html, body{

margin:0;

padding:0;

background-color:#ffffff;

}

canvas{

display:block;

position:absolute;

top:0;

left:0;

}

JS(Babel)

/\*\*

\* Raindrop fragment shader, being used by PIXI.js in the EffectCanvas object

\* {{uniforms: {time: {type: string, value: number}, iResolution: {type: string, value: [\*]}}, fragment: string}}

\*/

const shaderData = {

uniforms: {

iResolution: {

type: 'v2',

value: [

window.innerWidth,

window.innerHeight,

],

},

vTextureSize: {

type: 'v2',

value: [

0,

0,

],

},

uTextureForeground: {

type: 'sampler2D',

value: null,

},

uTextureBackground: {

type: 'sampler2D',

value: null,

},

uTextureDropShine: {

type: 'sampler2D',

value: null,

},

},

fragment: `

precision mediump float;

//Textures

uniform sampler2D uTextureForeground;

uniform sampler2D uTextureBackground;

uniform sampler2D uTextureDropShine;

//Canvas image data

uniform sampler2D uSampler;

//The resolution and coordinates of the current pixel

uniform vec2 iResolution;

uniform vec2 vTextureSize;

varying vec2 vTextureCoord;

//Function to get the vec2 value of the current coordinate

vec2 texCoord(){

return vec2(gl\_FragCoord.x, iResolution.y - gl\_FragCoord.y) / iResolution;

}

//Scales the bg up and proportionally to fill the container

vec2 scaledTextureCoordinate(){

float ratioCanvas = iResolution.x / iResolution.y;

float ratioImage = vTextureSize.x / vTextureSize.y;

vec2 scale = vec2(1, 1);

vec2 offset = vec2(0, 0);

float ratioDelta = ratioCanvas - ratioImage;

if(ratioDelta >= 0.0){

scale.y = (1.0 + ratioDelta);

offset.y = ratioDelta / 2.0;

}else{

scale.x = (1.0 - ratioDelta);

offset.x = -(ratioDelta / 2.0);

}

return (texCoord() + offset) / scale;

}

//Alpha-blends two colors

vec4 blend(vec4 bg, vec4 fg){

vec3 bgm = bg.rgb \* bg.a;

vec3 fgm = fg.rgb \* fg.a;

float ia = 1.0 - fg.a;

float a = (fg.a + bg.a \* ia);

vec3 rgb;

if(a != 0.0){

rgb = (fgm + bgm \* ia) / a;

}else{

rgb = vec3(0.0,0.0,0.0);

}

return vec4(rgb,a);

}

vec2 pixel(){

return vec2(1.0, 1.0) / iResolution;

}

//Get color from fg

vec4 fgColor(){

return texture2D(uSampler, vTextureCoord);

}

void main(){

vec4 bg = texture2D(uTextureBackground, scaledTextureCoordinate());

vec4 cur = fgColor();

float d = cur.b; // "thickness"

float x = cur.g;

float y = cur.r;

float a = smoothstep(0.65, 0.7, cur.a);

vec4 smoothstepped = vec4(y, x, d, a);

vec2 refraction = (vec2(x, y) - 0.5) \* 2.0;

vec2 refractionPos = scaledTextureCoordinate() + (pixel() \* refraction \* (256.0 + (d \* 512.0)));

vec4 tex = texture2D(uTextureForeground, refractionPos);

float maxShine = 390.0;

float minShine = maxShine \* 0.18;

vec2 shinePos = vec2(0.5, 0.5) + ((1.0 / 512.0) \* refraction) \* -(minShine + ((maxShine-minShine) \* d));

vec4 shine = texture2D(uTextureDropShine, shinePos);

tex = blend(tex,shine);

vec4 fg = vec4(tex.rgb, a);

gl\_FragColor = blend(bg, fg);

}

`,

};

/\*\*

\* Application Class

\* Bootstraps the entire application and initializes all objects

\*/

class Application {

/\*\*

\* Application constructor

\*/

constructor() {

this.width = window.innerWidth;

this.height = window.innerHeight;

// Define the assets that PIXI needs to preload to use later in the application

this.loader = PIXI.loader

.add('https://stefanweck.nl/codepen/alpha.png')

.add('https://stefanweck.nl/codepen/shine.png')

.add('https://stefanweck.nl/codepen/background.jpg')

.add('https://stefanweck.nl/codepen/foreground.jpg')

.load(() => this.initialize());

