Intro to Angular THREE.js

02/27/2023

Angular Three.js

Angular Three.js is a powerful combination of Angular, a popular JavaScript framework, and Three.js, a powerful library for creating 3D graphics in the browser. It provides developers with a powerful and flexible toolset for building complex and interactive 3D applications in the browser.

Index

Module 1

Components

Components play an important role in Angular Three.js by providing a way to encapsulate and organize code for different parts of the 3D scene. Each component has its own lifecycle, which includes initialization, rendering, and destruction.

By creating multiple components for different parts of the 3D scene, we can easily manage and reuse code, leading to a more modular and efficient application. Components can communicate with each other by sharing data through input and output properties.

For example, we can create a component for the camera, another component for the lighting, and another component for a 3D model. We can then combine these components to create a complex 3D scene.

In the initialization phase, we can use the ngOnInit() method to set up the 3D scene and load any necessary resources, such as textures or models. In the rendering phase, we can use the ngOnChanges() method to update the scene based on any changes to the input properties.

Overall, using components in Angular Three.js can lead to a more organized and efficient codebase, making it easier to build and maintain complex 3D scenes.

Module 1 Section 1

Definitions

Angular is a powerful JavaScript framework that allows developers to build dynamic web applications with ease. It provides a set of tools and features for creating responsive user interfaces, managing data, and handling user interactions.

THREE.js is a popular JavaScript library for creating 3D graphics and animations in web applications. It provides a wide range of features for creating complex 3D scenes, including cameras, lights, materials, and geometries.

Combined Power

When combined, Angular and THREE.js can provide a powerful platform for creating interactive 3D web applications. Here are some of the ways that Angular and THREE.js work together:

1. Component-based architecture: Angular's component-based architecture makes it easy to create reusable components for user interfaces. THREE.js can be integrated into Angular components to create 3D graphics and animations, and these components can be used throughout an application.

2. Data binding and event handling: Angular's data binding and event handling features allow developers to easily connect user interface components with data and application logic. This can be used to create dynamic and responsive 3D graphics and animations in an Angular application.
3. Dependency injection: Angular's dependency injection system makes it easy to manage dependencies between components and services. This can be used to integrate THREE.js into an Angular application and manage dependencies between different 3D graphics and animations.
4. Angular animations: Angular's built-in animation system allows developers to create sophisticated animations and transitions for user interfaces. This can be used in combination with THREE.js to create complex 3D animations and effects.

Overall, Angular and THREE.js can work together seamlessly to provide a powerful platform for creating interactive 3D web applications. Developers can leverage the strengths of both frameworks to create sophisticated and engaging user experiences that take advantage of the latest web technologies.

THREE.js 3D graphics

THREE.js is a powerful JavaScript library that provides a wide range of 3D graphics capabilities for web applications. Here are some of the most exciting features of THREE.js:

1. Scene graph: THREE.js provides a scene graph data structure that allows developers to easily create and manage complex 3D scenes. This includes support for hierarchical transformations, which make it easy to move, rotate, and scale objects within a scene.
2. Geometries: THREE.js provides a wide range of built-in geometries for creating 3D objects, including spheres, cubes, cylinders, and more. These geometries can be easily customized and combined to create more complex shapes.
3. Materials: THREE.js provides a variety of materials that can be used to give objects in a scene realistic textures and lighting effects. This includes

support for diffuse, specular, and emissive materials, as well as reflections, refractions, and more.

4. Lighting: THREE.js provides a variety of lighting options, including point lights, directional lights, and spotlights. These can be used to create realistic lighting effects in a scene, such as shadows, highlights, and ambient lighting.
5. Cameras: THREE.js provides a variety of camera types, including perspective and orthographic cameras. These can be used to control the view of a scene and create a variety of different visual effects.
6. Animation: THREE.js provides a powerful animation system that allows developers to create complex animations and transitions for objects in a scene. This includes support for keyframe animation, morph targets, and more.
7. Particles: THREE.js provides a particle system that allows developers to create and animate large numbers of particles in a scene. This can be used to create a variety of visual effects, such as smoke, fire, and explosions.

Overall, THREE.js provides a powerful set of 3D graphics capabilities for web applications. Developers can leverage these features to create sophisticated and engaging user experiences that take advantage of the latest web technologies.

User Interaction and Animations

THREE.js provides built-in support for user interaction and animations, which allows developers to create rich and engaging 3D experiences on the web. Here are some of the key features of THREE.js related to user interaction and animations:

Mouse and touch interaction: THREE.js provides built-in support for mouse and touch events, such as clicks, taps, and swipes. Developers can use these events to control the movement, rotation, and scaling of objects in a scene, as well as trigger animations and other visual effects.

Interacting with the scene using the mouse and keyboard is an essential part of creating an interactive 3D graphics application with Angular THREE.js. With these tools, you can control the position, orientation, and movement of objects in the scene, as well as change the camera view and perform other actions.

