodanig dat jullie denken dat een gemiddelde speler die het spel nog nooit gespeeld heeft pas na minimaal vijf pogingen ninja-niveau haalt.

Opdracht 30 Puzzels: Tegelzetter

Eerder in deze module heb je misschien al kennismaking gedaan met spellen als *Galgje*, *Codekraker*, *Nim* of *Memory*. Dit zijn spellen die meer draaien om strategie dan om behendigheid. De spellen bevatten een puzzlelement.

In deze opdracht kijken we naar de **puzzel Tegelzetter**. Bij dit spel moet je tegelpatronen maken. Zie figuur 3.39.



FIGUUR 3.39

199. Open *H3O30.js* in de browser en speel *Tegelzetter*.

Veel puzzelspellen voldoen volgens de literatuur aan een aantal vaste eisen, zoals:

- i. Het doel van de puzzel is duidelijk
- ii. De puzzel voelt als een haalbare uitdaging
- iii. Het is eenvoudig om een eerste stap richting de oplossing te maken
- iv. De voortgang van de speler is zichtbaar
- v. De moeilijkheidsgraad van de aangeboden puzzels neemt geleidelijk toe
- vi. Voor elke moeilijkheidsgraad zijn er meerdere puzzels

200. Voldoet *Tegelzetter* aan alle hierboven gestelde eisen? Geef een korte onderbouwing van je mening per geformuleerde eis.

201. Voor elk level (of: elke moeilijkheidsgraad) biedt dit spel meerdere mogelijke puzzels (met uitzondering van level 1). Geef twee redenen om per level meerdere opties te geven.

In de praktijk houdt een spelontwikkelaar voor zowel de moeilijkheidsgraad van de eerste puzzel als de ontwikkeling in moeilijkheidsgraad rekening met de **doelgroep**. Zo ken je vast wel spellen die speciaal gemaakt zijn voor kinderen onder de tien jaar.

202. Noem minimaal twee andere spelelementen waar een ontwikkelaar rekening zal houden met de doelgroep van het spel.

De puzzelopdrachten voor dit spel staan in een apart bestand *H3O30_puzzels.js*.

203. Open *H3O30_puzzels.js* in jouw editor. Hoe is de (enige) opdracht van level 1 geformuleerd?

204. Pas de puzzel van level 1 aan naar een andere eenvoudige puzzel.

LET OP: gebruik geen getallen groter dan 5!

Ook voor puzzelspellen geldt dat de makers graag spelers in een *flow* (figuur 3.31) houden.

205. Hoe beoordeel jij het beginniveau van dit spel? En de ontwikkeling van niveau?

206. In *H3O30_puzzels.js* staan ook de opdrachten van de volgende levels. Voeg aan elk level een puzzel toe met een vergelijkbaar niveau als de andere puzzels van dat level.

LET OP: denk goed na, want het is ook mogelijk om een puzzel te maken die onoplosbaar is!

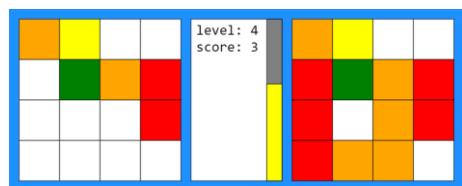
207. Ook dit spel heeft game settings. Open *H3O30.js* in jouw editor en bekijk de game **settings** (bovenaan). Stel de maximale speeltijd per level in op 25 seconden.

208. Het spel bevat ook een setting voor 4x4 puzzels. Speel het spel in deze modus.



Het kan best lastig zijn om puzzels te bedenken en te controleren of ze net onmogelijk zijn om op te lossen. Daarom heeft de ontwikkelaar een *ontwerpmodus* ingebouwd, waarmee je nieuwe mogelijkheden kunt verkennen.

Op dit moment bevat *H3O30_puzzels.js* voor elk 4x4-level twee puzzels, maar die zijn nog wel identiek.



FIGUUR 3.40

209. Stel de *ontwerpmodus* in en gebruik deze om nieuwe puzzels te bedenken. Heb je een leuke puzzel gevonden? Pas dan de levels in *H3O30_puzzels.js* aan (scroll naar onderen voor de 4x4 puzzels.)

210. Test de nieuwe levels door ze een klasgenoot te laten spelen. vindt hij/zij dat je de opbouw in moeilijkheidsgraad van de puzzels goed hebt gekozen? (Zijn alle puzzels oplosbaar?!)

3.6 Tijd

Een kenmerkend verschil tussen spellen en bijvoorbeeld een animatie of boek is de **interactiviteit**: spelers ervaren de vrijheid om zelf het spelverloop te bepalen. Die vrijheid betekent dat spelontwikkelaars geen **directe controle** hebben over de acties van de speler. De vrijheid blijft beperkt door **indirecte controle**, op basis van de **spelregels** en slim spelontwerp. Spelers zijn te beïnvloeden via onder andere het ontwerp van het speelveld, geformuleerde doelen, geluid en opgelegde beperkingen.

Als je *Tegelzetter* (zie vorige opgave) hebt gespeeld, heb je kennismakend met **tijd** als factor in een spel. De tijd was een **beperking** voor de speler. Tijd kan zowel een doel als een beperking zijn. In *voorbeeld 27* (zie figuur 3.41) is tijd een doel: als je de cirkel gedurende een bepaalde tijd weet te volgen, heb je het level (doel) behaald. Bij *Tegelzetter* (opdracht 30) is tijd een beperking: als je de puzzel niet op tijd oplost, ben je af.

Bij deze twee spellen is de tijd zichtbaar aanwezig, maar dat is niet altijd zo. Bij spellen zoals *Appelvanger* (opdracht 29) of *Jumper III* (opdracht 25) krijgt de speler een beperkte tijd om iets te doen. Bij *Appelvanger* vallen de appels steeds sneller, waardoor de **reactietijd** wordt beperkt. Bij *Jumper* worden de platforms (vanaf level 4) steeds kleiner, waardoor de **beslistijd** afneemt (zie figuur 3.42). Bovendien bewegen de platforms ten opzichte van elkaar, zodat het kiezen van het juiste moment een factor wordt. Dit vraagt om een bepaald soort behendigheid: **timing**.

