Prisma simulatie

Table of contents

Shape.js

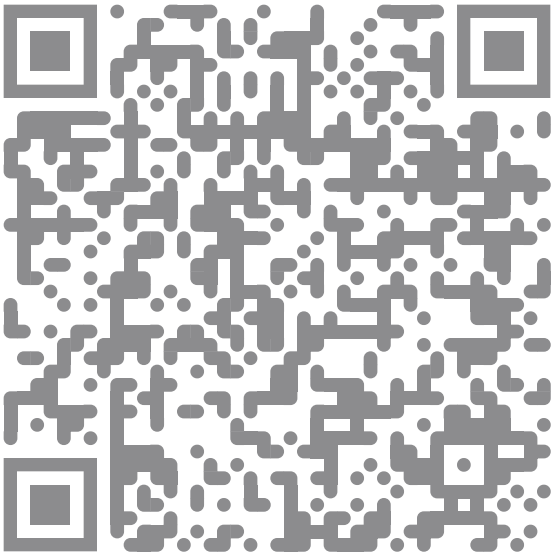
Dit stukje code definieert een JavaScript object genaamd Shape, dat verschillende basis berekeningen bevat. Laten we eens kijken naar elk van de methoden:

De functie `intersectRay()` bepaalt het snijpunt tussen een straal en een vorm. Eerst berekent het de hoek van de straal en past het vervolgens iets aan om precisieproblemen te vermijden. Vervolgens doorloopt het elke rand van de vorm en controleert of de straal ermee snijdt. Als er een snijpunt wordt gevonden, slaat het het snijpunt en de normalen (loodrechte richtingen) op dat punt op. Na het vinden van alle snijpunten, identificeert het het dichtstbijzijnde snijpunt ten opzichte van het startpunt van de straal en stuurt informatie hierover, inclusief het snijpunt, de vorm en de normaal op dat punt.

De functie `contains()` controleert of een gegeven punt `(mx, my)` zich binnen een veelhoek bevindt die wordt gedefinieerd door een array (een lijst als het ware) van punten. Het maakt gebruik van het ray casting-algoritme, waarbij een straal wordt uitgezonden vanaf het gegeven punt en wordt geteld hoe vaak deze de randen van de veelhoek kruist. Als het aantal kruisingen oneven is, ligt het punt binnen de veelhoek; anders ligt het buiten.

Deze code wordt gebruikt als basis voor alle andere vormen, dus de `Circle.js`, `Line.js`, `Rectangle.js` & `Triangle.js` zijn allemaal opgebouwd met deze code als basis en krijgen de vorm door een aantal punt te definiëren, dit scheelt weer code en zorgt voor een soepeler gebruik tussen meerdere vormen

Deze code is te vinden op Shape.js (https://github.com/MattterSteege/Prism-Simulation/blob/master/Shape.js)



ShapeJsQR.svg

Line.js

Dit stukje code definieert een JavaScript klasse genaamd `Line`, die lijnen representeert en tekent op een canvas. Laten we het in meer detail bekijken:

De `Line` klasse wordt gedefinieerd met een constructor die parameters zoals beginpunt (`x1`, `y1`), lengte, breedte, hoek (in graden) en vulling (optioneel) accepteert. De constructor initialiseert de eigenschappen van de lijn, zoals positie, lengte, breedte, hoek en vulkleur. De methode `updatePoints` wordt ook aangeroepen om de punten van de lijn te bijwerken op basis van de opgegeven eigenschappen.

De `draw` methode wordt gebruikt om de lijn te tekenen op een gegeven tekencontext (`ctx`). Het stelt eerst de vulkleur en lijnkleur in op basis van de opgegeven vulling. Vervolgens begint het een nieuw pad, verplaatst naar het eerste punt van de lijn, en tekent vervolgens lijnen naar elk punt van de lijn. Ten slotte sluit het het pad door terug te keren naar het eerste punt en tekent het de lijn op het canvas met behulp van de `stroke` methode van de context.

De `Line` klasse is afgeleid van een basis klasse genaamd `Shape`, wat suggereert dat het waarschijnlijk andere eigenschappen en methoden erft die relevant zijn voor geometrische vormen.