Here are some examples of how you can use the mouse and keyboard to interact with a Three.js scene in an Angular application:

Mouse Interactions

Click and drag to rotate the camera view around a central point
Hold down the left mouse button and drag to pan the camera view left or right
Hold down the right mouse button and drag to pan the camera view up or down
Use the mouse scroll wheel to zoom in and out of the scene
Keyboard Interactions

Use the arrow keys to move the camera view up, down, left, or right
Use the "W", "A", "S", and "D" keys to move the camera view forward, left, backward, or right
Use the "Q" and "E" keys to roll the camera view left or right
Use the "R" key to reset the camera view to its original position and orientation
To implement these interactions in your Angular THREE.js application, you can use event listeners to detect mouse and keyboard input and then update the position, orientation, or movement of objects in the scene accordingly. For example, you might use the "mousemove" event to detect when the user is moving the mouse and update the camera view by rotating or panning it in response. Similarly, you might use the "keydown" event to detect when the user is pressing a key and update the camera view by moving it in the corresponding direction.

Overall, interacting with the scene using the mouse and keyboard is a powerful way to create an immersive 3D graphics experience with Angular THREE.js. By giving users control over the camera view and other

aspects of the scene, you can create dynamic, interactive applications that are both engaging and fun to use.

Raycasting: THREE.js provides a powerful raycasting system that allows developers to detect when a user clicks on an object in a scene. This can be used to implement complex user interactions, such as object picking, hover effects, and more.

Animation system: THREE.js provides a comprehensive animation system that allows developers to create complex animations and transitions for objects in a scene. This includes support for keyframe animation, morph targets, and more. Animations can be triggered by user interaction events, such as clicks or taps, or they can be triggered automatically based on the state of the scene.

Interpolation: THREE.js provides built-in support for interpolation, which allows developers to smoothly animate changes to an object's position, rotation, and scaling over time. This creates a more fluid and natural-looking animation.

Particle systems: THREE.js provides a particle system that allows developers to create and animate large numbers of particles in a scene. This can be used to create a variety of visual effects, such as smoke, fire, and explosions, which can be triggered by user interaction events.

Overall, THREE.js provides a powerful set of features for user interaction and animations, which allows developers to create sophisticated and engaging 3D experiences on the web. With these tools, developers can create immersive and interactive 3D applications that take advantage of the latest web technologies.

Class Session

Session Objective: By the end of this session, learners will have an understanding of what Angular THREE.js is, and the advantages of using it for 3D web development.

Session Plan:

1. Introduction (2 minutes): Begin the session by introducing yourself and giving a brief overview of what the session will cover. Then, ask the learners if they have any prior experience with Angular or THREE.js.
2. Angular THREE.js Overview (5 minutes): Explain what Angular THREE.js is and how it combines the power of Angular with the 3D graphics capabilities of THREE.js library. Provide some examples of what can be achieved with Angular THREE.js, such as building interactive 3D games or data visualization.
3. Advantages of using Angular THREE.js (5 minutes): Discuss the advantages of using Angular THREE.js for 3D web development, such as:
    - Integration with the Angular framework, making it easy to build complex 3D scenes using components and services.
    - Access to the powerful 3D graphics capabilities of THREE.js library.
    - Built-in support for user interaction and animations.
4. Conclusion (3 minutes): Summarize the key points covered in the session and ask if learners have any questions or comments. Provide learners with additional resources for learning more about Angular THREE.js, such as online tutorials or documentation.

This session is designed to provide learners with a broad overview of what Angular THREE.js is and its advantages. It's important to emphasize the practical applications of Angular THREE.js and how it can be used to build real-world projects. Additionally, learners should be encouraged to ask questions and participate in the session to stay engaged and motivated.

Module 1 Section 2

Set up

1. First, create a new Angular component in your project using the Angular CLI by running the following command in your terminal: ng generate component my-3d-scene.

2. Next, install the Three.js library using npm by running the following command in your terminal: npm install three. This will download and install the Three.js library in your project.

3. Import the necessary Three.js classes in your my-3d-scene.component.ts file at the top of the file, like this:

```
import * as THREE from 'three';
```

Three.js scene

What is the the basic structure of a Three.js scene
The basic structure of a Three.js scene consists of three main components: a scene object, a camera object, and a renderer object. Here's a brief overview of each component:

Scene object: The scene object is the container for all the 3D objects that are being rendered. It's essentially the canvas on which the 3D graphics are drawn. The scene object can contain any number of 3D objects, such as meshes, lights, cameras, and more.

Camera object: The camera object defines the position and orientation of the "virtual camera" that's used to render the 3D scene. There are several types of cameras available in Three.js, including perspective and orthographic cameras. The camera object also defines the field of view, aspect ratio, and other settings that affect the view of the scene.