}

/\*\*

\* Initialize is ran when the image loader is done loading all resources

\* @return void

\*/

initialize() {

// Create the Stats object and append it to the DOM

this.stats = new Stats();

this.stats.domElement.style.position = 'absolute';

this.stats.domElement.style.left = '0px';

this.stats.domElement.style.top = '0px';

this.stats.domElement.style.zIndex = '9000';

document.body.appendChild(this.stats.domElement);

// Create a new instance of the EffectCanvas which is going to produce all of the visuals

this.effectCanvas = new EffectCanvas(this.width, this.height, this.loader);

// Resize listener for the canvas to fill browser window dynamically

window.addEventListener('resize', () => this.resizeCanvas(), false);

// Start the initial loop function for the first time

this.loop();

}

/\*\*

\* Simple resize function. Reinitializing everything on the canvas while changing the width/height

\* @return {void}

\*/

resizeCanvas() {

this.width = window.innerWidth;

this.height = window.innerHeight;

this.effectCanvas.resize(this.width, this.height);

}

/\*\*

\* Update and render the application at least 60 times a second

\* @return {void}

\*/

loop() {

window.requestAnimationFrame(() => this.loop());

this.stats.begin();

this.effectCanvas.update(this.width, this.height);

this.effectCanvas.render();

this.stats.end();

}

}

/\*\*

\* EffectCanvas Class

\*/

class EffectCanvas {

/\*\*

\* EffectCanvas constructor

\*/

constructor(width, height, loader) {

// Create and configure the renderer

this.renderer = new PIXI.autoDetectRenderer(width, height, {

antialias: false,

transparent: false,

});

this.renderer.autoResize = true;

document.body.appendChild(this.renderer.view);

// Create a container object called the `stage`

this.stage = new PIXI.Container();

// Create a graphics object that is as big as the scene of the users window

// Else the shader won't fill the entire screen

this.background = new PIXI.Graphics();

this.background.fillAlphanumber = 0;

this.background.beginFill('0xffffff');

this.background.drawRect(0, 0, width, height);

this.background.endFill();

this.background.alpha = 0;

this.stage.addChild(this.background);

// Create the DropletManager and pass it the stage so it can insert the droplet containers into it

this.dropletManager = new DropletManager(this.stage, loader);

// Send information about the textures and the size of the background texture through the uniforms to the shader

shaderData.uniforms.uTextureDropShine.value = loader.resources['https://stefanweck.nl/codepen/shine.png'].texture;

shaderData.uniforms.uTextureBackground.value = loader.resources['https://stefanweck.nl/codepen/background.jpg'].texture;

shaderData.uniforms.uTextureForeground.value = loader.resources['https://stefanweck.nl/codepen/foreground.jpg'].texture;

shaderData.uniforms.vTextureSize.value = [

loader.resources['https://stefanweck.nl/codepen/background.jpg'].texture.width,

loader.resources['https://stefanweck.nl/codepen/background.jpg'].texture.height,

];

// Create our Pixi filter using our custom shader code

this.dropletShader = new PIXI.Filter('', shaderData.fragment, shaderData.uniforms);

// Apply it to our object

this.stage.filters = [this.dropletShader];

}

/\*\*

\* Simple resize function which redraws our graphics object that should fill the entire screen

\* @return {void}

\*/

resize(width, height) {

this.renderer.resize(width, height);

this.background.clear();

this.background.beginFill('0xffffff');

this.background.drawRect(0, 0, width, height);

this.background.endFill();

}

/\*\*

\* Updates the application and every child of the application

\* @return {void}

\*/

update(width, height) {

this.updateShader(width, height);

this.dropletManager.update(width, height);

}

/\*\*

\* Updates the uniform values in the shader

\* @return {void}

\*/

updateShader(width, height) {

this.dropletShader.uniforms.iResolution = [

width,

height,

];

}

/\*\*

\* Renders the application and every child of the application

\* @return {void}

\*/

render() {

this.renderer.render(this.stage);

}

}

/\*\*

\* DropletManager class

\*/

class DropletManager {

/\*\*

\* EffectCanvas constructor

\*/

constructor(stage, loader) {

let smallDropletAmount = 9000;

let largeDropletAmount = 200;

//Quick implementation to make sure there aren't out of this world thunderstorms on mobile

if(stage.width < 700){

smallDropletAmount = 3000;

largeDropletAmount = 150;

