Prisma simulatie

Table of contents

Shape.js

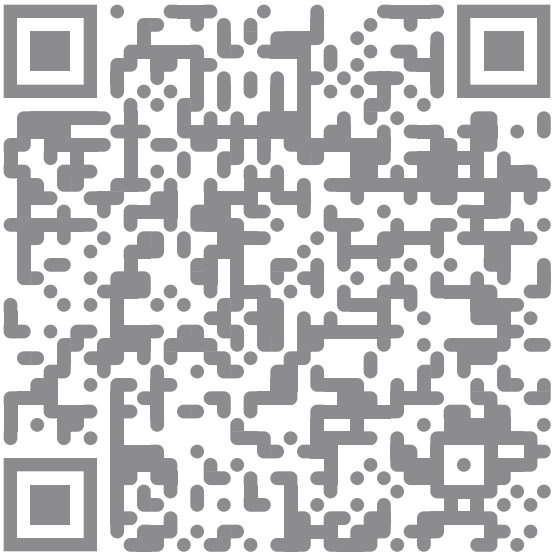
Dit stukje code definieert een JavaScript object genaamd Shape, dat verschillende basis berekeningen bevat. Laten we eens kijken naar elk van de methoden:

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

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.

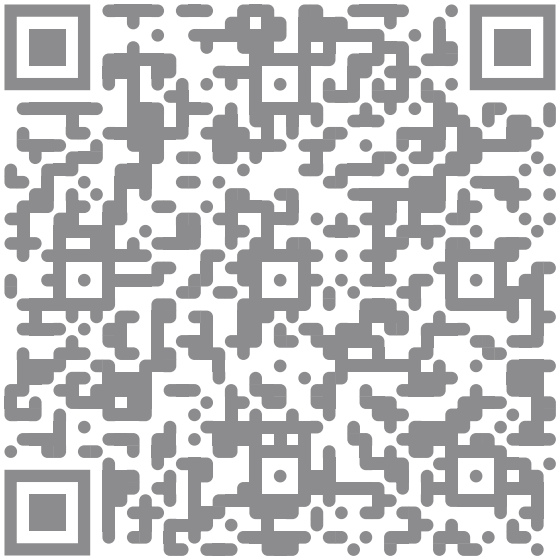
```
//LightDispersion.js
function getSellmeierValue(wavelength) {
    wavelength = wavelength * 1E-9; // Convert nm to m
    const L2 = wavelength * wavelength
    const B1 = 1.03961212;
    const B2 = 0.231792344;
    const B3 = 1.01046945;

    // 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.