Renderer object: The renderer object is responsible for drawing the 3D graphics onto the screen. It takes the 3D scene and camera objects as input, performs the necessary calculations to project the 3D objects onto a 2D plane, and renders the resulting image onto the screen. The renderer object can be configured with various settings, such as the resolution, antialiasing, and background color of the scene.

In addition to these three main components, there are other elements that can be added to a Three.js scene, such as lights, materials, and textures. These additional elements are used to define the appearance and behavior of the 3D objects in the scene. Overall, the basic structure of a Three.js scene is designed

to provide a flexible and powerful framework for creating 3D graphics on the web.

Add a 3D object

4. In the same file, create a new method called createScene that will set up the Three.js scene. Here's an example of how to create a scene, add a cube to it, and manipulate its position and rotation:

```
createScene() {
  // Create a new scene object
  const scene = new THREE.Scene();

  // Create a new cube geometry
  const geometry = new THREE.BoxGeometry();

  // Create a new material for the cube
  const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });

  // Create a new mesh object by combining the geometry and material
  const cube = new THREE.Mesh(geometry, material);

  // Set the position of the cube
  cube.position.x = 0;
  cube.position.y = 0;
  cube.position.z = 0;

  // Set the rotation of the cube
  cube.rotation.x = 0.5;
  cube.rotation.y = 0.5;

  // Add the cube to the scene
  scene.add(cube);

  return scene;
}
```

ngOnInit() method

5. Next, in the ngOnInit() method of your component, create a new instance of the scene using the createScene() method, and add it to the component's view. Here's an example:

```
ngOnInit() {
  // Create a new scene
  const scene = this.createScene();

  // Create a new camera object
  const camera = new THREE.PerspectiveCamera(
    75, // Field of view
    window.innerWidth / window.innerHeight, // Aspect ratio
    0.1, // Near clipping plane
    1000 // Far clipping plane
  );

  // Create a new renderer object
  const renderer = new THREE.WebGLRenderer();

  // Set the size of the renderer to match the size of the window
  renderer.setSize(window.innerWidth, window.innerHeight);

  // Add the renderer to the component's view

this.rendererContainer.nativeElement.appendChild(renderer.domElement)
;

  // Render the scene using the camera and renderer
  renderer.render(scene, camera);
}
```

6. Finally, run your Angular app using the ng serve command, and you should see a green cube displayed in your scene, with its position and rotation set as specified in the createScene() method. You can experiment with changing the position and rotation values to see how they affect the appearance of the cube in the scene.

Change object Color

To change the color of a Three.js cube, you can modify the material applied to the cube. Here's an example of how to change the color of a cube:

```
// Create a red cube
const cubeGeometry = new THREE.BoxGeometry(1, 1, 1);
const cubeMaterial = new THREE.MeshBasicMaterial({ color: 0xff0000
});
const cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

// Add the cube to the scene
scene.add(cube);
```

In this example, we create a BoxGeometry for the cube, which defines the size and shape of the cube. We then create a MeshBasicMaterial for the cube, which sets the color of the cube to red using the color property and the hexadecimal value 0xff0000.

To change the color of the cube, you can modify the color property of the material. For example, to change the cube color to green, you could do this:

```
cube.material.color.set(0x00ff00);
```

Alternatively, you could create a new material with a different color and apply it to the cube:

```
const newMaterial = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
cube.material = newMaterial;
```

This creates a new MeshBasicMaterial with a green color and applies it to the cube using the material property.

Changing the color of a Three.js object is a simple process that involves modifying the material applied to the object. The MeshBasicMaterial used in this example is a simple material that only has a color property, but there are many

other types of materials available in Three.js that can be used to create more complex and realistic 3D objects.

Lights

Importance of Lighting in 3D graphics

Lighting is crucial in 3D graphics as it affects the overall appearance and realism of the scene. Without proper lighting, objects in the scene can appear flat, dull, and lacking in depth. On the other hand, with the use of appropriate lighting techniques, the scene can be transformed into a visually compelling and immersive environment.

In addition, lighting can also be used to create a certain mood or atmosphere within the scene. For example, bright and warm lighting can convey a feeling of happiness and positivity, while dark and cool lighting can create a sense of mystery or danger.

Overall, lighting is an important element in creating realistic and engaging 3D graphics.

Types of lights in Three.js

There are several types of lights available in THREE.js, each with its own unique characteristics and properties. Understanding the different types of lights and how to use them is essential for creating visually appealing and realistic 3D graphics scenes.

Here are some of the most commonly used types of lights in THREE.js:

AmbientLight: This light type provides a uniform amount of light to all objects in the scene, regardless of their position or orientation. It is typically used to simulate indirect lighting, such as light that bounces off walls or other surfaces.

PointLight: This light type emits light in all directions from a single point in space, similar to a light bulb. It can cast shadows and is often used to simulate light sources such as lamps or light fixtures.

DirectionalLight: This light type emits light in a specific direction from an infinite distance away, similar to the sun. It casts shadows and is often used to simulate natural light sources such as sunlight or moonlight.