Over het algemeen biedt dit stukje code functionaliteit voor het maken en tekenen van lijnen op een canvas, en het maakt gebruik van een rotatiefunctie (`rotatePoints`) om de punten van de lijn bij te werken op basis van de opgegeven hoek.

Triangle.js

Dit stukje code definieert een `Triangle`-klasse die een driehoek vertegenwoordigt en ervan erft van de `Shape`-klasse. Laten we eens kijken naar wat dit codefragment doet:

De `Triangle`-functie is de constructor van de `Triangle`-klasse. Deze functie initialiseert een nieuwe driehoek met de opgegeven x- en y-coördinaten, breedte, hoek (in graden) en vulkleur. Als er geen waarden worden opgegeven, worden standaardwaarden gebruikt. De hoek wordt genormaliseerd om ervoor te zorgen dat deze binnen het bereik van 0 tot 360 graden blijft.

De `draw`-methode tekent de driehoek op een gegeven context (`ctx`). Het stelt eerst de vulkleur en lijnkleur van de context in. Vervolgens begint het een nieuw pad en verplaatst het naar het eerste punt van de driehoek. Daarna doorloopt het alle punten van de driehoek en tekent het lijnen tussen deze punten. Na het sluiten van het pad wordt de omtrek van de driehoek getekend door de `stroke`-methode van de context op te roepen.

De `updatePoints`-methode berekent de hoekpunten van de driehoek op basis van de huidige positie, breedte en hoek van de driehoek. Het berekent de hoogte van de driehoek op basis van de breedte (aangenomen wordt dat het een gelijkzijdige driehoek is) en gebruikt vervolgens trigonometrie om de coördinaten van de drie hoekpunten te berekenen. Daarna worden de punten geroteerd op basis van de opgegeven hoek, en worden ze opgeslagen in het `points`-attribuut van de driehoek.

Kortom, dit stukje code implementeert een `Triangle`-klasse die een driehoek representeert, deze kan tekenen op een canvas en de hoekpunten kan updaten op basis van de positie, breedte en hoek van de driehoek.

Circle.js

Dit stukje code implementeert een JavaScript klasse genaamd `Circle`, die cirkels representeert en tekent op een canvas. Laten we het in meer detail bekijken:

De `Circle` klasse wordt gedefinieerd met een constructor die parameters zoals positie (`x`, `y`), straal (`w`) en vulkleur (`fill`) accepteert. De constructor initialiseert de eigenschappen van de cirkel, zoals positie, straal en vulkleur. Indien deze waarden niet zijn opgegeven, worden standaardwaarden gebruikt.

De `draw` methode wordt gebruikt om de cirkel te tekenen op een gegeven tekencontext (`ctx`). Het stelt eerst de vulkleur en lijnkleur in op basis van de opgegeven vulling. Vervolgens begint het een nieuw pad, tekent lijnen tussen de opeenvolgende punten op de omtrek van de cirkel en sluit het pad. Het vullen van de cirkel is uitgeschakeld in deze implementatie, maar het tekenen van de omtrek gebeurt met de opgegeven vulkleur.

De `contains` methode controleert of een gegeven punt (`mx`, `my`) binnen de grenzen van de cirkel valt. Dit wordt gedaan door te berekenen of de afstand tussen het punt en het middelpunt van de cirkel kleiner is dan de straal van de cirkel.

De `updatePoints` methode berekent de punten op de omtrek van de cirkel met behulp van de opgegeven resolutie. Het berekent de x- en y-coördinaten van elk punt op de omtrek door de straal te vermenigvuldigen met de cosinus en sinus van de hoek rond de cirkel.

Over het algemeen biedt dit stukje code functionaliteit voor het creëren en tekenen van cirkels op een canvas, evenals het controleren of een punt binnen de cirkel valt. Het maakt gebruik van een objectgeoriënteerde benadering om cirkelvormige vormen te manipuleren en ermee te interageren binnen een JavaScript-toepassing.