Tijd is vaak gekoppeld aan snelheid. Bij *Bluebird II* (opdracht 28) komen de pilaren steeds sneller op je af. Dat vraagt van de speler naast timing ook een bepaalde **handelingssnelheid**. Omdat er een reeks van pilaren tegelijkertijd zichtbaar is, kan de speler bovendien **anticiperen**: hij kan zich voorbereiden op de komst van (een combinatie van) pilaren. Tijd wordt vaak gebruikt als middel om de moeilijkheidsgraad te laten oplopen als je verder in een spel komt.

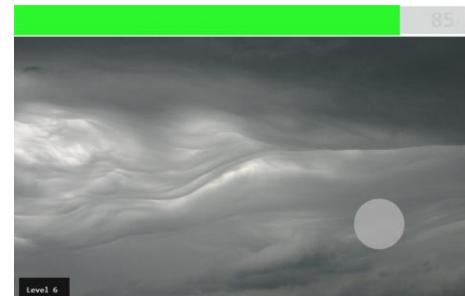
Qua *user experience* zorgt tijd voor een gevoel van stress of haast en op het moment dat een doel gehaald is voor een gevoel van opluchting (beloning). Als tijd als beperking wordt ervaren, kan dit er voor zorgen dat spelers minder goed gaan nadenken en gehaste beslissingen nemen onder tijdsdruk. Aan de andere kant kan het ook zorgen voor (of vragen om) een verhoogde concentratie of **focus**.

Voor het inzetten van tijd hebben we in deze module al gebruik gemaakt van *frameCount* en *frameRate*. Javascript kent meerdere tijdfuncties, die we hier niet uitgebreid zullen bespreken, omdat het hier vooral gaat om de ervaring van tijd door de speler. *Voorbeeld 28* toont wel een aantal voorbeelden van enkele van deze functies en het gebruik van de klasse *Timer* die we voor het eerst in opdracht 30 hebben ingezet.

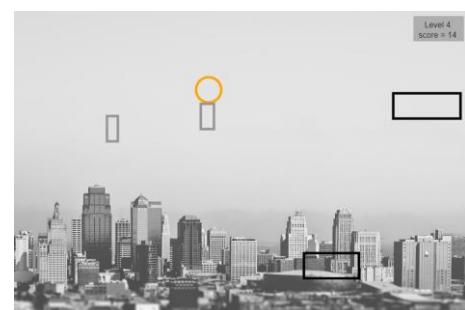
Opdracht 31 FollowMe

Bij het spel *FollowMe* moet je met je muis een object onafgebroken zien te volgen.

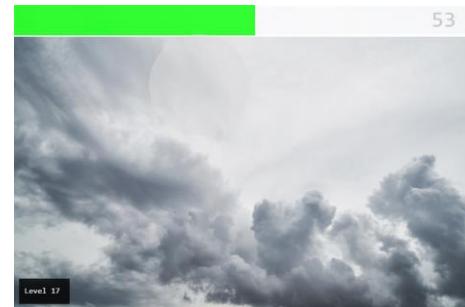
211. Open *H3O31.js* in de browser en speel *FollowMe*. Dit is een variant op *voorbeeld 27*.
212. In de theorie hierboven van paragraaf 3.6 staat een aantal **vetgedrukte** termen. Geef van elk van deze termen aan of ze een rol spelen in dit spel en zo ja welke rol.
213. Open *H3O31.js* in jouw editor en bekijk de game **settings** (bovenaan). Pas de instellingen aan, zodanig dat je vanaf ongeveer level 5 moeite krijgt om het level te halen.
214. Bij het spel ben je nu nooit af. Bedenk spelregels om het spel uit te breiden.



FIGUUR 3.41



FIGUUR 3.42

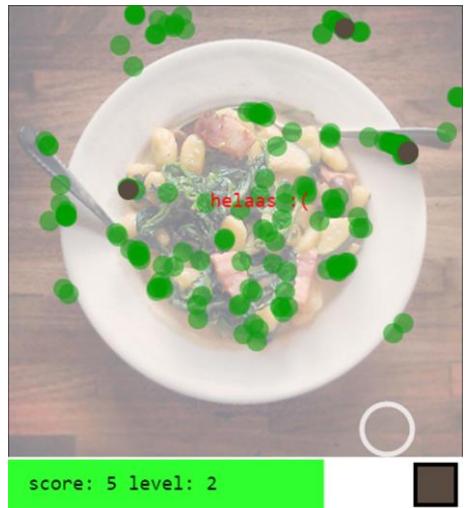


FIGUUR 3.43

Opdracht 32 RGBlik

Heb jij een scherpe blik voor het vinden van kleuren in een afbeelding? Bewijs het met *RGBlik*. Rechts onder (zie figuur 3.44) zie je de kleur die gevonden moet worden. Omdat het vinden van de precieze kleur erg lastig is, is er een mate van tolerantie: je mag er een beetje naast zitten. Het is jouw taak om met de witte cirkel naar de juiste plek in de afbeelding te gaan. In figuur 3.44 is dit de speler niet gelukt. Je ziet drie uitvergrote pixels: die waren precies goed. De groene bollen tonen pixels die binnen de tolerantie vallen.

215. Open *H3O32.js* in de browser en speel *RGBlik*.
216. Beschrijf hoe tijd in dit spel een beperking is.
217. Leg uit dat tijd in dit spel ook een doel is.
218. Open *H3O32.js* in jouw editor en bekijk de game **settings** (bovenaan). Beschrijf van elke afzonderlijke setting of een hogere waarde zorgt voor een grotere of kleinere uitdaging.
219. Het spel bevat de methode **nieuwLevel**. Beschrijf manieren om de moeilijkheidsgraad een volgend level te verhogen door gebruik te maken van variabelen in de gamesettings.
(Je hoeft het niet te programmeren.)



FIGUUR 3.44

Opdracht 33 Zoekspelletje II

Zoekspelletje II is een vervolg op *opdracht 21* van hoofdstuk 2 en een variant op de vorige opdracht.