SpotLight: This light type emits light in a cone shape from a single point in space, similar to a flashlight. It casts shadows and is often used to simulate focused light sources such as spotlights or torches.

To use these lights in your THREE.js application, you can create instances of each light type and add them to your scene using the add method. You can also adjust the properties of each light, such as its color, intensity, position, and direction, using the various properties and methods available for each light type.

For example, to create a PointLight in your scene, you can use the following code:

```
const pointLight = new THREE.PointLight(0xffffff, 1, 100);
pointLight.position.set(0, 10, 0);
scene.add(pointLight);
```

This code creates a new PointLight with a white color, an intensity of 1, and a range of 100 units. It then sets the position of the light to (0, 10, 0) and adds it to the scene.

Overall, understanding the different types of lights in THREE.js and how to use them is essential for creating realistic and visually appealing 3D graphics scenes. By experimenting with different types of lights and adjusting their properties, you can create a wide variety of lighting effects and achieve the desired look and feel for your application.

Light Properties

In Three.js, there are different types of lights, each with its own set of properties. Here are some common properties of lights in Three.js:

Position: The position of the light in 3D space.

Manipulating Positions

There are many exciting ways to manipulate a light's position within a scene in Three.js. Here are a few examples:

> **Keyframe Animation:** You can use keyframe animation to create a smooth transition of a light's position over time. You can define keyframes at different points in time and interpolate the position of the light between those keyframes.
> **User Interaction:** You can allow the user to interact with the light by moving it around the scene using the mouse or keyboard. This can create an interactive and immersive experience for the user.
> **Procedural Animation:** You can use procedural animation to create complex and interesting movements for the light. For example, you can create a light that follows a path or bounces around the scene in a random pattern.
> **Scripting:** You can use scripting to program the movement of the light. This gives you complete control over the light's position and allows you to create complex animations and behaviors.

Overall, the possibilities for manipulating a light's position within a scene are nearly limitless, and can help to create dynamic and engaging 3D graphics.

Color: The color of the light. This can be set using RGB values or by using one of the built-in color constants.

Intensity: The brightness of the light. This can be adjusted to create brighter or dimmer lighting.

Distance: The distance that the light reaches. This can be adjusted to create a larger or smaller light cone.

Angle: The angle of the light cone. This is relevant for spotlights and can be adjusted to control the size of the spotlight.

Decay: The rate at which the intensity of the light decreases with distance from the source.

Cast Shadow: Whether the light should cast shadows or not.

These properties can be used to control the appearance of the light and how it affects the scene. By adjusting these properties, you can create different lighting effects and moods within the scene. For example, a bright, warm-colored light with high intensity and a large distance can create a sunny, daytime atmosphere, while a dim, cool-colored light with a small distance and a high decay rate can create a spooky, nighttime atmosphere.

Details

1. Light colors: In Three.js, lights have a color property that determines the hue and intensity of the light. You can set the color using a THREE.Color object, which accepts either RGB or HSL values. For example, to create a red point light, you can use new THREE.PointLight(0xff0000, 1).
2. Light intensity: The intensity property of a light determines how bright it appears in the scene. The default value for most light types is 1.0, but you can adjust it to make the light brighter or dimmer. For example, to create a dimmer point light, you can use new THREE.PointLight(0xffffff, 0.5).
3. Shadows: Some light types (such as point lights and spot lights) can cast shadows in the scene. In order to enable shadows, you need to set the castShadow property of the light to true, and set up a shadow camera to capture the scene from the light's perspective. You can also configure the quality of the shadows using properties like shadowMapSize and shadowBias.
4. Light helpers: In order to visualize the position and direction of lights in the scene, Three.js provides light helper objects that display as visual aids.

For example, you can create a THREE.PointLightHelper or THREE.DirectionalLightHelper object to show the position and direction of a point light or directional light, respectively.

5. Environment maps: In addition to direct lighting from light sources, Three.js also supports environment maps that simulate reflections and ambient lighting in the scene. Environment maps are textures that are mapped onto the surfaces of objects in the scene, and can create a more realistic lighting environment. You can create an environment map using a THREE.CubeTexture object, which is made up of six images representing the six faces of a cube.

Environment maps

Environment maps are a technique used in computer graphics to create the illusion of a reflective or refractive surface. They are essentially images that are wrapped around the entire scene, providing the necessary information for objects to reflect their surroundings realistically. Environment maps can be created using a variety of methods, such as photographing real-life environments, generating computer-generated images, or capturing real-time environments using a 360-degree camera.

In Three.js, environment maps can be implemented using the CubeTexture class, which is used to define a texture map that represents the environment surrounding the scene. This cube texture is applied to reflective materials in the scene, such as mirrors or metal objects, allowing them to accurately reflect the environment around them.

Environment maps can greatly enhance the realism of a scene, as they provide an accurate representation of the reflections and lighting present in the environment. By accurately reflecting the surroundings, environment maps can also provide important visual cues to the viewer, such as the location of objects or the general lighting conditions in the scene.