}

this.options = {

spawnRate: {

small: 0.6,

large: 0.05,

},

spawnsPerFrame: {

small: 200,

large: 5,

},

spawnMass: {

small: {

min: 1,

max: 2,

},

large: {

min: 7,

max: 10,

},

},

poolDroplets: {

small: {

min: smallDropletAmount - 500,

max: smallDropletAmount,

},

large: {

min: largeDropletAmount - 100,

max: largeDropletAmount,

},

},

maximumMassGravity: 17,

maximumMass: 21,

dropletGrowSpeed: 1,

dropletShrinkSpeed: 2,

dropletContainerSize: 100,

};

// Define a position matrix so we can calculate all the edges of a droplet in a single loop

this.positionMatrix = [

[-1, -1],

[1, -1],

[-1, 1],

[1, 1],

];

this.smallDroplets = [];

this.largeDroplets = [];

this.dropletSmallTexture = loader.resources['https://stefanweck.nl/codepen/alpha.png'].texture;

this.dropletLargeTexture = loader.resources['https://stefanweck.nl/codepen/alpha.png'].texture;

// Create a container for all the droplets

this.smallDropletContainer = new DropletPool(Droplet, this.dropletSmallTexture, this.options.poolDroplets.small.min, this.options.poolDroplets.small.max);

this.largeDropletContainer = new DropletPool(LargeDroplet, this.dropletLargeTexture, this.options.poolDroplets.large.min, this.options.poolDroplets.large.max);

stage.addChild(this.largeDropletContainer);

stage.addChild(this.smallDropletContainer);

}

/\*\*

\* Updates the application and every child of the application

\* @return {void}

\*/

update(width, height) {

DropletManager.removeLargeOffscreenDroplets(width, height, this.largeDroplets, this.largeDropletContainer);

// Trigger the spawn function for a small droplet as much times as is configured in the options

for (let i = 0; i < this.options.spawnsPerFrame.small; i++) {

this.spawnNewSmallDroplet(width, height);

}

// Trigger the spawn function for a large droplet as much times as is configured in the options

for (let i = 0; i < this.options.spawnsPerFrame.large; i++) {

this.spawnNewLargeDroplet(width, height);

}

// Check if we need to do anything with a large Droplet

// We don't process small droplets because they are 'dumb' objects that don't move after they've spawned

this.checkLargeDropletLogic();

}

/\*\*

\* Checks whether a big droplet hits a smaller droplet, if so, it grows by half of the smaller droplets size

\* @return {void}

\*/

checkLargeDropletLogic() {

// Store the length of the array so the for loop doesn't have to do that every run

const largeDropletsLength = this.largeDroplets.length;

for (let i = largeDropletsLength - 1; i >= 0; i--) {

this.updateLargeDropletSize(this.largeDroplets[i]);

this.checkDropletMovement(this.largeDroplets[i]);

this.checkLargeToSmallDropletCollision(this.largeDroplets[i]);

this.checkLargeToLargeDropletCollision(this.largeDroplets[i]);

this.removeLargeDroplets(i);

}

}

/\*\*

\* Function that checks if a single large Droplet should be removed

\* @param i - The current droplet that we are processing

\*/

removeLargeDroplets(i) {

if (this.largeDroplets[i].mass === 0 && this.largeDroplets[i].toBeRemoved === true) {

this.largeDropletContainer.destroy(this.largeDroplets[i]);

this.largeDroplets.splice(i, 1);

}

}

/\*\*

\* Function that updates the size of a single large Droplet

\* @param droplet

\*/

updateLargeDropletSize(droplet) {

// If a droplet needs to be removed, we have to shrink it down to 0

if (droplet.toBeRemoved === true) {

this.shrinkDropletSize(droplet);

} else {

this.growDropletSize(droplet);

}

// Update the width and height of the droplet based on the new mass of the droplet

droplet.width = droplet.mass \* 6;

droplet.height = droplet.mass \* 7;

}

/\*\*

\* Shrink a droplet based on the configured shrink speed. If it will be too small, we set the mass to 0

\* @param {LargeDroplet} droplet

\*/

shrinkDropletSize(droplet) {

if (droplet.mass - this.options.dropletShrinkSpeed <= 0) {

droplet.mass = 0;

} else {

droplet.mass -= this.options.dropletShrinkSpeed;

}

}

/\*\*

\* Grow a droplet based on the targetMass he has

\* @param {LargeDroplet} droplet

\*/

growDropletSize(droplet) {

// If a droplet has already reached its target mass, exit here

if (droplet.mass === droplet.targetMass) {

return;