220. Open *H3O33.js* in de browser en speel *Zoekspelletje II*.

De gamesettings zijn op dit moment zodanig dat de beslistijd telkens afneemt als je succes hebt.

221. Hoe heb je dat tijdens het spelen ervaren? Hoe veranderde jouw focus? Ontwikkelde zich bij jou een gevoel van stress?
222. Onder *opdracht 30* staan een aantal eisen voor puzzelspellen beschreven. Aan welke van deze eisen voldoet dit spel?

Dit spel kent andere mogelijkheden om de moeilijkheidsgraad in stappen te verhogen.

223. Open *H3O33.js* in jouw editor en bekijk de game **settings** (bovenaan). Stel *alfa* in op **0.75** en *delta alfa* op **0.1**. Bekijk en beschrijf het resultaat.
224. Stel *delta diameter* in op **5**. Bekijk en beschrijf het resultaat.
225. Verander de game settings zodanig dat jij vindt dat het spel de juiste flow heeft.

Opdracht 34 Typerend

226. Open *H3O34.js* in de browser en speel *Typerend* (figuur 3.45).
227. Beschrijf wat de spelregels van het spel zijn en hoe de score wordt bepaald. (Ontdek dit door het spel te spelen.)
228. In de theorie van deze paragraaf (§ 3.6) staat een aantal **vetgedrukte** termen. Geef van elk van deze termen aan of ze een rol spelen in dit spel en zo ja welke rol.
229. Open *H3O34.js* in jouw editor en bekijk de game **settings** (bovenaan). Zorg dat de beschikbare tijd voor elk level hetzelfde blijft, maar dat het aantal letters dat je minimaal goed moet intypen elk level met 3 wordt verhoogd. Speel het spel.
230. In figuur 3.45 zie je dat een speler op dit moment drie in te typen letters kan zien. Wat is volgens jou het ideale aantal? Is 1 letter bijvoorbeeld beter dan 5 letters? Gebruik de settings om dit aantal in te stellen en onderzoek of jouw aanname klopt.
231. Optimaliseer de game settings voor jouw leeftijsgenoten.



FIGUUR 3.45

3.7 Maak je eigen game

We hadden je graag nog meer verteld over objecten, programmeertechnieken, grafische trucs, andere soorten games en spelwerelden en trucs om de speelervaring verder te beïnvloeden, zoals *power ups*, *lootboxes* of het *unlocken* van nieuwe levels. Misschien ken je ze van spellen die je zelf hebt gespeeld. We doen het niet, want stiekem hopen we dat je handen inmiddels jeuken om je eigen game te maken. Dat begint meestal niet met programmeren, maar met zaken als:

- een origineel plan voor een spel
- nadelen over *user experience* en *flow*
- een complete set met spelregels
- een globale opzet van het spel met bijvoorbeeld een flowchart
- het identificeren van objecten in het spel
- een plan voor de vormgeving (een schets op papier werkt vaak het best!)

Ook met een goede voorbereiding zal je er achter komen dat het maken van een spel bijna nooit in één keer helemaal goed gaat, al was het maar omdat je tijdens de ontwikkeling nieuwe, nog betere ideeën krijgt. Heb je een eerste versie? Laat hem dan testen door je klasgenoten. Omdat zij het spel niet door en door kennen zoals jij, is het goed mogelijk dat ze spelersgedrag zullen vertonen waar je nog geen rekening mee had gehouden.

Om je op weg te helpen hebben we in de werkruimte al een map *mijnGame* gemaakt waarin je jouw spel kunt gaan maken. We hebben er een voorbeeld neergezet van een *fullscreen* spel dat je kunt bereiken door te klikken op . Dit voorbeeld bevat nog een aantal trucs voor mensen die meer uitdaging zoeken:

- spelen in de browser op volledig scherm
(ook als tussentijds de grootte van het scherm wordt aangepast)
- gebruik van andere **bibliotheken** (*libraries*) voor vormen en geluid; zie voor meer informatie:
<https://p5js.org/libraries>
<https://p5js.org/reference/#/libraries/p5.sound>
- Het gebruik van een apart bestand met gegevens (data) in JSON-formaat (*JavaScript Object Notation*). Zie voor meer informatie:
https://www.w3schools.com/js/js_json_intro.asp

Tot slot nog een lijstje met wat bronnen. Om problemen te voorkomen: let goed op of je een bepaalde bron mag gebruiken of niet. Op de meeste sites staat beschreven onder welke voorwaarden je iets mag gebruiken.

- afbeeldingen / sprites (veel rechtenvrij):
<https://pixabay.com>
<https://www.needpix.com>
<https://www.pxfuel.com>
<https://opengameart.org>
<http://www.publicdomainfiles.com>
- geluidsbronnen (veel rechtenvrij):
<http://www.soundinggames.com>
<https://www.bensound.com>
<https://freesound.org>
<https://bigsoundbank.com>

Natuurlijk zijn er vele andere bronnen op het internet. Nog veel leuker is het om je eigen sprites en sounds te maken! Wat je ook kiest: veel plezier met het maken en spelen van je eigen spel.

H4 GAMIFICATION

4.1 Inleiding

Je kunt hoofdstuk 1 en 2 van deze module doorwerken door alle opdrachten te maken, maar je kunt ook bewijzen dat je over voldoende *skills* beschikt, door bij elke paragraaf één opdracht te maken. Als je de opdracht correct hebt uitgevoerd, heb je een level gehaald en daarmee bewezen dat je de bijbehorende stof begrepen hebt.

De *gamification*-opdrachten vind je onder de knop **GA**. De aanpak is omgekeerd aan die van de andere opdrachten. De uitwerkingen zijn hier niet beschikbaar, want *jij* maakt de uitwerkingen. Als je de opdracht selecteert, zie je wat je eindresultaat moet zijn. In het bijbehorende uitwerkingen-bestand is steeds al een beginnetje voor je gemaakt. Om een gamification-level te halen moet *jij* de opdracht zelf afmaken.

