Creating a scroll based animation.

Learn how to use Three.JS as a background as a basic HTML page

Activate the scroll

html,

body

{

    overflow: hidden;

}

We removed the above code to allow scrolling again. And we did below

.webgl {

  position: fixed;

this keeps camera in front of view port as you scroll. Although, if you scroll too high or low, you get elastic scroll problems and see white space appear. So lets make the clear

const renderer = new THREE.WebGLRenderer({

  canvas: canvas,

  alpha: true,

});

We are adding alpha: true to make the canvas transparent to see the background. By default the alpha is 0.

Now lets change the background on the html.

html {

  background: #1e1a20;

}

Now lets add objects

const material = new THREE.MeshToonMaterial({

  color: parameters.materialColor,

});

//meshes

const mesh1 = new THREE.Mesh(new THREE.TorusGeometry(1, 0.4, 16, 60), material);

const mesh2 = new THREE.Mesh(new THREE.ConeGeometry(1, 2, 32), material);

const mesh3 = new THREE.Mesh(

  new THREE.TorusKnotGeometry(0.8, 0.35, 100, 16),

  material

);

scene.add(mesh1, mesh2, mesh3);

the toon material only appears when light is present

const directionalLight = new THREE.DirectionalLight("#ffffff", 1);

directionalLight.position.set(1, 1, 0);

scene.add(directionalLight);

after light is added, lets fix the tweaker to work. Need the onchange to see changes

gui.addColor(parameters, "materialColor").onChange(() => {

  material.color.set(parameters.materialColor);

});

Lets work on the gradient texture. We use a gradient file to fix this. The image we use is only 3 pixels and each is a different version of the color we want to display gradient. SO we need a texture loader and to provide it.

const textureLoader = new THREE.TextureLoader();

const gradientTexture = textureLoader.load("textures/gradients/3.jpg");

Now we provide it to the material

const material = new THREE.MeshToonMaterial({

  color: parameters.materialColor,

  gradientMap: gradientTexture,

});

We have an issue. The colors are blended. If there is not much light, the material will try to grab the different shades. By default WebGL will mix the colors, so we need to change that. So we need to provide a magFliter.

gradientTexture.magFilter = THREE.NearestFilter;

so this tells it to grab the closest pixel color instead of mixing!

Now lets separate the objects to the three sections. Because of FOV, the objects will stay in place regardless of screen resolution. This helps us and means we don’t need to worry about different screen sizes.

mesh1.position.y = -objectsDistance \* 0;

mesh2.position.y = -objectsDistance \* 1;

mesh3.position.y = -objectsDistance \* 2;

we also added a variable called objectsDistance and passed it a static number like 2. We change to 4 to keep separate objects off screen

Now we are going to add some life to the scene with permanent rotation.

const sectionMeshes = [mesh1, mesh2, mesh3];

We start with an array of the meshes.

for (const mesh of sectionMeshes) {

    mesh.rotation.x = elapsedTime \* 0.1;

    mesh.rotation.y = elapsedTime \* 0.12;

  }

Then we add this to the tick() function

Now we want to move camera with the scroll. We need to get the scroll value then.

const scrollY = window.scrollY;

this is a quick way to get how many pixels we have scrolled.

JS is being triggered before scrolls. We need to update the value when the user is scrolling. So we add event called scrolling. This is what it fully looks like:

let scrollY = window.scrollY;

window.addEventListener("scroll", () => {

  scrollY = window.scrollY;

  console.log(scrollY);

});

This will update the scrollY value continuously. Now lets fix camera. Each section has same size of Viewport. So when we scroll a camera height, we need to get to next screen. First we divide by size.height

  camera.position.y = -scrollY / sizes.height;

Which is the size of our viewport. This basically makes us go down one unit per viewport. But we need to go down 4 units. So here is the fixed code:

 camera.position.y = (-scrollY / sizes.height) \* objectsDistance;

Now, we have one object per Viewport distance! Now lets move the objects slightly from the center.

mesh1.position.x = 2;

mesh2.position.x = -2;

mesh3.position.x = 2;

one cool thing to add is a parallax. Its about seeing one thing from one observation point. It gives you depth of things. We are going to achieve this by making the camera move with the cursor.

First we need the position of the cursor

const cursor = {};

cursor.x = 0;

cursor.y = 0;

window.addEventListener("mousemove", () => {

  console.log("moving boi");

});

We do this to test movement. Now we add event

const cursor = {};

cursor.x = 0;

cursor.y = 0;

window.addEventListener("mousemove", (event) => {

  cursor.x = event.clientX;

  cursor.y = event.clientY;

});

There is an issue though. The values are always positive. We want our camera to be able to go left and right, but this will not do. So we normalize these values

const cursor = {};

cursor.x = 0;

cursor.y = 0;

window.addEventListener("mousemove", (event) => {

  cursor.x = event.clientX / sizes.width - .5;

  cursor.y = event.clientY / sizes.height - .5;

});

This will help us move our camera accordingly. Now lets use the values to move the camera. We add this to the tick function:

 //animate camera

  camera.position.y = (-scrollY / sizes.height) \* objectsDistance;

  const parallaxX = cursor.x;

  const parallaxY = cursor.y;

we add

camera.position.x = parallaxX;

  camera.position.y = parallaxY;

and we have two issues. Scrolling does not move the canvas now, and the camera is moving async with the mouse by oddly. We need to fix.