Rectangle.js

Dit stukje code implementeert een JavaScript klasse genaamd `Rectangle`, die rechthoeken representeert en tekent op een canvas. Laten we het in meer detail bekijken:

De `Rectangle` klasse wordt gedefinieerd met een constructor die parameters zoals de positie van de linkerbovenhoek (`x`, `y`), de breedte (`w`), de hoogte (`h`), de rotatiehoek (`angle`) en de vulkleur (`fill`) accepteert. Als een parameter niet wordt opgegeven, worden standaardwaarden gebruikt. Vervolgens wordt de methode `updatePoints` aangeroepen om de hoekpunten van de rechthoek te berekenen en bij te werken.

De `draw` methode wordt gebruikt om de rechthoek te tekenen op een gegeven tekencontext (`ctx`). Het stelt eerst de vulkleur en lijnkleur in op basis van de opgegeven vulling. Vervolgens begint het een nieuw pad en worden de hoekpunten van de rechthoek doorlopen om een gesloten vorm te creëren. Ten slotte wordt het pad gesloten en de omtrek van de rechthoek getekend.

De `contains` methode controleert of een gegeven punt (`mx`, `my`) binnen de grenzen van de rechthoek valt. Dit wordt gedaan door te controleren of de x- en y-coördinaten van het punt binnen de x- en y-bereiken van de rechthoek liggen.

De `updatePoints` methode berekent en werkt de hoekpunten van de rechthoek bij op basis van de huidige positie, breedte, hoogte en rotatiehoek van de rechthoek. Het gebruikt de huidige eigenschappen van de rechthoek om de hoekpunten te genereren en roteert ze vervolgens indien nodig.

Over het algemeen biedt dit stukje code functionaliteit voor het creëren en tekenen van rechthoeken op een canvas, evenals het controleren of een punt binnen de rechthoek valt en het bijwerken van de hoekpunten op basis van de huidige eigenschappen van de rechthoek.

LightDispersion.js

De brekingsindex van een materiaal is een fundamentele eigenschap die beschrijft hoe licht zich door dat medium voortplant. Het is de verhouding van de lichtsnelheid in een vacuüm tot de lichtsnelheid in het materiaal. Hoe hoger de brekingsindex, hoe langzamer het licht door het medium reist en hoe meer het buigt of breekt wanneer het van het ene naar het andere medium gaat.

Nu is de brekingsindex van een materiaal geen constante waarde; het hangt af van de golflengte van het licht. Dit fenomeen staat bekend als dispersie, en het verklaart waarom een prisma wit licht kan opsplitsen in zijn samenstellende kleuren (een regenboog). Verschillende golflengten van licht hebben verschillende brekingsindices in hetzelfde materiaal, waardoor ze onder verschillende hoeken buigen.

Om het gedrag van licht in verschillende materialen nauwkeurig te modelleren, heb je een manier nodig om de brekingsindex voor elke gegeven golflengte te berekenen. Hier komt de Sellmeier-vergelijking om de hoek kijken.

De Sellmeier-vergelijking berekend de brekingsindex van een transparant materiaal (zoals glas) relateert aan de golflengte van licht. Het werd ontwikkeld door experimentele gegevens aan een wiskundig model aan te passen. De functie `getSellmeierValue(wavelength)`. in de gegeven code implementeert deze vergelijking, met behulp van vooraf bepaalde constanten (B1, B2, B3, C1, C2 en C3) die specifiek zijn voor het gemodelleerde materiaal.

De vergelijking gaat als volgt:

$$n^2\left(\lambda\right) = 1 + \frac{B_1\lambda^2}{\lambda^2 - C_1} + \frac{B_2\lambda^2}{\lambda^2 - C_2} + \frac{B_3\lambda^2}{\lambda^2 - C_3}$$

daarbij is:

- n de brekingsindex
- λ de golflengte van het licht in vacuüm in meters
- B1, B2, B3, C1, C2 en C3 de experimenteel bepaalde Sellmeier-coëfficiënten

Maar de Sellmeier-vergelijking alleen is niet genoeg om het gedrag van licht in verschillende media volledig te beschrijven. Je moet ook rekening houden met de

brekingsindex van lucht, die licht varieert met de golflengte. De functie `getAirIndex(Wavelength)`. berekent de brekingsindex van lucht met behulp van een empirische formule die de golflengte als input neemt.

