To create particles, it is like creating a Mesh. You need geometry and material. Can use buffer geometries like cube or w/e.

Can use PointsMaterial. Instead of creating a Mesh, we will create Points. It handles particles.

const particlesGeometry = new THREE.SphereBufferGeometry(1, 32, 32);

const particlesMaterial = new THREE.PointsMaterial({

    size: 0.02,

    sizeAttenuation: true

})

The sizeattenuation is basically perspective. If it is close to camera particles will appear larger. Size is just the size of the particle.

const particles = new THREE.Points(particlesGeometry, particlesMaterial);

scene.add(particles);

here is how we added a sphere of particles.

What is we want to create our own geometry?

const particles = new THREE.Points(particlesGeometry, particlesMaterial);

scene.add(particles);

//our own geometry

const count = 500;

const positions = new Float32Array(count \* 3);

for (let i = 0; i < count \* 3; i++) {

  positions[i] = Math.random();

}

particlesGeometry.setAttribute(

  "position",

  new THREE.BufferAttribute(positions, 3)

);

Here is the first step to filling the float32array with random coordinates to put the particles. This basically ends up filling a cube from 0-1 on x, y, and z axes.

But we want particles everywhere. So lets make that happen.

for (let i = 0; i < count \* 3; i++) {

  positions[i] = (Math.random() - 0.5) \* 10;

}

This is basically how we do that and it ends up looking like we are in space. Add custom

const particleTexture = textureLoader.load("/textures/particles/2

particlesMaterial.map = particleTexture;

Some particles seem to be hiding particles behind them. Lets fix that.

particlesMaterial.transparent = true;

particlesMaterial.alphaMap = particleTexture;

although we still have a bug where some are not fully transparent. This is because particles are drawn in the same order they are created. And webGL does not know which ones are in front of other ones. How we will fix this.

We can fix this using the Alpha Test.

It is a random number from 0-1 and lets GPU know when to render a pixel based on transparency.

particlesMaterial.alphaTest = 0.001;

this mostly fixes this but not fully.

The next solution is the Depth Test

particlesMaterial.depthTest = false;

While this does fix the issue, this will create bugs with other particles or objects in the scene.

The next solution is using the depthwrite.

particlesMaterial.depthWrite = false;

gives a good result and what we are looking for.

This is most of the time the right solution to use for scenes. The depthWrite.

We basically tell it not to write in the depth buffer area. Aka don’t write on objets.

The fourth idea is BLENDING

Can have diff result on end and pref impact.

particlesMaterial.blending = THREE.AdditiveBlending;

This basically adds color to particles if that are stacked ontop of each other instead of ignoring color behind a pixel. changes the color a bit

Almost look like it glows a bit. Combining lights.

We can also have different colors per particles. Basically change the particles attributes.

Full code ends up looking like

const particlesGeometry = new THREE.BufferGeometry(1, 32, 32);

const particlesMaterial = new THREE.PointsMaterial({

  size: 0.1,

  sizeAttenuation: true,

});

particlesMaterial.color = new THREE.Color(0xff88cc);

particlesMaterial.transparent = true;

particlesMaterial.alphaMap = particleTexture;

// particlesMaterial.alphaTest = 0.001;

// particlesMaterial.depthTest = false;

particlesMaterial.depthWrite = false;

particlesMaterial.blending = THREE.AdditiveBlending;

particlesMaterial.vertexColors = true;

//Points

const particles = new THREE.Points(particlesGeometry, particlesMaterial);

scene.add(particles);

//our own geometry

const count = 20000;

const positions = new Float32Array(count \* 3);

const colors = new Float32Array(count \* 3);

for (let i = 0; i < count \* 3; i++) {

  positions[i] = (Math.random() - 0.5) \* 10;

  colors[i] = Math.random();

}

particlesGeometry.setAttribute(

  "position",

  new THREE.BufferAttribute(positions, 3)

);

particlesGeometry.setAttribute("color", new THREE.BufferAttribute(colors, 3));

The funny thing about this is that the base color is actually impacting the other colors and acting as a base color.

Now we are going to animate the particles.

Make like a snow effect in the tick() function:

 particles.rotation.x = elapsedTime \* 0.2;

now we want control over each particles.

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

    const i3 = i \* 3;

    particlesGeometry.attributes.position.array[i3 + 1] = Math.sin(elapsedTime);

  }

This makes things come to a center, but we need to animate it so tell the render to update()

So we add this below the for loop in the tick() function

particlesGeometry.attributes.position.needsUpdate = true;

  for (let i = 0; i < count; i++) {

    const i3 = i \* 3;

    const x = particlesGeometry.attributes.position.array[i3 + 0];

    particlesGeometry.attributes.position.array[i3 + 1] = Math.sin(

      elapsedTime + x

    );

  }

  particlesGeometry.attributes.position.needsUpdate = true;

this is the full block for the tick() for a wave! – particle waves

You should not do this thought, because we are updating thousands and thousands of arrays. There is a better solution.

We would really do this by using a custom shader. Instead of using Points – we need to make a custom shader. Custom shaders are the best way to do an animate particles. We will learn this coming up shortly.