In addition to reflecting the environment, environment maps can also be used to create the illusion of a refractive surface, such as water or glass. By distorting

the environment map as it passes through a refractive material, the appearance of the material can be adjusted to accurately simulate the refraction of light.

Overall, environment maps are an important technique in computer graphics for creating realistic and immersive scenes. They allow for accurate reflections and refractions, which are essential for creating convincing materials and environments in 3D graphics.

Implementing Env Maps

#Environment maps can be implemented in several ways in 3D graphics. Here are some common methods:

Cube maps: A cube map is a texture that represents the environment surrounding an object by mapping six images onto the six faces of a cube. The cube map can be applied to a reflective object, such as a chrome sphere, to create realistic reflections of the surrounding environment. In Three.js, the CubeTexture class can be used to create a cube map.

Spherical maps: A spherical map is a texture that represents the environment surrounding an object as a single spherical image. Spherical maps can be used to create reflections and lighting effects in a scene. In Three.js, the Texture class can be used to create a spherical map.

HDR maps: High Dynamic Range (HDR) maps are a type of image format that captures a wider range of brightness and color information than traditional image formats. HDR maps can be used to create realistic lighting and reflections in a scene. In Three.js, the HDRCubeTexture and HDRTexture classes can be used to work with HDR maps.

Procedural maps: Procedural maps are generated mathematically rather than being based on an actual image. They can be used to create a variety of environment effects, such as clouds or fog. In Three.js, the ShaderMaterial class can be used to create custom procedural maps.

One important thing to keep in mind when working with lights in Three.js is that they can have a significant impact on performance. Adding too many lights, or lights with high intensity or complex shadows, can slow down rendering and make the scene more difficult to interact with. It's important to find a balance between visual quality and performance when working with lights.

Demos

Cameras

Three.js provides several different camera types, each of which is suitable for different use cases. Here are some of the most commonly used camera types in Three.js:

Perspective Camera: The perspective camera is similar to the way the human eye perceives the world. It creates a sense of depth and distance by rendering objects that are closer to the camera larger than objects that are farther away. The perspective camera is the most commonly used camera in Three.js and is suitable for most 3D applications.

Orthographic Camera: The orthographic camera, also known as a parallel projection camera, does not create a sense of depth or distance. Instead, it renders objects at the same size, regardless of their distance from the camera. This makes it useful for certain types of applications, such as technical drawings, where it is important to maintain a consistent scale.

Cube Camera: The cube camera creates a 360-degree view of the scene by rendering six images from the perspective of a cube. This can be useful for creating skyboxes or reflections.

Stereo Camera: The stereo camera is used to create a stereoscopic 3D effect, which creates the illusion of depth by rendering two images from slightly different perspectives.

Each camera type has its own unique properties and settings that can be adjusted to customize its behavior. For example, the perspective camera has a field of view setting that can be used to adjust the amount of the scene that is visible, while the orthographic camera has a zoom setting that can be used to adjust the scale of the scene. Choosing the right camera type and adjusting its settings can be key to creating a compelling 3D experience in Three.js.

Positioning

Example of how you could create a perspective camera and position it in the scene:

```
// Create a camera and position it
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
camera.position.z = 5;

// Add the camera to the scene
scene.add(camera);
```

In this code, we've created a new PerspectiveCamera object with a field of view of 75 degrees, an aspect ratio that matches the window size, and a near clipping plane of 0.1 and a far clipping plane of 1000. We've then set the camera's position to (0, 0, 5) so that it is positioned 5 units away from the origin along the z-axis.

When we add the camera to the scene using scene.add(camera), it affects the view of the scene by determining what portion of the 3D world is visible in the 2D rendering. The camera acts as the "eye" of the viewer and determines what objects are visible and how they are scaled and positioned in the final image. In the case of a perspective camera, the field of view determines the angle of the "cone of vision" that the camera captures, while the near and far clipping planes determine how close or far objects can be before they are no longer visible. By adjusting the camera's properties, we can control what the viewer sees and how they perceive the 3D world.

Camera Settings

Here's an example of how you can use different camera settings to fine-tune the camera's view of the scene:

```
// Create a camera and position it
const camera = new THREE.PerspectiveCamera(45, window.innerWidth /
window.innerHeight, 1, 1000);
camera.position.z = 5;

// Add the camera to the scene
scene.add(camera);

// Change the field of view and aspect ratio
camera.fov = 60;
camera.aspect = 2;
camera.updateProjectionMatrix();
```

In this example, we first create a perspective camera with a field of view of 45 degrees, an aspect ratio that matches the window size, and a near clipping plane of 1 and a far clipping plane of 1000. We then set the camera's position to (0, 0, 5) so that it is positioned 5 units away from the origin along the z-axis, and add it to the scene.