4.2 Levels

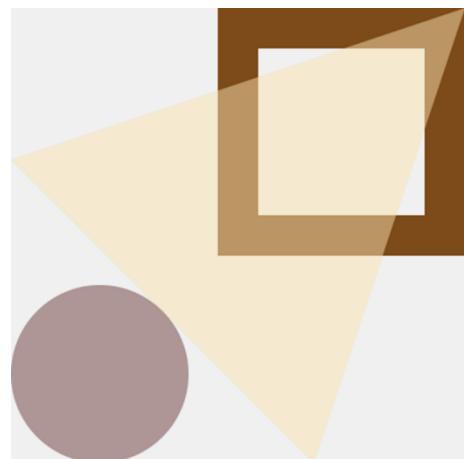
Level 1 bij §1.1 vormen, kleuren & positie

Open *HGAO1.js* in de browser. Je ziet dan het eindresultaat van uit figuur 4.1. Jij moet deze figuur namaken.

Open *HGAO1U.js* in jouw *editor*. Hierin is al een beginnetje gemaakt.

Voer de volgende taken uit:

- Voeg linksonder een cirkel in met een diameter van 176 pixels en RGB-kleurcode 175,150,150.
- Voeg rechtsboven een vierkant toe met zijdes van 225 pixels en een rand van 40 pixels dik met RGB-kleurcode 125,75,25. Zorg dat het vierkant geen vulkleur heeft en achter de driehoek komt te staan.
- Zorg dat de driehoek voor 50% doorzichtig is, zodat je het kader van het vierkant weer volledig kunt zien.



FIGUUR 4.1

Level 2 bij §1.2 verplaatsen & draaien

Open *HGAO2.js* in de browser. Je ziet het eindresultaat van figuur 4.2. Je moet deze figuur namaken.

Open *HGAO2U.js* in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- `rectMode(CENTER)` en `angleMode(DEGREES)` worden in de `setup` aangeroepen.
- `translate(30 + 40, 100);` zorgt er daarom voor dat het eerste vierkant (met zijdes van 80 pixels) 30 pixels van de linkerrand van het canvas wordt getekend.
- Om de volgende drie vierkanten te tekenen, wordt telkens dezelfde combinatie van twee regels (met `rect` en `translate`) gebruikt.



FIGUUR 4.2

Programmeer de volgende stappen:

- Geef de vier getoonde vierkanten een rand van 5 pixels dik van de kleur *lemonchiffon*.
- Zorg dat elk vierkant ten opzichte van zijn voorganger (vanaf links gezien) 15° met de klok mee gedraaid is.
- Gebruik `push` en `pop` om te zorgen dat `translate(80 + 30, 0);` steeds alleen voor een horizontale verplaatsing zorgt.
- Heb je een vast patroon gevonden dat zich herhaalt? Breid dan het aantal vierkanten uit naar 8 zodat figuur 4.2 ontstaat.

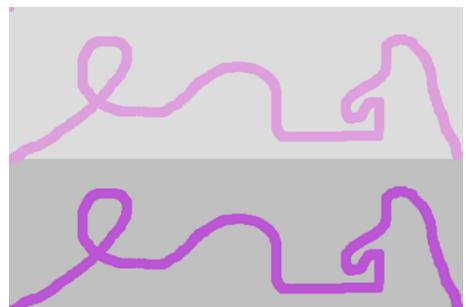
Level 3 bij §1.3 variabelen gebruiken

Open HGAO3.js in de browser. Beweeg je muis, om tot een eindresultaat te komen, vergelijkbaar met figuur 4.3. Je moet dit tekenspel namaken.

Open HGAO3U.js in jouw editor. Regel 9, 14 & 16 zijn leeg.

Voer de volgende taken uit:

- Voeg in regel 9 code toe die gebruik maakt van de variabele hoogte (zie regel 1) om een *silver* rechthoek te tekenen over de onderste helft van het canvas (zie figuur 4.3).
- Voeg in regel 14 code toe die op de plaats waar de muis zich bevindt een cirkel met diameter 10 tekent.
- Voeg in regel 16 een coderegel toe om het programma af te maken volgens het voorbeeld.



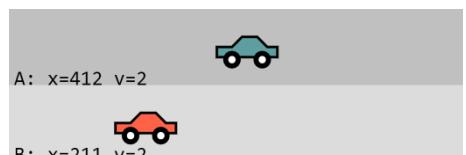
FIGUUR 4.3

Level 4 bij §1.4 loopfunctie inzetten

Open HGAO4.js in de browser. Je ziet een race tussen twee auto's. In figuur 4.4 zie je daarvan een momentopname.

Open HGAO4U.js in jouw editor. In de code zijn al een aantal keuzes gemaakt:

- Er zijn variabelen aangemaakt voor de coördinaten en de snelheid van de twee auto's.
- Er is een variabele *finish* gemaakt gedeclareerd, die aangeeft voor welke waarde van *x* de auto de finish heeft bereikt.
- Er zijn tekstregels gemaakt, om de actuele x-positie van beide auto's te kunnen weergeven. Hierbij is voor auto B de functie *round* gebruikt.
- Er is een functie *tekenAuto* geprogrammeerd, waarmee de twee auto's kunnen worden getoond.



FIGUUR 4.4

Voer de volgende taken uit:

- Roep de functie *tekenAuto* twee maal aan (regel 24 en 25 zijn hiervoor leeg gelaten) om auto A (met kleur *cadetblue*) en auto B (met kleur *tomato*) te tekenen.
- Laat de auto's rijden. Gebruik hierbij de gedeclareerde variabelen voor de snelheid van de auto's.
- Gebruik *constrain* om te zorgen dat de auto's niet verder kunnen rijden dan de door *finish* aangegeven x-positie.
- Zorg dat behalve de x-positie ook de snelheid van de auto's wordt getoond (zie figuur 4.4).
- In de eindversie gaat auto B steeds sneller. Declareer bovenaan een variabele *acceleratieB* met de waarde *0.01*.
- Zorg dat de snelheid van B elke loop toeneemt met *acceleratieB*.
- Zorg dat de snelheid van B met één cijfer achter de komma op het scherm wordt getoond.

Level 5 bij §1.5 functies maken

Open HGAO5.js in de browser en maak kennis met Toby (figuur 4.5).