}

// Check if we can grow the droplet based on the configured grow speed

if (droplet.mass + this.options.dropletGrowSpeed >= droplet.targetMass) {

droplet.mass = droplet.targetMass;

} else {

droplet.mass += this.options.dropletGrowSpeed;

}

}

/\*\*

\* Check whether a large droplet should be moving or not

\* @param {LargeDroplet} droplet

\* @return {void}

\*/

checkDropletMovement(droplet) {

// If the droplet is going to be removed at the end of this loop, don't bother checking it

if (droplet.toBeRemoved === true) {

return;

}

// Check if the droplets mass is high enough to be moving, and if the droplet is not moving yet

if (droplet.mass < this.options.maximumMassGravity && droplet.dropletVelocity.y === 0 && droplet.dropletVelocity.x === 0) {

// There's a slight chance that the droplet starts moving

if (Math.random() < 0.01) {

droplet.dropletVelocity.y = Utils.getRandomInt(0.5, 3);

}

} else if (droplet.mass < this.options.maximumMassGravity && droplet.dropletVelocity.y !== 0) {

// There's a slight chance that the droplet shifts to the left or the right, just like real droplets attach to droplets near them

if (Math.random() < 0.1) {

droplet.x += Utils.getRandomInt(-10, 10) / 10;

}

// There's a slight chance that the droplet stops moving

if (Math.random() < 0.1) {

droplet.dropletVelocity.y = 0;

}

} else if (droplet.mass >= this.options.maximumMassGravity && droplet.dropletVelocity.y < 10) {

// The droplet is falling because it is too heavy, its speed and direction are now set

droplet.dropletVelocity.y = Utils.getRandomInt(10, 20);

droplet.dropletVelocity.x = Utils.getRandomInt(-10, 10) / 10;

}

// Increase the x and y positions of the droplet based on its velocity

droplet.y += droplet.dropletVelocity.y;

droplet.x += droplet.dropletVelocity.x;

}

/\*\*

\* Checks in which small droplet arrays the large droplet is positioned

\* @param {Droplet} droplet

\*/

getDropletPresenceArray(droplet) {

// Define a set of array indexes through which we hava to search for collision

const arrayIndexes = [];

const length = this.positionMatrix.length;

// Loop through each positionMatrix to calculate the position of every edge of a droplet

for (let i = 0; i < length; i++) {

const edgePosition = {

x: Math.floor((droplet.x + ((droplet.width / 7) \* this.positionMatrix[i][0])) / this.options.dropletContainerSize),

y: Math.floor((droplet.y + ((droplet.height / 7) \* this.positionMatrix[i][1])) / this.options.dropletContainerSize),

};

// Always push the first position in the arrayIndexes array, we use that value to compare the other edges to

if (i === 0) {

arrayIndexes.push(edgePosition);

continue;

}

// If the current position differs from the first position, store the new value because that means that this is also an array we need to check for collision

if (arrayIndexes[0].x !== edgePosition.x || arrayIndexes[0].y !== edgePosition.y) {

arrayIndexes.push(edgePosition);

}

}

return arrayIndexes;

}

/\*\*

\* Check the collision between one large Droplet and all the other Droplets

\* @param droplet

\*/

checkLargeToLargeDropletCollision(droplet) {

if (droplet.toBeRemoved === true) {

return;

}

// Store the length of the droplets array so we have that valua cached in the for loop

const length = this.largeDroplets.length;

for (let i = length - 1; i >= 0; i--) {

// Don't bother checking this droplet against itself

if (droplet.x === this.largeDroplets[i].x && droplet.y === this.largeDroplets[i].y) {

continue;

}

// Calculate the difference in position for the horizontal and the vertical axis

const dx = droplet.x - this.largeDroplets[i].x;

const dy = droplet.y - this.largeDroplets[i].y;

// Calculate the distance between the current droplet and the current other droplet

const distance = Math.sqrt((dx \* dx) + (dy \* dy));

// If the distance between the droplets is close enough, the droplet colliding increases in size

if (distance <= (droplet.width / 7) + (this.largeDroplets[i].width / 7)) {

if (droplet.mass + this.largeDroplets[i].mass <= this.options.maximumMass) {

droplet.targetMass = droplet.mass + this.largeDroplets[i].mass;

} else {

droplet.targetMass = this.options.maximumMass;