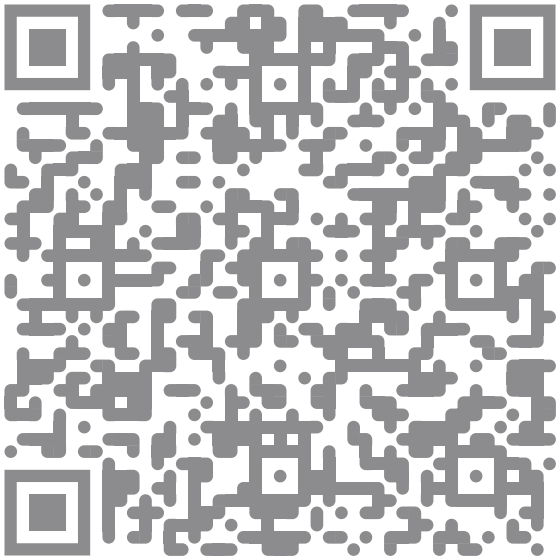
```
//LightDispersion.js
function getSellmeierValue(wavelength) {
    wavelength = wavelength * 1E-9; // Convert nm to m
    const L2 = wavelength * wavelength
    const B1 = 1.03961212;
    const B2 = 0.231792344;
    const B3 = 1.01046945;

    // Constants for converting to metric
    const C1 = 6.00069867E-3 * 1E-12; //
    const C2 = 2.00179144E-2 * 1E-12;
    const C3 = 1.03560653E2 * 1E-12;

    // Calculate the Sellmeier value using Sellmeier equation
    return Math.sqrt( 1 + B1 * L2 / ( L2 - C1 ) + B2 * L2 / ( L2 - C2 )
+ B3 * L2 / ( L2 - C3 ) );
}

function getAirIndex(wavelength) {
    return 1 +
        5792105E-8 / ( 238.0185 - Math.pow( wavelength * 1E6, -2 ) ) +
        167917E-8 / ( 57.362 - Math.pow( wavelength * 1E6, -2 ) );
}
```

Deze code is te vinden op LightDispersion.js (https://github.com/MattterSteege/Prism-Simulation/blob/master/LightDispersion.js)



LDJsQR.svg

Ray.js

Dit stukje code implementeert een JavaScript klasse genaamd `Ray`, die een straal representeert en tekent op een canvas.

De `Ray` klasse wordt gedefinieerd met een constructor (regels 5-16) die parameters zoals positie (`x`, `y`), hoek (`angle`), golflengte (`waveLength`), vulling (`fill`), en interne lichtkleur (`lightColorInternal`) accepteert. De constructor initialiseert de eigenschappen van de straal zoals positie, hoek, golflengte, vulling en interne lichtkleur. Indien deze waarden niet zijn opgegeven, worden standaardwaarden gebruikt. De hoek wordt genormaliseerd en de punten van de straal worden bijgewerkt.

De functie `draw(ctx)` (regels 22-79) tekent de straal op een gegeven tekencontext (`ctx`). Het berekent de straal delen (`rayParts`) door middel van de `calculateRay` methode en tekent zowel de lamp als de straaldelen. Indien de gebruiker aangeeft dat normaalvectoren moeten worden weergegeven (`user.showNormals`), worden deze ook getekend.

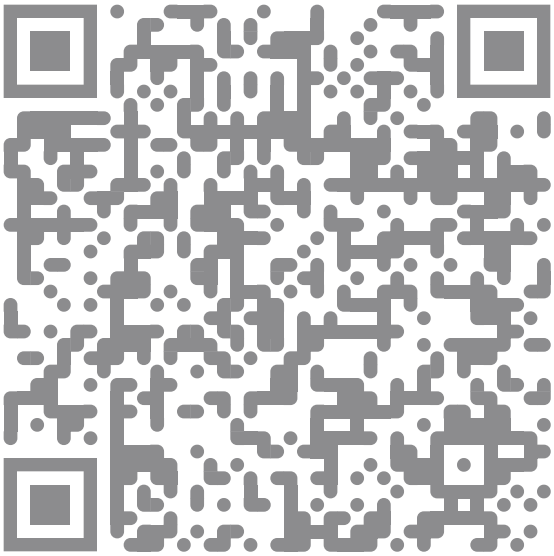
De functie `calculateRay(shapes)` (regels 81-179) berekent de delen van de straal (`rayParts`) door de hoek en de breking van de straal te bepalen bij interacties met objecten (`shapes`). Het houdt rekening met reflectie en breking aan de hand van Sellmeier's wet en controleert of de straal binnen een object is.