Open HGAO5U.js in jouw editor. Vanaf regel 17 wordt Toby getekend.

Voer de volgende taken uit:

- Maak een functie *tekenToby* om Toby te tekenen. Zorg dat aan de functie een parameter *s* kan worden meegegeven, die aangeeft op welke schaal Toby moet worden getekend.
- Roep *tekenToby* aan. Gebruik hierbij als argument de variabele *schaal* die bovenaan is gedeclareerd.
- Vanaf regel 10 is een if-else-structuur geprogrammeerd. Pas deze aan, zodanig dat de variabele *schaal* met 1% toeneemt als er op de muis wordt geklikt en anders de waarde *1* krijgt.



FIGUUR 4.5

Level 6 bij §1.6 voorwaarden

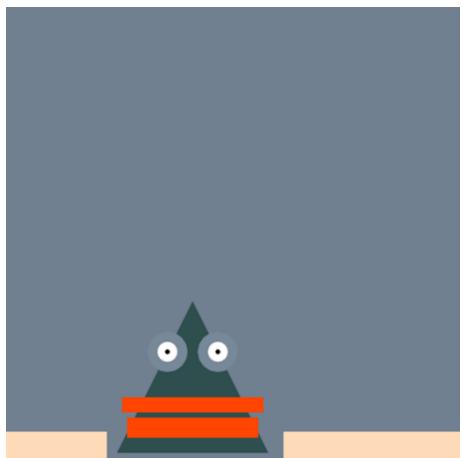
Open *HGAO6.js* in de browser. Gebruik de pijltjestoetsen (links en rechts) om Toby veilig te laten landen tussen de obstakels zoals in figuur 4.6. Je moet dit spelletje maken.

Open *HGAO6U.js* in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er zijn variabelen gedeclareerd voor de positie van Toby.
- Er zijn functies gemaakt om Toby en de obstakels te tekenen.

Voer de volgende taken uit:

- a. Zorg dat Toby valt. Elke loop moet hij één pixel naar beneden vallen.
- b. Zorg dat de achtergrond wit wordt als Toby de onderkant van het canvas raakt. Roep op dat moment `noLoop` aan, zodat Toby stopt met bewegen.
- c. Voeg code toe, zodat Toby tijdens het vallen naar links en rechts kan bewegen. Het drukken op een pijltjestoets moet steeds een stap van 5 pixels verplaatsing opleveren.
- d. Beperk Toby's beweging: zorg dat Toby links en rechts altijd volledig binnen het canvas blijft.
- e. Zorg dat de achtergrond zwart wordt als Toby een obstakel raakt. Roep op dat moment `noLoop` aan, zodat Toby stopt met bewegen. Zie figuur 4.7.



FIGUUR 4.6



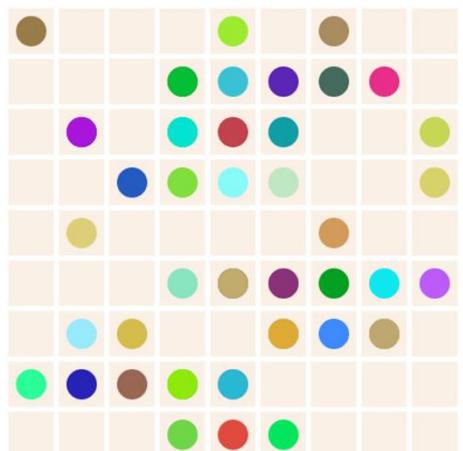
FIGUUR 4.7

Level 7 bij §1.7 vaste herhalingen

Open *HGAO7.js* in de browser. Je ziet een raster met een willekeurig stippenpatroon, zoals in figuur 4.8. Laad de pagina enkele keren opnieuw, zodat je ziet dat zowel de positie als de kleur van de stippen willekeurig is.

Open *HGAO7U.js* in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er is een variabele `aantal` gedeclareerd die aangeeft hoeveel stippen er moeten worden getekend.
- De functie `tekenRaster` tekent op dit moment één cel van het raster in kolom 4 van rij 1.
- De functie `tekenStip` tekent op dit moment één stip met kleur `darkgoldenrod` in kolom 4 van rij 1 met behulp van de parameters `x` en `y`.



FIGUUR 4.8

Voer de volgende taken uit:

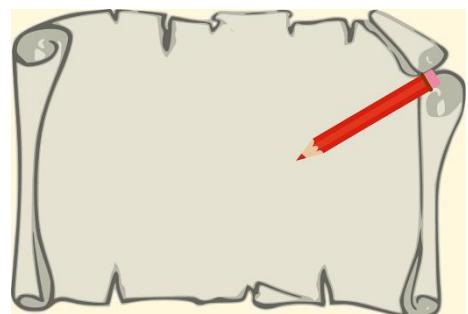
- a. Pas de functie `tekenRaster` aan, zodat een raster van 9×9 cellen wordt getoond. Gebruik hiervoor een vaste herhaling.
- b. Pas regel 11 aan, zodat de stip in het midden van een willekeurige cel in het raster wordt geplaatst.
- c. Gebruik een vaste herhaling om meerdere stippen in het raster te plaatsen. Gebruik hierbij de variabele `aantal`.
- d. Pas de functie `tekenStip` aan, zodat de stippen een willekeurige kleur krijgen.
- e. Verklaar dat je bijna nooit werkelijk 50 stippen te zien krijgt.
- f. EXTRA: Wil je een scherm met dansende stippen? Haal dan de regel met `noLoop` weg. Voeg eventueel `frameRate(10);` toe aan de `setup`, om te voorkomen dat je hoofdpijn krijgt...

Level 8 bij §2.2 sprites

Open *HGAO8.js* in de browser. Je ziet een stuk perkament en een roodpotlood, dat meebeekt met de muis. Merk op dat de cursor samenvalt met de punt van het potlood tot een zekere grens: de potloodpunt blijft altijd binnen bepaalde marges op het canvas.