}

// The other droplet should be removed at the end of this loop

this.largeDroplets[i].toBeRemoved = true;

}

}

}

/\*\*

\* Checks whether a big droplet hits a smaller droplet, if so, it grows by half of the smaller droplets size

\* @param {LargeDroplet} droplet

\* @return {void}

\*/

checkLargeToSmallDropletCollision(droplet) {

if (droplet.toBeRemoved === true) {

return;

}

// Define a set of array indexes through which we have to search for collision

const arrayIndexes = this.getDropletPresenceArray(droplet);

for (let i = 0; i < arrayIndexes.length; i++) {

// If the small droplet doesn't exist anymore, we can continue to the next value in the loop

if (typeof this.smallDroplets[arrayIndexes[i].x] === 'undefined' || typeof this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y] === 'undefined') {

continue;

}

// Store the length of the array so the for loop doesn't have to do that every run

const smallDropletsLength = this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y].length;

for (let c = smallDropletsLength - 1; c >= 0; c--) {

// Calculate the difference in position for the horizontal and the vertical axis

const dx = droplet.x - this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c].x;

const dy = droplet.y - this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c].y;

// Calculate the distance between the current droplet and the current other droplet

const distance = Math.sqrt((dx \* dx) + (dy \* dy));

// If the distance is small enough we can increase the size of the large droplet and remove the small droplet

if (distance <= (droplet.width / 7) + (this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c].width / 7)) {

if (droplet.mass + (this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c].mass / 3) <= this.options.maximumMass) {

droplet.targetMass = droplet.mass + (this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c].mass / 3);

}

// Remove the small droplet and put it back in the object pool

this.smallDropletContainer.destroy(this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y][c]);

this.smallDroplets[arrayIndexes[i].x][arrayIndexes[i].y].splice(c, 1);

}

}

}

}

/\*\*

\* Spawns a new small droplet on the screen based on the spawn chance

\* @param {number} width

\* @param {number} height

\* @return {void}

\*/

spawnNewSmallDroplet(width, height) {

// If our random value doesn't match the given spawn rate, we don't spawn a droplet

if (Math.random() > this.options.spawnRate.small) {

return;

}

// Get a new droplet object from the pool

const droplet = this.smallDropletContainer.get();

// If the pool decided that we can't add more droplets, exit here

if (droplet === null) {

return;

}

const position = {

x: Utils.getRandomInt(0, width),

y: Utils.getRandomInt(0, height),

};

const mass = Utils.getRandomInt(this.options.spawnMass.small.min, this.options.spawnMass.small.max);

const arrayIndex = {

x: Math.floor(position.x / this.options.dropletContainerSize),

y: Math.floor(position.y / this.options.dropletContainerSize),

};

// Make sure the droplet updates with a new position and radius

droplet.x = position.x;

droplet.y = position.y;

droplet.mass = mass;

droplet.width = droplet.mass \* 8;

droplet.height = droplet.mass \* 8;

if (typeof this.smallDroplets[arrayIndex.x] === 'undefined') {

this.smallDroplets[arrayIndex.x] = [];

}

if (typeof this.smallDroplets[arrayIndex.x][arrayIndex.y] === 'undefined') {

this.smallDroplets[arrayIndex.x][arrayIndex.y] = [];

}

this.smallDroplets[arrayIndex.x][arrayIndex.y].push(droplet);

}

/\*\*

\* Spawns a new large droplet on the screen based on the spawn chance

\* @param {number} width

\* @param {number} height

\* @return {void}

\*/

spawnNewLargeDroplet(width, height) {

// If our random value doesn't match the given spawn rate, we don't spawn a droplet

if (Math.random() > this.options.spawnRate.large) {

return;

}

// Get a new droplet object from the pool

const droplet = this.largeDropletContainer.get();

// If the pool decided that we can't add more droplets, exit here

if (droplet === null) {

return;

}

// Make sure the droplet updates with a new position and radius

const mass = Utils.getRandomInt(this.options.spawnMass.large.min, this.options.spawnMass.large.max);

droplet.x = Utils.getRandomInt(0, width);

droplet.y = Utils.getRandomInt(-100, height / 1.5);

droplet.mass = mass / 2;

droplet.targetMass = mass;

droplet.width = droplet.mass \* 6;

droplet.height = droplet.mass \* 7;

droplet.dropletVelocity.x = 0;

droplet.toBeRemoved = false;

this.largeDroplets.push(droplet);