To fine-tune the camera's view of the scene, we can change the camera's field of view and aspect ratio. In this example, we change the field of view to 60 degrees and the aspect ratio to 2. We then call camera.updateProjectionMatrix() to update the camera's internal projection matrix with the new settings.

Changing the field of view can affect how much of the scene is visible in the camera's view, with a larger field of view showing more of the scene. Changing the aspect ratio can affect the proportions of the image, with a larger aspect ratio making the image wider. By adjusting these settings, we can create a customized view of the scene that meets our needs.

Creating multiple views with multiple cameras

In THREE.js, you can create multiple views within the same scene by using multiple cameras. Each camera can have its own position, orientation, and settings, allowing you to create multiple perspectives on the same 3D scene.

To create multiple views with multiple cameras, you can follow these general steps:

Create and configure multiple cameras: You can create new camera instances using the THREE.PerspectiveCamera or THREE.OrthographicCamera classes, depending on your needs. Each camera can be configured with its own settings such as field of view, aspect ratio, near and far clipping planes, and so on.

Create and configure multiple viewports: A viewport is a rectangular area on the screen where a specific camera view will be displayed. You can create multiple viewports by dividing the screen into different sections, or by overlaying them on top of each other. Each viewport can be configured with its own dimensions and position on the screen.

Render the scene multiple times: For each camera and viewport combination, you need to render the scene separately. To do this, you can create a new renderer instance and set its camera and viewport properties accordingly. You can then call the render method on the renderer to render the scene with the specified camera and viewport.

Here's an example of how you can create multiple views with two cameras in THREE.js:

```
const camera1 = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
camera1.position.set(0, 0, 5);

const camera2 = new THREE.OrthographicCamera(-2, 2, 2, -2, 0.1,
1000);
camera2.position.set(0, 0, 5);
```

```
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const view1 = new THREE.Vector4(0, 0, window.innerWidth / 2,
window.innerHeight);
const view2 = new THREE.Vector4(window.innerWidth / 2, 0,
window.innerWidth / 2, window.innerHeight);

function render() {
  // Render the scene with Camera 1 in Viewport 1
  renderer.setViewport(view1.x, view1.y, view1.z, view1.w);
  renderer.setScissor(view1.x, view1.y, view1.z, view1.w);
  renderer.setScissorTest(true);
  renderer.render(scene, camera1);

  // Render the scene with Camera 2 in Viewport 2
  renderer.setViewport(view2.x, view2.y, view2.z, view2.w);
  renderer.setScissor(view2.x, view2.y, view2.z, view2.w);
  renderer.setScissorTest(true);
  renderer.render(scene, camera2);
}

function animate() {
  requestAnimationFrame(animate);
  render();
}

animate();
```

In this example, we create two cameras - a PerspectiveCamera and an OrthographicCamera. We also create two viewports by dividing the screen into two halves. Finally, we render the scene twice - once with Camera 1 in Viewport 1, and once with Camera 2 in Viewport 2. This results in two different views of the same scene on the screen.

Additional Features

Orthographic Camera: In addition to perspective cameras, Three.js also supports orthographic cameras. Orthographic cameras do not have a field of view and instead project the scene onto a flat plane. This can be useful for certain types of scenes, such as 2D games or architectural visualization.

Camera Controls: Three.js provides built-in camera controls that allow users to interact with the scene using the mouse or touch gestures. These controls include orbit controls, which allow the user to orbit the camera around a target point, as well as trackball controls, which allow the user to rotate, pan, and zoom the camera.

Camera Helpers: Three.js provides camera helper objects that can be added to the scene to visualize the camera's position and direction. For example, a THREE.CameraHelper object can be used to create a wireframe representation of the camera's frustum.

Stereo Cameras: Three.js also supports stereo cameras, which are used to create 3D effects such as stereoscopic vision or virtual reality. Stereo cameras consist of two cameras with slightly different positions and are used to render two images that are offset from each other to create a sense of depth.

Key Concepts

In Three.js, the camera is responsible for defining the perspective through which the scene is viewed. Specifically, it determines the position, orientation, and field of view of the virtual camera used to render the scene onto the canvas.

One important concept related to the camera is the concept of perspective projection. This is a technique used to project a 3D scene onto a 2D image plane (i.e., the canvas) from a given point of view, taking into account the distance between the camera and the objects in the scene.

In Three.js, the perspective camera is the most commonly used camera type, and it uses perspective projection to achieve the illusion of depth and distance in a 3D scene. The perspective camera has a defined field of view, which determines how wide of an angle the camera sees. This is usually measured in degrees and can be adjusted using the camera.fov property.

The aspect ratio of the camera's view is also important. It determines the ratio of the width to the height of the camera's view. This can be calculated using the width and height of the canvas, and is often set automatically by the renderer.

Another important concept related to the camera is the position and orientation of the camera in the scene. In Three.js, the position of the camera is represented by a vector (camera.position), which defines the location of the camera in 3D space. The orientation of the camera is represented by a quaternion (camera.quaternion) or a rotation vector (camera.rotation), which defines the direction that the camera is facing.