Open *HGAO8U.js* in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er zijn variabelen gedeclareerd (*p* & *perkament*) die bedoeld zijn om een afbeeldingsobject in op te slaan
- Er zijn variabelen gedeclareerd (*pX* & *pY*) die straks de coördinaten bevatten waar het potlood *p* moet worden getoond
- Er zijn variabelen gedeclareerd (*margeHorizontaal* & *margeVerticaal*) die aangeven hoe dicht de potloodpunt bij de randen van het canvas mag komen (in pixels).
- De gebruikte afbeeldingen bevinden zich in de map *images* in de sublocatie:
backgrounds/perkament.svg
sprites/potlood_400.png
- Er is een coderegel gemaakt, om de afbeelding van het perkament te laden



FIGUUR 4.9

Voer de volgende taken uit:

- a. Voeg code toe om de perkament-afbeelding vooraf te laden in het object *perkament*.
- b. Haal de `//` weg zodat `background(perkament);` wordt uitgevoerd.
(Laat de andere regel met `background` gewoon staan!)
- c. Voeg code toe om de afbeelding van het potlood vooraf te laden in het object *p*.
- d. Maak gebruik van de variabelen *pX* en *pY* om het potlood op in het canvas te tonen.
- e. Voeg coderegels toe, zodanig dat de waardes *pX* en *pY* afhangen van de positie van de muis.
- f. Gebruik de eigenschap *hoogte* van het potlood-object om te zorgen dat de cursor samenvalt met de potloodpunt.
- g. Gebruik *margeHorizontaal* en *margeVerticaal* en de functie `constrain` om *pX* & *pY* in te perken, zodanig dat de potloodpunt links en rechts 125 pixels en boven en onder 50 pixels van de rand van het canvas blijft.

Level 9 bij §2.3 arrays

Open *HGAO9.js* in de browser. Als je de muis beweegt zie je een reeks met stippen die het pad van de muis volgt zoals in figuur 4.10. Om precies te zijn wordt er bij de cursor steeds een nieuwe stip gemaakt. Telkens als dat gebeurt, verdwijnt er aan het eind van de reeks een stip, zodat het totaal aantal stippen gelijk blijft.



FIGUUR 4.10

Open *HGAO9U.js* in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er zijn twee arrays *stipX* & *stipY* gedeclareerd, waarin de x- en y-coördinaten van de stippen worden opgeslagen
- Er wordt nu één stip getekend op de coördinaten [60,60] en een RGB-kleur waarvan de hoeveelheid rood eveneens 60 is

Voer de volgende taken uit:

- a. Gebruik een herhaling en de arrays *stipX* en *stipY* om de tien stippen op het scherm te tonen. Gebruik voor de hoeveelheid rood van de RGB-vulkeur de x-coördinaat van de stip.
- b. Gebruik `push` om tijdens elke loop van de *draw* een waarde aan de arrays *stipX* en *stipY* toe te voegen. De nieuwe waardes moeten overeenkomen met actuele x- en y-coördinaat van de muis.
- c. Met `shift` verwijder je het eerste element uit een array. Gebruik deze functie om te zorgen dat elke loop van de *draw* de *oudste* stip van de lijst (aan het eind van de keten) verdwijnt.

Level 10 bij §2.4 arrays van plaatjes

Open HGAO10.js in de browser. Je ziet *Flatboy* springen door een groen graslandschap (figuur 4.11). Het beeldmateriaal is rechtenvrij gedownload via www.gameart2d.com (voor *Flatboy*) en www.publicdomainfiles.com (voor het graslandschap). Kijk eens rond op deze websites. Wie weet doe je inspiratie op voor het gebruik in een eigen spel!

Open HGAO10U.js in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er zijn variabelen gemaakt voor de x-positie (*flatboyX*) en de breedte en hoogte waarin de sprite moet worden getoond (*flatboyBreedte* & *flatboyHoogte*)
- Het boven genoemde landschap is geladen met **preload** en wordt door **background** gebruikt.



FIGUUR 4.11

Voer de volgende taken uit:

- a. Declareer animatie als lege array en gebruik **preload** om hierin alle beeldjes van een springende *Flatboy* te laden. De gebruikte beeldjes bevinden zich in de map *images* in de sublocatie: [sprites/flatboy/Jump\(1\).png](#) (1 t/m 15)
- b. Zorg dat (alleen) het eerste beeldje van de array animatie getoond wordt op [200,100]. Gebruik *flatboyBreedte* & *flatboyHoogte* voor het juiste formaat van de afbeelding.
- c. Zorg dat *Flatboy* gaat springen met behulp van alle beeldjes in de array animatie. Gebruik de modulus-functie om te zorgen dat de animatie zich steeds herhaalt.
- d. Zorg dat *Flatboy* gaat bewegen vanaf de positie *flatboyX* met een snelheid van 5 pixels per frame.
- e. Zorg dat *Flatboy* weer op x-positie **-140** wordt geplaatst als deze rechts 'uit beeld' verdwenen is.

Level 11 bij §2.6 objecten maken

Open HGAO11.js in de browser. Je ziet een trein die door een sneeuwlandschap rijdt. Met de pijltjes-toetsen links en rechts van je toetsenbord kun je de snelheid van de trein veranderen.



FIGUUR 4.12

Open HGAO11U.js in jouw *editor*. In de code zijn al een aantal keuzes gemaakt:

- Er is een object spoor gemaakt. Hierover moet de trein gaan rijden.
- In de **preload** zijn afbeeldingen geladen voor zowel de achtergrond als de trein.