De functie `updatePoints()` (regels 181-185) berekent de punten van de straal op basis van de positie, breedte en hoogte van de straal, en roteert deze punten volgens de hoek. Het bepaalt ook het uitstootpunt van de straal (`emittingPoint`).

De functie `calculateRefractedAngle(n1, n2, angleIncidence)` (regels 187-211) berekent de gebroken hoek op basis van de brekingsindices (`n1` en `n2`) en de invalshoek (`angleIncidence`). Het houdt rekening met totale interne reflectie.

De functie `calculateCriticalAngle(n1, n2)` (regels 213-221) berekent de kritieke hoek voor totale interne reflectie op basis van de brekingsindices (`n1` en `n2`).

Deze code is te vinden op Ray.js (https://github.com/MattterSteege/Prism-Simulation/blob/master/Ray.js)



RayJsQR.svg

Hoe gebruik je de simulatie

Om de simulatie beter te kunnen begrijpen kan je er het best natuurlijk zelf mee gaan spelen, dat doe je zo:

De simulatie downloaden

Je kan de simulatie heel makkelijk gebruiken, je doet dat zo:

Download de simulatie

1. je gaat naar Mijn github pagina voor de Prismasimulatie (https://github.com/MattterSteege/Prism-Simulation/)
2. Klik `Code` en dan op `Download` + `ZIP`

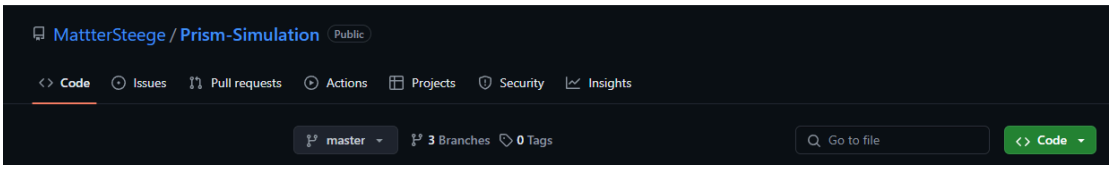


Image 1

3. Pak het ZIP-bestand uit door met de `rechtmuisknop` op het bestand te klikken, dan op `Alles` + `uitpakken...` te klikken en als op `Uitpakken` te klikken

