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

```
Shape.prototype.intersectRay = function(ray, shape) {
    //ray
    const angleRadians = Math.atan2(ray.to.y - ray.from.y, ray.to.x -
ray.from.x);
    const tinyMovement = 0.1; // Define a very small movement value
    const x1 = ray.from.x + tinyMovement * Math.cos(angleRadians); // Move
x1 slightly closer to x2
    const y1 = ray.from.y + tinyMovement * Math.sin(angleRadians); // Move
y1 slightly closer to y2
    const x2 = ray.to.x;
    const y2 = ray.to.y;


    //shape
    const points = shape.points;
```

```
    const n = points.length;

    let intersections = [];

    if (n > 2) {
        //loop through all the edges
        let _x1, _y1, _x2, _y2,
            _x3, _y3, _x4, _y4;

        for (let i = 0; i < n; i++) {
            _x1 = points[i].x;
            _y1 = points[i].y;
            _x2 = points[(i + 1) % n].x;
            _y2 = points[(i + 1) % n].y;

            _x3 = x1;
            _y3 = y1;
            _x4 = x2;
            _y4 = y2;

            const intersection = VectorIntersectsVector(_x1, _y1, _x2, _y2,
_x3, _y3, _x4, _y4);
            if (intersection) intersections.push(intersection);
        }
    }

    if (n === 2) {
        //loop through all the edges
        let _x1, _y1, _x2, _y2,
            _x3, _y3, _x4, _y4;

        _x1 = points[0].x;
        _y1 = points[0].y;
        _x2 = points[1].x;
        _y2 = points[1].y;

        _x3 = x1;
        _y3 = y1;
        _x4 = x2;
```

```
        _y4 = y2;

        const intersection = VectorIntersectsVector(_x1, _y1, _x2, _y2, _x3,
_y3, _x4, _y4);
        if (intersection) intersections.push(intersection);
    }

    if (intersections.length === 0) {
        return;
    }

    let closestIntersection = null;
    let closestDistance = 10000;

    intersections.forEach((intersection) => {
        const distance = Math.sqrt(Math.pow(intersection.x - x1, 2) +
Math.pow(intersection.y - y1, 2));
        if (distance < closestDistance) {
            closestIntersection = intersection;
            closestDistance = distance;
        }
    });

    //get the of the ray from [closestIntersection] to [from]
    const angleOfRay = normalizeDegreeAngle(RadiansToDegrees(Math.atan2(y1 -
closestIntersection.y, x1 - closestIntersection.x)));
    const normal1 = normalizeDegreeAngle(closestIntersection.normals[0]);
    const normal2 = normalizeDegreeAngle(closestIntersection.normals[1]);

    //check which is closer and set closestIntersection.normals to that
    if (getClosestNumber(angleOfRay, [normal1, normal2]) === normal1) {
        closestIntersection.normals = normal1;
    } else {
        closestIntersection.normals = normal2;
    }

    //check if the ray is hitting the shape from the inside
```

```
    return {
        from: {
            x: x1,
            y: y1
        },
        to: {
            x: closestIntersection.x,
            y: closestIntersection.y
        },
        shape: shape,
        normal: closestIntersection.normals
    };

    function VectorIntersectsVector(x1, y1, x2, y2, x3, y3, x4, y4) {
        const den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
        if (den === 0) {
            return;
        }

        const t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / den;
        const u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / den;

        const angleDegrees1 = RadiansToDegrees(Math.atan2(y2 - y1, x2 - x1))
+ 90;
        const angleDegrees2 = RadiansToDegrees(Math.atan2(y2 - y1, x2 - x1))
- 90;


        if (t >= 0 && t <= 1 && u >= 0 && u <= 1) {
            return {
                x: x1 + t * (x2 - x1),
                y: y1 + t * (y2 - y1),
                normals: [
                    angleDegrees1,
                    angleDegrees2
                ]
            };
        }
    }
```

```
}

Shape.prototype.contains = function(mx, my) {
    // Check if a point (mx, my) is inside a polygon defined by an array of
points
    const points = this.points;

    let isInside = false;
    const n = points.length;

    for (let i = 0, j = n - 1; i < n; j = i++) {
        const xi = points[i].x;
        const yi = points[i].y;
        const xj = points[j].x;
        const yj = points[j].y;

        const intersect =
            ((yi > my) !== (yj > my)) &&
            (mx < (xj - xi) * (my - yi) / (yj - yi) + xi);

        if (intersect) {
            isInside = !isInside;
        }
    }

    return isInside;
}
```

Rectangle.js

Dit stukje code definieert een JavaScript klasse genaamd `Rectangle`, die een rechthoekige vorm vertegenwoordigt. Laten we elk onderdeel bekijken:

1. **Constructor (`Rectangle`):** De constructor wordt gebruikt om een nieuw rechthoekobject te maken. Het accepteert parameters zoals de positie van de linkerbovenhoek (`x`, `y`), de breedte (`w`) en de hoogte (`h`) van de rechthoek, de rotatiehoek (`angle`) en de vulkleur (`fill`). 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.

2. **Draw methode (`draw`):** Deze methode tekent de rechthoek op een canvascontext (`ctx`). Eerst wordt de vulling en de lijnkleur ingesteld op de opgegeven vulkleur. Vervolgens wordt een nieuw pad gestart en worden de hoekpunten van de rechthoek doorlopen om een gesloten vorm te creëren. Tot slot wordt het pad gesloten en de omtrek van de rechthoek getekend.

3. **Contains methode (`contains`):** Deze methode controleert of een gegeven punt (`mx`, `my`) zich binnen de grenzen van de rechthoek bevindt. Het controleert of de x- en y-coördinaten van het punt binnen de x- en y-bereiken van de rechthoek liggen.

4. **UpdatePoints methode (`updatePoints`):** Deze 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.

In essentie biedt deze klasse functionaliteit om rechthoekige vormen te maken, te tekenen, te controleren op puntinclusie en de hoekpunten bij te werken op basis van de huidige eigenschappen van de rechthoek.

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.

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.

Circle.js

Dit stukje code implementeert een Circle-klasse die een cirkel vertegenwoordigt en ervan afgeleide functionaliteiten biedt, zoals het tekenen van de cirkel op een canvas en het controleren of een punt binnen de cirkel valt.

Hier is wat het doet:

Cirkelresoluties: Definieert het aantal punten dat wordt gebruikt om de omtrek van de cirkel te benaderen. Dit beïnvloedt de nauwkeurigheid van de getekende cirkel. Constructormethode: Initialiseert een nieuwe cirkel met de opgegeven positie (x, y), straal (w) en vulkleur (fill). Als deze waarden niet zijn opgegeven, worden standaardwaarden gebruikt. Tekenen van de cirkel: De draw-methode tekent de cirkel op het canvas. Het stelt eerst de vullings- en lijnkleur in, begint 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. Controle of een punt binnen de cirkel valt: De contains-methode controleert of een gegeven punt (mx, my) binnen de grenzen van de cirkel valt door te berekenen of de afstand tussen het punt en het middelpunt van de cirkel kleiner is dan de straal van de cirkel. Update van de punten: 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. In essentie biedt dit stukje code een objectgeoriënteerde benadering om een cirkel te vertegenwoordigen en ermee te interageren, waardoor het gemakkelijk is om cirkelvormige vormen te tekenen, te manipuleren en er interacties mee uit te voeren binnen een JavaScript-toepassing.

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:

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

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 ) );
}
```