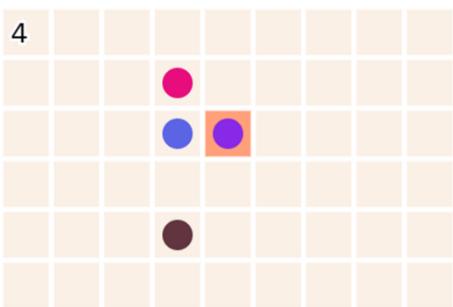
Voer de volgende taken uit:

- a. Maak een object *trein* met de attributen *x* (waarde: **100**), *y* (**100**), *snelheid* (**-10**), *schaal* (**5**) en *sprite* (**null**).
- b. Voeg een methode **toon** toe die er bij aanroepen voor zorgt dat de afbeelding van de trein wordt getoond op het coördinaat [x,y] van het object *trein*. Gebruik hiervoor het attribuut *sprite*.
LET OP: *sprite* heeft nu nog de 'waarde' **null**. Hoe los je dat op?
- c. Verwijder de **//** voor de regel *trein.toon();* in de *draw* zodat de trein zichtbaar wordt.
- d. De trein is op dit moment te groot. Gebruik het attribuut *schaal* van *trein* om de trein **5** maal kleiner te tonen (dan het oorspronkelijke afbeeldingsbestand).
- e. Voeg een methode **beweeg** toe die er voor zorgt dat de x-positie van *trein* veranderd, door er de waarde van het attribuut *snelheid* bij op te tellen.
- f. Breid **beweeg** uit, zodat de snelheid van de trein met 1 afneemt als op de linker pijltoets wordt gedrukt en met 1 toeneemt als op de rechter pijltoets wordt gedrukt.
- g. De trein mag niet te hard rijden: zorg dat de snelheid tussen de -25 en 25 blijft.
- h. Als de trein het canvas aan een kant verlaat, moet hij weer aan de andere kant verschijnen.
Breid **beweeg** uit, zodat ook aan deze eis wordt voldaan.
- i. Pas de beginpositie van *trein* aan, zodat hij in eerste instantie vanaf rechts het canvas binnenrijdt.

Level 12 bij §2.7 object dat antwoordt

Open *HGAO12.js* in de browser. Je ziet het raster van een geluksspelletje. Als je met je muis klikt, verschijnt een stip. Klik je opnieuw, dan verschijnt er een extra stip.

Het aantal stippen staat linksboven. Dat aantal is ook jouw score, want de bedoeling is om te blijven klikken, zonder dat er een stip op de door jou geselecteerde cel komt en dat wordt steeds moeilijker als er meer stippen worden geplaatst. Je kunt er voor kiezen om je muis ingedrukt te houden, of om steeds een nieuwe positie te zoeken. Als je af bent (zoals in figuur 4.13) dan stopt het spel. Herlaad de pagina voor een nieuw spel (b.v. met F5).



FIGUUR 4.13

Open *HGAO12U.js* in jouw editor. Het grootste gedeelte van het spel hebben we voor je geprogrammeerd. Het is de bedoeling dat je jezelf 'in de code inleeft' door deze goed te bestuderen. We noemen alvast:

De methode `plaatsStip` wordt aangeroepen als er met de muis wordt geklikt en plaatst dan willekeurig stippen binnen het raster (die worden getoond met `tekenStip`). Na het tekenen van een stip wordt gecontroleerd of de stip zich op dezelfde plek bevindt als de zojuist geplaatste stip. Hiervoor is de methode `controleerRaak`. Deze methode moet `true` antwoorden als de stip zich op de plaats van de met de muis geselecteerde cel bevindt en anders `false`. De methode `controleerRaak` is nog niet af.

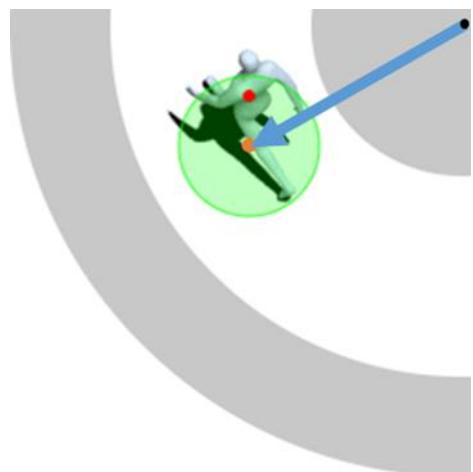
Voer de volgende taken uit:

- Breed de methode `controleerRaak` uit, zodat aan bovenstaande eis wordt voldaan.
- Had je al ontdekt dat je vals kan spelen? Je kan namelijk ook op de muis klikken, als die zich niet in het canvas bevindt. Pas het spel aan, zodat er alleen een nieuwe stip geplaatst wordt, als de muis zich in het speelveld (dus: op het canvas) bevindt.

Level 13 bij §2.8 klasse van objecten

Open *HGAO13.js* in de browser. Je ziet het bovenaanzicht van een man in een baan. Met de linker- en rechterpijl-toets kun je hem laten wandelen. Als de man op een grijs deel terecht komt, kleuren de randen van de baan donkergrijs. Dit gebeurt slechts bij benadering:

De sprite heeft een lastige vorm. Daarom wordt met behulp van de groene cirkel (zie figuur 4.14) bepaald of de man één van de randen raakt. De blauwe pijl is de onderlinge afstand tussen het middelpunt van de baan en het middelpunt van de cirkel. We zeggen dat de man 'goed loopt' zolang deze afstand groter is dan de straal van de kleine cirkel en kleiner dan de straal van de grote cirkel.



FIGUUR 4.14

Open *HGAO13U.js* in jouw editor. Een groot deel staat al klaar:

- Er is een object `speler` met attributen, sprites en methodes.
- Ten behoeve van het ontwikkelen is code toegevoegd, zodat je de man met een muisklik op zijn plaats kan laten staan.

Voer de volgende taken uit:

- Schrijf het object `speler` om naar een klasse `Man`. Zorg er hierbij voor dat je, als je de `//` voor regel 98 weghaalt, een instantie van de klasse `Man` met de juiste waarden van de attributen maakt.
- Maak een klasse `Circuit` met in ieder geval het attribuut `kleur` en de methode `teken`. Als de `//` voor regel 97 wordt weggehaald, moet een instantie van `Circuit` worden gemaakt met als straal van de binneste cirkel `75` en een breedte (van de witte baan) van `100`.
- Verwijder alle `//` en `/* */` uit de code. Dit zorgt er onder andere voor dat de methode `raakt` van het object `speler` wordt aangeroepen. Deze methode geef op dit moment altijd `false` als antwoord. Breid de methode uit, zodat het eindresultaat van *HGAO13* ontstaat.

Level 14 bij §2.9 array van objecten

In dit laatste level verkennen we het basisprincipe van een *shooter*.