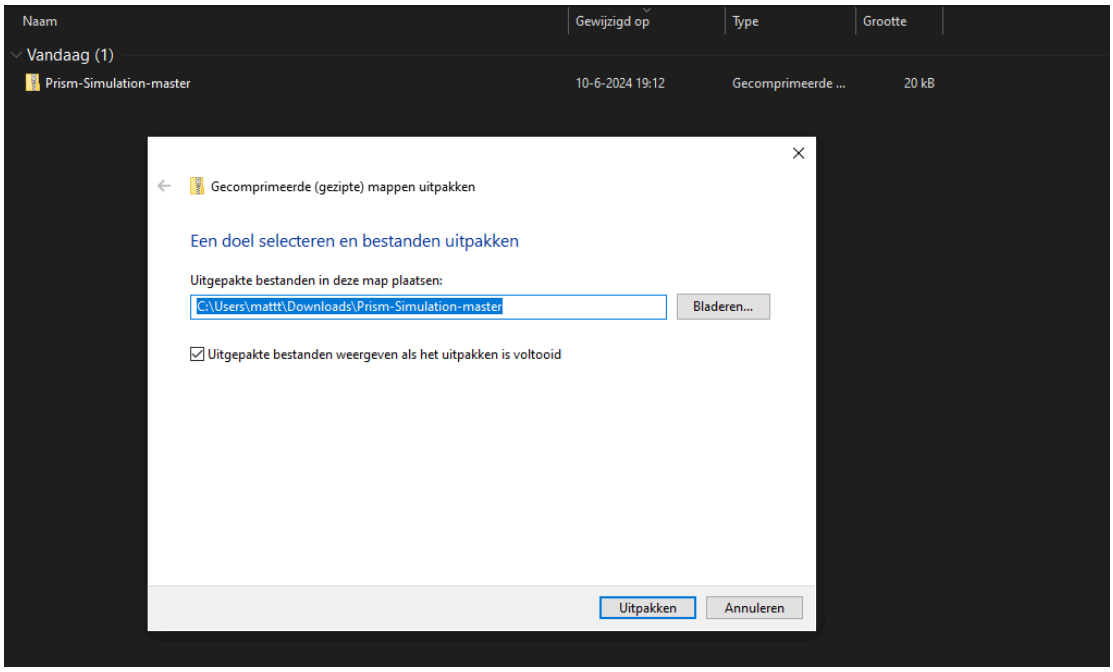


Image 2

4. Dubbelklik op index.html en de simulatie opent automatisch in je geselecteede browser

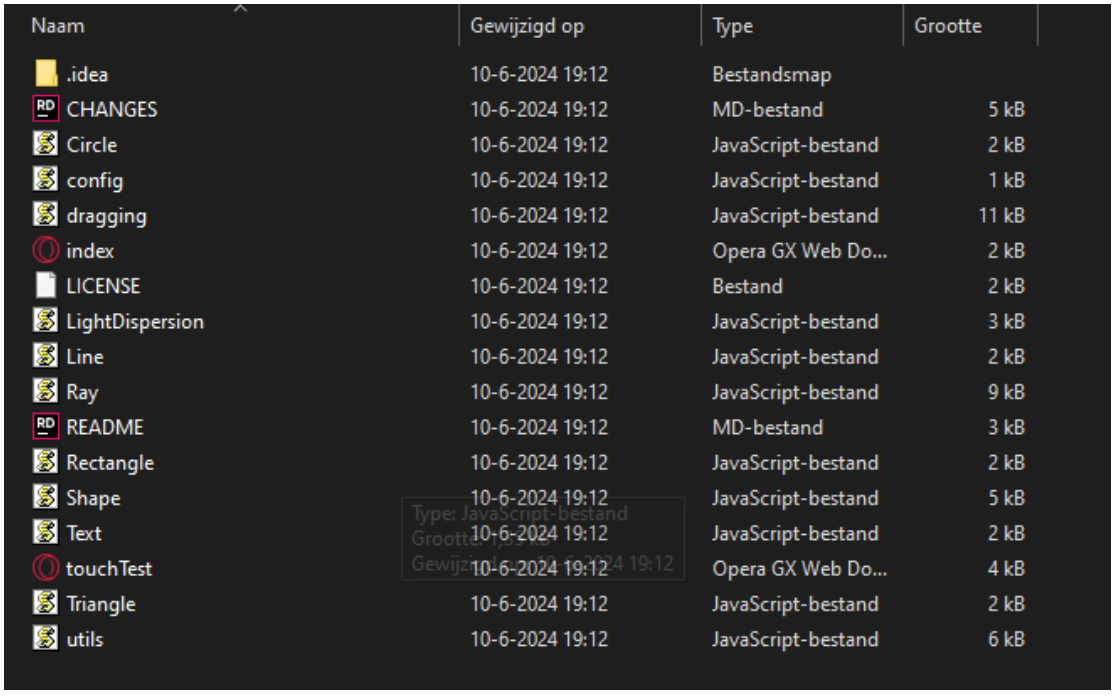


Image 3

De simulatie gebruiken

De simulatie is heel makkelijk te gebruiken en dat gaat zo:

- `linkermuisknop indrukken op een vorm en slepen`: sleept die vorm over het canvas
- `scrollen met je muis op een vorm`: draait de vorm

De simulatie aanpassen (vooraf)