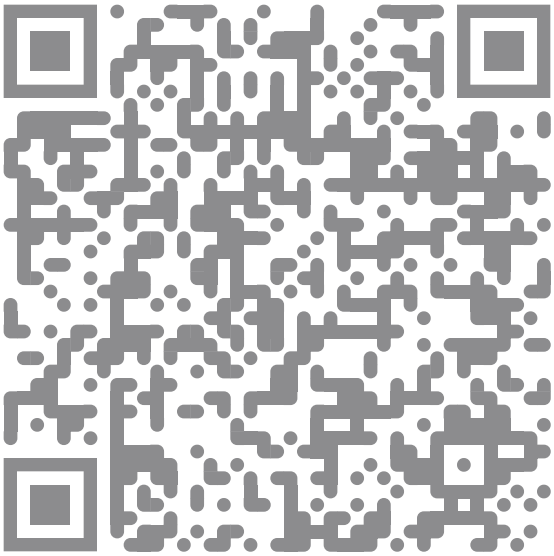
De functie `calculateRay(shapes)` (regels 81-179) berekent de delen van de straal (`rayParts`) door de hoek en de breking van de straal te bepalen bij interacties met objecten (`shapes`). Het houdt rekening met reflectie en breking aan de hand van Sellmeier's wet en controleert of de straal binnen een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

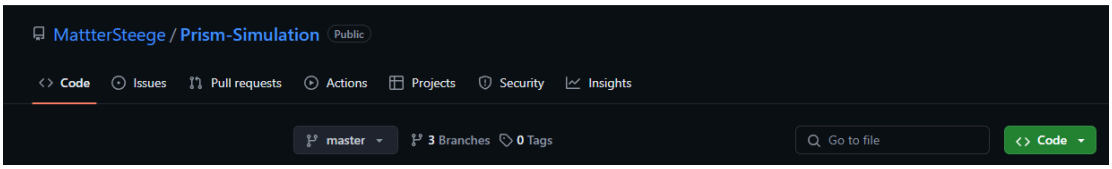


Image 1

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

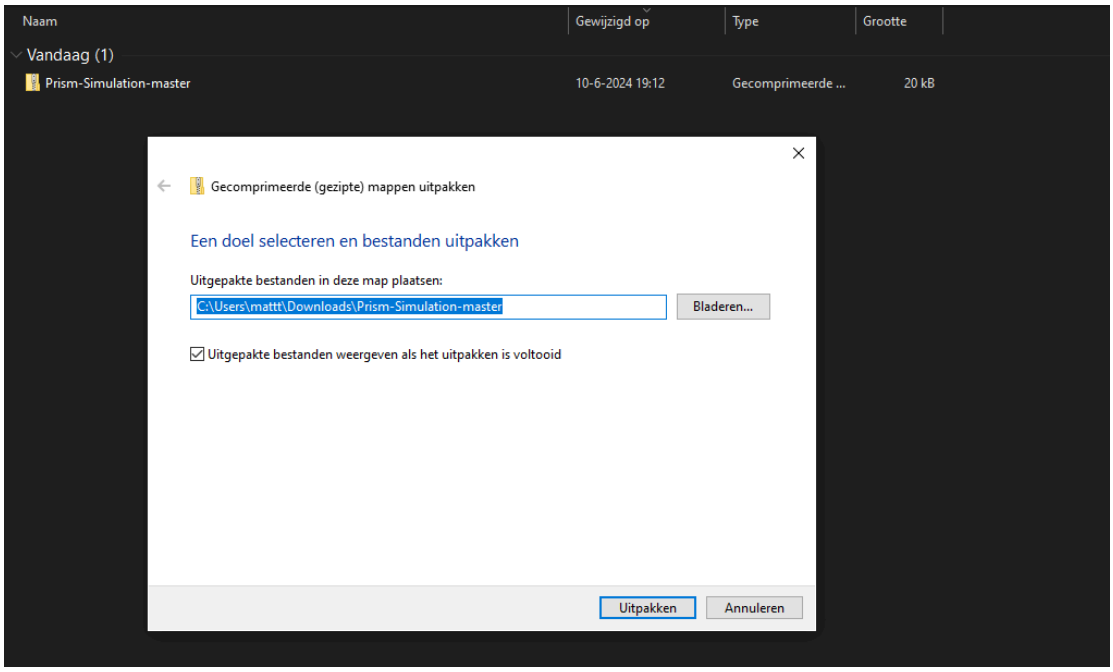


Image 2

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

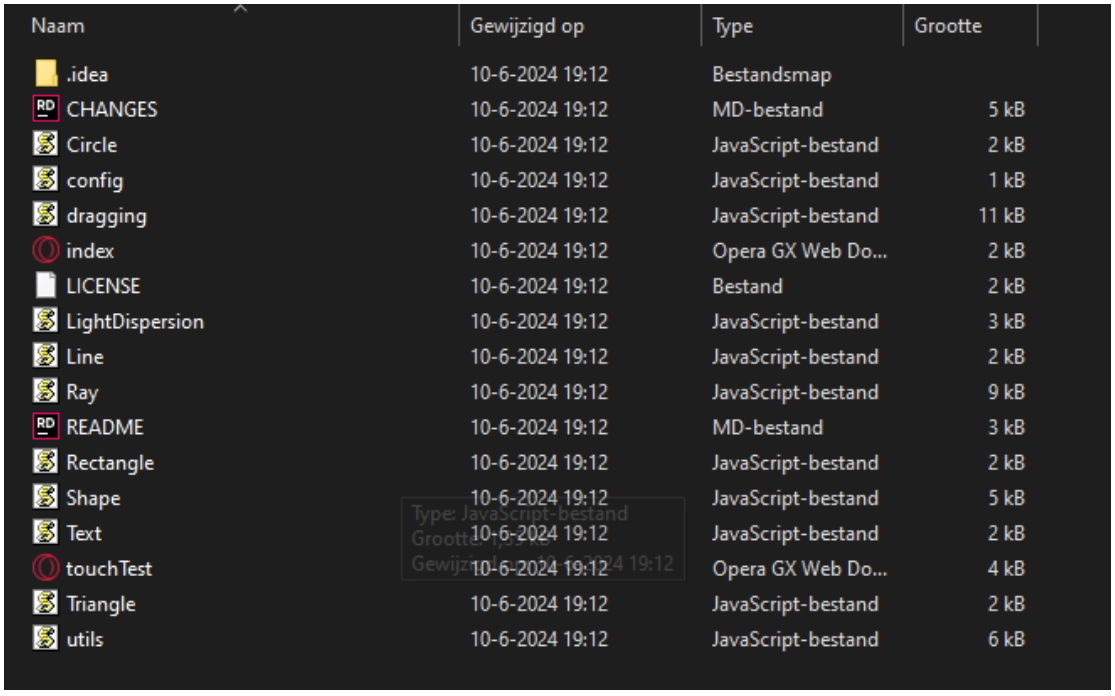


Image 3

De simulatie 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)