It's important to note that changing the position or orientation of the camera can dramatically affect the perspective of the scene. Moving the camera closer to or farther away from objects can change the perceived size of those objects, while rotating the camera can change the perceived direction of movement in the scene.

Overall, understanding the concepts behind the perspective of the camera is essential for working with Three.js, as it determines how the 3D scene is projected onto the 2D canvas and how the viewer perceives the scene.

Add additional 3D objects

Here's an example of how to add a cylinder and a cone to the scene, and how to manipulate their position and rotation:

```
// Create a cylinder
const cylinderGeometry = new THREE.CylinderGeometry(1, 1, 2, 32);
const cylinderMaterial = new THREE.MeshBasicMaterial({ color:
```

```
0xff0000 });
const cylinderMesh = new THREE.Mesh(cylinderGeometry,
cylinderMaterial);
cylinderMesh.position.set(2, 0, 0); // Set the position of the
cylinder
cylinderMesh.rotation.set(0, Math.PI / 2, 0); // Set the rotation of
the cylinder
scene.add(cylinderMesh); // Add the cylinder to the scene

// Create a cone
const coneGeometry = new THREE.ConeGeometry(1, 2, 32);
const coneMaterial = new THREE.MeshBasicMaterial({ color: 0x00ff00
});
const coneMesh = new THREE.Mesh(coneGeometry, coneMaterial);
coneMesh.position.set(-2, 0, 0); // Set the position of the cone
coneMesh.rotation.set(0, -Math.PI / 2, 0); // Set the rotation of the
cone
scene.add(coneMesh); // Add the cone to the scene
```

In this example, we use the THREE.CylinderGeometry and THREE.ConeGeometry classes to create the cylinder and cone, respectively. We then create THREE.Mesh objects by passing in the geometry and material for each shape. We set the position and rotation of each mesh using the position and rotation properties, and add them to the scene using the add method of the scene object. Finally, we render the scene using a renderer as before.

Note that the position and rotation of a mesh are specified using THREE.Vector3 and THREE.Euler objects, respectively. You can also use the translateX, translateY, translateZ, rotateX, rotateY, and rotateZ methods of a mesh object to modify its position and rotation.

Grouping Objects

Objects can be grouped together in Three.js using the THREE.Group class, which acts as a container for multiple meshes or other groups. Grouping objects

together can make it easier to manage complex scenes by organizing related objects and applying transformations to them as a whole.

To create a group, you can simply instantiate a THREE.Group object and add meshes or other groups to it using the add method, like so:

```
// Create a group
const group = new THREE.Group();

// Create some meshes
const mesh1 = new THREE.Mesh(geometry1, material1);
const mesh2 = new THREE.Mesh(geometry2, material2);
const mesh3 = new THREE.Mesh(geometry3, material3);

// Add the meshes to the group
group.add(mesh1);
group.add(mesh2);
group.add(mesh3);

// Add the group to the scene
scene.add(group);
```

In this example, we create a THREE.Group object called group and instantiate three meshes, mesh1, mesh2, and mesh3, with different geometries and materials. We then add the meshes to the group using the add method and add the group to the scene using the add method of the scene object.

Now, any transformations applied to the group will affect all of its child objects. For example, if we want to rotate the entire group, we can simply apply a rotation to the group object:

```
group.rotation.x += 0.01;
group.rotation.y += 0.01;
```

This will rotate all of the meshes in the group around the x and y axes by a small amount each frame.

Using groups can make it easier to manage more complex scenes with many objects and nested hierarchies, and can allow for more flexible manipulation of objects as a whole.

Post-processing effects

Post-processing effects are a powerful way to enhance the visuals of a Three.js scene. They allow you to apply various effects to the final output of the rendering pipeline, such as color grading, bloom, depth of field, and more. To use post-processing effects in Three.js, you can leverage the EffectComposer class and various effect classes provided by the postprocessing module.

To get started with post-processing effects in Three.js, you first need to create an instance of the EffectComposer class and pass in the renderer object as a parameter. Then, you can create instances of different effect classes and add them to the composer using the addPass method. Here's an example:

```
import { EffectComposer, RenderPass, BloomPass } from
'postprocessing';

const composer = new EffectComposer(renderer);
const renderPass = new RenderPass(scene, camera);
const bloomPass = new BloomPass({
  strength: 1.5,
  kernelSize: 25,
  sigma: 4.0,
  resolutionScale: 0.5
});

composer.addPass(renderPass);
composer.addPass(bloomPass);
```

In the example above, we first create an instance of the EffectComposer class using the renderer object. We then create a RenderPass instance, which will render the scene and camera into a texture. Finally, we create a BloomPass

instance with some parameters to control the bloom effect, and add both passes to the composer.