To fix the mouse, we just need to negate one of the parallax’s.

const parallaxY = - cursor.y;

Now for the scroll. The issue is that we are moving the camera on the x and y values. But we already changes y to do the scroll. We are going to put the camera inside a group. But we are going to apply the parallax on the group. Then the camera inside the group will still move with the scroll. Lets go

const cameraGroup = new THREE.Group();

scene.add(cameraGroup);

So now we add the camera to the group instead of to the scene.

cameraGroup.add(camera);

Now we make the group work with the cursor..? Not the make the group scroll? confused

  camera.position.y = (-scrollY / sizes.height) \* objectsDistance;

  const parallaxX = cursor.x;

  const parallaxY = -cursor.y;

  cameraGroup.position.x = parallaxX;

  cameraGroup.position.y = parallaxY;

Now this is good but not very realistic and feels too mechanical. We are going to add some “easing” or smoothing or lerping to it. And we are going to use a well-known formula.

So on each frame instead of moving camera straight to target, we are going to move a tenth closer to the destination. Then on the next frame another tenth, and another, etc, etc.

 const parallaxX = cursor.x;

  const parallaxY = -cursor.y;

  cameraGroup.position.x += (parallaxX - cameraGroup.position.x) \* 0.1;

  cameraGroup.position.y += (parallaxY - cameraGroup.position.x) \* 0.1;

There is an issue, if you test this on a high freq screen, you will call the tick() function faster and it will move faster toward object. So we need to make an adjustment so that the experience is the same across devices. So we need the delta time. Time between frames.

let previousTime = 0;

const tick = () => {

  const elapsedTime = clock.getElapsedTime();

  const deltaTime = elapsedTime - previousTime;

  previousTime = elapsedTime;

we add this to get the deltaTime and update the previous time.

Then we use the deltaTime:

 cameraGroup.position.x +=

    (parallaxX - cameraGroup.position.x) \* 0.1 \* deltaTime;

  cameraGroup.position.y +=

    (parallaxY - cameraGroup.position.y) \* 0.1 \* deltaTime;

It moves very slow because deltaTime is in seconds. And every frame is around 0.016. So we change static value to 5.

  camera.position.y = (-scrollY / sizes.height) \* objectsDistance;

  const parallaxX = cursor.x \* 0.5;

  const parallaxY = -cursor.y \* 0.5;

  cameraGroup.position.x +=

    (parallaxX - cameraGroup.position.x) \* 5 \* deltaTime;

  cameraGroup.position.y +=

    (parallaxY - cameraGroup.position.x) \* 5 \* deltaTime;

Now we are going to add particles to help the user gage depth. So lets create some particles. So lets create our own bufferGeometry.

//particles

const particlesCount = 200;

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

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

  positions[i \* 3 + 0] = Math.random();

  positions[i \* 3 + 1] = Math.random();

  positions[i \* 3 + 2] = Math.random();

}

const particlesGeometry = new THREE.BufferGeometry();

particlesGeometry.setAttribute(

  "position",

  new THREE.BufferAttribute(positions, 3)

);

//material

const particlesMaterial = new THREE.PointsMaterial({

  color: parameters.materialColor,

  sizeAttenuation: true,

  size: 0.03,

});

//Points

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

scene.add(particles);

here is how we quickly add particles to the scene. We need to fix positioning, but we have them on the screen. Lets focus on changing the x (horizontal) and z (depth), we can use random values that can be as much positive as they are negative.

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

  positions[i \* 3 + 0] = (Math.random() - 0.5) \* 10;

  positions[i \* 3 + 1] =

    objectsDistance \* 0.7 - Math.random() \* objectsDistance \* 3;

  positions[i \* 3 + 2] = (Math.random() - 0.5) \* 10;

}

We make some small adjustments to positions to get things to fill up the site. Now lets add particles to the onchange of the lilgui to be played with!

gui.addColor(parameters, "materialColor").onChange(() => {

  material.color.set(parameters.materialColor);

  particlesMaterial.color.set(parameters.materialColor);

});

Now lets work on triggered rotations. Basically, when we change viewport we want the object to do a small spin

let scrollY = window.scrollY;

let currentSection = 0;

we use the event listener to figure out what section we are in and if we are changing sections due to the round function.

window.addEventListener("scroll", () => {

  scrollY = window.scrollY;

  const newSection = Math.round(scrollY / sizes.height);

  if (newSection != currentSection) {

    currentSection = newSection;

    console.log("changes section");

  }

});

Now lets animate the objects if in section. We are going to use GSAP

npm install [gsap@3.5.1](mailto:gsap@3.5.1)

import gasp from 'gsap'

console.log(gsap);

just to make sure we got it.

window.addEventListener("scroll", () => {

  scrollY = window.scrollY;

  const newSection = Math.round(scrollY / sizes.height);

  if (newSection != currentSection) {

    currentSection = newSection;

    gsap.to(sectionMeshes[currentSection].rotation, {

      duration: 1.5,

      ease: "power2.inOut",

      x: "+=6",

      y: "+=3",

    });

  }

});

We do all of this. Easing is just how it does the animation. Duration is how long it will take. But we need to change some of the movement in the tick() function that was over writing the x and y rotation values.

 for (const mesh of sectionMeshes) {

    mesh.rotation.x += deltaTime \* 0.1;

    mesh.rotation.y += deltaTime \* 0.12;

  }

Can make it cooler by adding z change

   gsap.to(sectionMeshes[currentSection].rotation, {

      duration: 2,

      ease: "power2.inOut",

      x: "+=5",

      y: "+=2.5",

      z: "+=1.5",

    });

  }

});