Je kan de simulatie ook aanpassen naar jouw eigen smaak je gaat hiervoor naar `config.js` in de bestanden die je in de vorige stap het gedownload dan kan je vervolgens de volgende dingen aanpassen:

showNormals

true: laat de normaal-lijnen van de simulatie zien. false: laat deze lijnen niet zien

AmountOfRays

Dit is een geheel getal dat aangeeft hoeveel lichtlijnen er getekend moeten worden, hoe meer lijnen hoe dichter bij de werkelijkheid, maar dit kost meer vermogen van je PC/Laptop

maxLightBounces

Dit is een geheel getal dat aangeeft wat de maximale hoeveelheid aan lichtbreking berekeningen er gedaan mogen worden. Dit is een maximum, de simulatie stop waneer de lichtstraal uit beeld gaat sowieso!

*mogelijke andere opties in `config.js` zijn niet gebruikt in de code!

De simulatie aanpassen (real-time)

Je kan de simulatie ook aanpassen tewijl hij bezig is, dit doe je door `user.[showNormals | AmountOfRays | maxLightBounces] = [waarde]` in te typen in de Chrome console. Deze open je door `Ctrl + Shift + I` op je toetsenbord in te toetsen.

Ook kan je nieuwe vormen toevoegen via de Chrome console dat gaat als volgt:

Driehoek

```
s.addShape(new Triangle(x, y, width, color));
```

Hierin is:

- `x` een x-coordinaat
- `y` een y-coordinaat
- `width` een breedte
- `color` een HEX kleur (zoals #ffffff voor wit)

Rechthoek

```
s.addShape(new Rectangle(x, y, width, heigth, color));
```

- `x` een x-coordinaat
- `y` een y-coordinaat
- `width` een breedte
- `height` een hoogte
- `color` een HEX kleur (zoals #ffffff voor wit)

Lijn

```
s.addShape(new Line(x, y, angle, length, width, color));
```

- `x` een x-coordinaat
- `y` een y-coordinaat

- `angle` de hoek in graden
- `length` de lengte
- `width` een breedte
- `color` een HEX kleur (zoals #ffffff voor wit)

Cirkel

```
s.addShape(new Circle(x, y, radius, color));
```

- `x` een x-coordinaat
- `y` een y-coordinaat
- `radius` de radius van de cirkel
- `color` een HEX kleur (zoals #ffffff voor wit)

Licentie

Dit project valt onder de MIT-licentie.

```
MIT License

Copyright (c) 2024 Matt ter Steege

Permission is hereby granted, free of charge, to any person obtaining a
copy
of this software and associated documentation files (the "Software"), to
deal
in the Software without restriction, including without limitation the
rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE
SOFTWARE.
```

De MIT-licentie is een permissieve vrije softwarelicentie, die ontwikkeld is door het Massachusetts Institute of Technology (MIT). Het is een van de meest gebruikte licenties in de open-source gemeenschap vanwege zijn eenvoud en brede toepasbaarheid.

Onder deze licentie krijgt elke gebruiker die een kopie van deze software en bijbehorende documentatiebestanden verkrijgt, de toestemming om zonder beperkingen te handelen met betrekking tot de software. Dit omvat onder andere het recht om de software te gebruiken, te kopiëren, aan te passen, samen te voegen, te publiceren, te distribueren, onder te licentiëren en/of te verkopen.

Er zijn echter enkele voorwaarden verbonden aan deze rechten. De belangrijkste voorwaarde is dat de originele copyrightvermelding en deze toestemming in alle kopieën of substantiële delen van de software opgenomen moeten worden. Dit betekent dat gebruikers, wanneer zij de software of een deel daarvan gebruiken of distribueren, altijd gerefereerd moeten worden aan de originele programmeur, in dit geval mij, Matt ter Steege.

Het is ook belangrijk op te merken dat de software wordt geleverd "zoals deze is", zonder enige vorm van garantie, expliciet of impliciet. Dit betekent dat de auteur niet verantwoordelijk kan worden gehouden voor eventuele problemen of schade die voortvloeit uit het gebruik van de software. Dit omvat, maar is niet beperkt tot, garanties van verkoopbaarheid, geschiktheid voor een bepaald doel en niet-inbreuk op rechten.