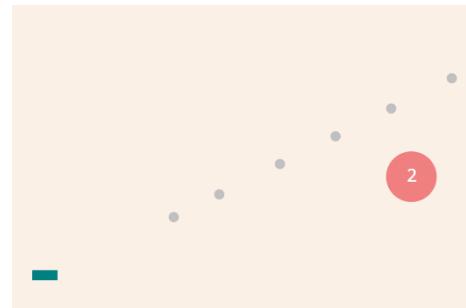
Open *HGAO14.js* in de browser. Jij bestuurt een kanon (met de pijltjestoetsen) en moet hiermee de vijand (figuur 4.15) raken. Elke keer dat je op de spatiebalk drukt, vuur je één kogel op de vijand af. Die is niet meteen ‘dood’: pas als je hem vijf keer raakt is het gelukt. Hoeveel kogels heb jij daar voor nodig? Hoe minder hoe beter! Lukt het jou om slechts vijf kogels te gebruiken?

Open *HGAO14U.js* in jouw *editor*. Een groot deel van de code staat al klaar:

- Er is een klasse *Vijand* met onder meer een methode *wordtGeraakt* waaraan een object *k* (een kogel) wordt meegegeven. Als de kogel de vijand raakt, antwoord de methode met *true*.
- Er is een klasse *Kogel* waarmee nieuwe kogels kunnen worden gemaakt.
- Er is een klasse *Kanon* met onder andere het attribuut *kogels* en de methode *schiet*. De array *kogels* is nu nog leeg. Elke keer als op de spatiebalk wordt gedrukt, moet een nieuwe instantie van de klasse *Kogel* worden gemaakt die wordt toegevoegd aan de array *kogels*.

Voer de volgende taken uit:

- a. Breid de methode *schiet* van de klasse *Kanon* uit, zodat er een instantie van de klasse *Kogel* aan de array *kogels* wordt toegevoegd als op de spatiebalk wordt gedrukt. LET OP: de kogels moeten wel de juiste x- en y-coördinaten meekrijgen!
- b. De klasse *Kogel* heeft een methode *teken* om een kogel in het canvas te tonen. Breid de *draw* met een herhaling, zodanig dat alle kogels in de array *kogels* te zien zijn.
- c. Breid de herhaling uit, zodat de kogels ook gaan bewegen.
- d. De methode *wordtGeraakt* van de klasse *Vijand* is al klaar voor gebruik, maar wordt nog niet aangeroeopen. Zet deze methode in binnen de *draw* en zorg er hierbij voor dat er een eindscherf (vergelijkbaar met *HGAO14*; zie figuur 4.16) verschijnt, als de vijand geen levens meer heeft.



FIGUUR 4.15



FIGUUR 4.16

INDEX

- aanroepen, 14
actie, 59
afbeeldingen, 27
animatie, 32
anticiperen, 73
array, 30, 48
asymmetrische spellen, 66
attribuut, 37
background, 4, 27
behendigheid, 59, 73
beloning, 67, 71
beperking, 73
beslistijd, 73, 74
bibliotheeken, 75
boolean, 42, 55
camelCase, 27
Cantorverzameling, 25
canvas, 4
createCanvas, 4
declareren, 9
design pattern, 64
directe controle, 73
dist, 17
doel, 67
doelgroep, 72
draaien, 9
draw, 4, 10
Droste-effect, 24
eigenschappen, 27
ellipse, 4
else, 16
en, 16
event handler, 59
event listener, 59
events, 59
extends, 51
false, 42
feedback, 67, 69, 70
fill, 4
flow, 67, 74
flowchart, 53, 64
focus, 73, 74
for-loop, 19
fractal, 25
frameCount, 32, 33
frameRate, 11, 32
frames, 32
fullscreen, 75
functiedefinitie, 14
functies, 14
game-loop, 64, 67
gamification, 76
gelijkwaardige spelers, 66
geluk, 58
get, 31
handelingssnelheid, 73
height, 9, 27
hexadecimale kleurcodes, 6
highscore, 71
hitbox, 63
hoofdprogramma, 54
if, 16, 19
image, 27, 35
indirecte controle, 73
initialisatie, 9
instantie, 45, 48
interactief, 59
interactiviteit, 59, 73
invoer, 59
key, 59
keyCode, 59
keyTyped, 60
klasse, 45, 48
kleur, 6
leereffect, 68
levels, 67
levens, 71
libraries, 75
lijst, 30
line, 15
loadImage, 27
loadPixels, 31
logische operatoren, 16
loopfunctie, 11
methode, 37
modulus, 33
moeilijkheidsgraad, 59, 67
mouselsPressed, 49, 61
mouseX, 9, 61
mouseY, 9, 61
multi-player, 65
Nim, 53
noLoop, 10, 11
notatie, 27
null, 37
object, 27, 30, 37
object-georiënteerd, 37
of, 16
ongelijkwaardige spelers, 66
opdracht, 67
optimale strategie, 58
overerving, 51
paradigma, 37
parameter, 7, 14, 42
parameters, 14
pop, 8
preload, 27, 28
push, 8
puzzel, 72
random, 21
reactietijd, 73
real-time, 66
rect, 4
recursie, 23
RGB, 6
rol, 66
rotate, 9, 20
score, 67
sequentieel, 66
settings, 69, 70, 71, 72, 73, 74
setup, 4, 10
shape, 5
Sierpinski-driehoek, 26
simultaan, 65, 66
single-player, 65
speelervaring, 58, 64
spelregels, 59, 67, 73
speltoestand, 64
sprite, 32
sprites, 27
spritesheet, 35
stapgrootte, 19
stopconditie, 24
strategie, 58, 69, 71
stress, 74
stroke, 4
subklasse, 51
superklasse, 51
symmetrische spellen, 66
syntax, 7
this, 37
tijd, 73
timing, 69, 73
toewijzing, 9
touches, 62
touchscreen, 62, 63
translate, 7
triangle, 5
true, 42
turn-based, 66
uitdaging, 67
user experience, 58, 59, 64, 67, 73
variabele, 9
volledig scherm, 75
voorwaarde, 16
width, 9, 27