}

/\*\*

\* Checks each droplet to see if it is positioned offscreen. If so, it's being pushed back into the object pool to be reused

\* @param {number} width

\* @param {number} height

\* @param {Array} dropletArray

\* @param {DropletPool} dropletContainer

\* @return {void}

\*/

static removeLargeOffscreenDroplets(width, height, dropletArray, dropletContainer) {

// Store the length of the array so the for loop doesn't have to do that every run

const length = dropletArray.length;

for (let i = length - 1; i >= 0; i--) {

if (dropletArray[i].x > width + 10 || dropletArray[i].x < -10 || dropletArray[i].y > height + 10 || dropletArray[i].y < -100) {

dropletContainer.destroy(dropletArray[i]);

dropletArray.splice(i, 1);

}

}

}

}

/\*\*

\* DropletPool class

\* Functions as an object pool so we can re-use droplets over and over again

\*/

class DropletPool extends PIXI.particles.ParticleContainer {

/\*\*

\* DropletPool constructor

\*/

constructor(ObjectToCreate, objectTexture, startingSize, maximumSize) {

super(maximumSize, {

scale: true,

position: true,

rotation: false,

uvs: false,

alpha: false,

});

this.ObjectToCreate = ObjectToCreate;

this.objectTexture = objectTexture;

this.pool = [];

this.inUse = 0;

this.startingSize = startingSize;

this.maximumSize = maximumSize;

this.initialize();

}

/\*\*

\* Initialize the initial batch of objects that we are going to use throughout the application

\* @return {void}

\*/

initialize() {

for (let i = 0; i < this.startingSize; i += 1) {

const droplet = new this.ObjectToCreate(this.objectTexture);

droplet.x = -100;

droplet.y = -100;

droplet.anchor.set(0.5);

// Add the object to the PIXI Container and store it in the pool

this.addChild(droplet);

this.pool.push(droplet);

}

}

/\*\*

\* Get an object from the object pool, checks whether there is an object left or it if may create a new object otherwise

\* @returns {object}

\*/

get() {

// Check if we have reached the maximum number of objects, if so, return null

if (this.inUse >= this.maximumSize) {

return null;

}

// We haven't reached the maximum number of objects yet, so we are going to reuse an object

this.inUse++;

// If there are still objects in the pool return the last item from the pool

if (this.pool.length > 0) {

return this.pool.pop();

}

// The pool was empty, but we are still allowed to create a new object and return that

const droplet = new this.ObjectToCreate(this.objectTexture);

droplet.x = -100;

droplet.y = -100;

droplet.anchor.set(0.5, 0.5);

// Add the object to the PIXI Container and return it

this.addChild(droplet);

return droplet;

}

/\*\*

\* Put an element back into the object pool and reset it for later use

\* @param element - The object that should be pushed back into the object pool to be reused later on

\* @return {void}

\*/

destroy(element) {

if (this.inUse - 1 < 0) {

console.error('Something went wrong, you cant remove more elements than there are in the total pool');

return;

}

// Move the droplet offscreen, we cant't set visible or rendering to false because that doesn't matter in a PIXI.ParticleContainer

// @see: https://github.com/pixijs/pixi.js/issues/1910

element.x = -100;

element.y = -100;

// Push the element back into the object pool so it can be reused again

this.inUse -= 1;

this.pool.push(element);

}

}

/\*\*

\* Droplet Class

\*/

class Droplet extends PIXI.Sprite {

/\*\*

\* Droplet constructor

\*/

constructor(texture) {

super(texture);

this.mass = 0;

}

}

/\*\*

\* LargeDroplet Class

\*/

class LargeDroplet extends Droplet {

/\*\*

\* Droplet constructor

\*/

constructor(texture) {

super(texture);

this.dropletVelocity = new PIXI.Point(0, 0);

this.toBeRemoved = false;

this.targetMass = 0;

}

}

/\*\*

\* Utilities Class has some functions that are needed throughout the entire application

\*/

class Utils {

/\*\*

\* Returns a random integer between a given minimum and maximum value

\* @param {number} min - The minimum value, can be negative

\* @param {number} max - The maximum value, can be negative

\* @return {number}

\*/

static getRandomInt(min, max) {

return Math.floor(Math.random() \* ((max - min) + 1)) + min;

}

}

/\*\*

\* Onload function is executed whenever the page is done loading, initializes the application

\*/

window.onload = () => {

// Create a new instance of the application

const application = new Application();

};