After setting up the composer, you need to render the scene using the composer's render method instead of the renderer's render method. This will render the scene into the composer's render target, apply the added passes, and finally output the result to the screen. Here's an example:

```
composer.render();
```

By default, the composer will output the final result to the screen. However, you can also access the output texture of the composer and use it as a texture for other materials, or even pass it through additional passes to create more complex effects.

Overall, using post-processing effects is a great way to add depth and visual interest to your Three.js scenes. By combining different effects and passes, you can create stunning visuals that go beyond what's possible with basic lighting and materials.

Class Session

Here's a potential design for a 30-minute class session for Module 1 Section 2 of the Angular THREE.js curriculum:

Title: Creating a Three.js Scene in Angular

Objectives:

Understand the basics of creating a Three.js scene in an Angular component
Learn how to add and manipulate 3D objects in the scene
Explore different camera types and how they affect the view of the scene
Class session:

Introduction (5 minutes)

Greet the students and briefly review the key concepts from the previous session
Explain that in this session, we will be learning how to create a Three.js scene in an Angular component and add 3D objects to the scene

Setting up the scene (10 minutes)

- Demonstrate how to import Three.js and set up a basic Angular component to display a Three.js scene
- Explain the basic structure of a Three.js scene, including the scene object, camera object, and renderer object.
- Show how to add a simple 3D object, such as a cube or sphere, to the scene and manipulate its position and rotation

Working with cameras (10 minutes)

- Discuss the different types of cameras available in Three.js, including perspective and orthographic cameras
- Demonstrate how to create and position a camera in the scene and explain how the camera affects the view of the scene
- Show how to use different camera settings, such as the field of view and aspect ratio, to fine-tune the camera's view of the scene

Adding more objects (5 minutes)

- Show how to add additional 3D objects to the scene, such as cylinders or cones, and how to manipulate their position and rotation
- Explain how objects can be grouped together to create more complex scenes

Conclusion and Q&A (5 minutes)

Review the key concepts covered in the session
Encourage students to ask questions and provide additional resources for them to explore on their own
Overall, this class session provides an introduction to creating a Three.js scene in an Angular component and explores the basics of adding and manipulating 3D objects in the scene. By the end of the session, students should have a good understanding of how to create a basic Three.js scene in Angular and how to work with different camera types to control the view of the scene.

Module 1 Section 3

Section 3: Materials and Textures in Three.js

- Overview of materials and textures in Three.js
- Applying different types of materials to 3D objects
- Loading and applying textures to 3D objects
- Creating bump maps to add texture to materials
- Combining materials and textures to create complex visual effects

Materials Overview

Materials and textures are essential components for creating realistic and visually appealing 3D scenes in Three.js. A material is an object that defines the appearance of a mesh, including its color, shininess, opacity, and other visual properties. Textures are used to provide fine details and realistic effects to a material's appearance, such as patterns, roughness, and reflections.

There are several types of materials and textures available in Three.js, including basic materials like MeshBasicMaterial, which provides a simple color for a mesh, and advanced materials like MeshStandardMaterial, which includes features like metalness, roughness, and normal mapping. Each material has its own set of properties and parameters that can be adjusted to achieve the desired effect.

Textures in Three.js can be applied to materials using different mapping techniques, such as UV mapping, which maps the texture to specific vertices of a mesh, and cube mapping, which applies the texture to the faces of a cube. Textures can also be created procedurally using built-in generators or custom algorithms.

Overall, understanding how to work with materials and textures is critical for creating visually stunning 3D scenes in Three.js.

Types of Materials

In Three.js, materials are used to define the appearance of 3D objects. Materials can be used to apply color, texture, transparency, and various other visual effects to an object. There are several different types of materials available in Three.js, each with its own properties and characteristics.

Some of the most commonly used materials include:

Basic Material: This material simply applies a solid color to the object, without any additional effects.

Lambert Material: This material simulates a matte surface, reflecting light evenly in all directions.

Phong Material: This material simulates a shiny surface, with highlights that appear in specific areas.

Standard Material: This material is similar to the Phong material, but with more advanced features like physically based rendering (PBR).

To apply a material to an object in Three.js, you first need to create a new instance of the material you want to use, and then set the material property of the object to that material. You can then adjust the various properties of the material to achieve the desired effect.

Different materials may require different properties to be set, and it's important to experiment with the different settings to achieve the desired result.

In addition to materials, textures can also be applied to 3D objects to provide additional visual detail. Textures can be used to simulate the appearance of various materials like wood, metal, or fabric, or to add patterns or images to the surface of an object.

Material Properties

In Three.js, materials are used to define the appearance of 3D objects. Each material has its own set of properties that determine how it interacts with light and other objects in the scene. Some of the common properties of materials in Angular Three.js are:

[list of material properties]

Color: Determines the base color of the material. This property is used to set the color of the material's diffuse component. The diffuse color of a material is the color that it appears to be under diffuse lighting conditions.

Opacity: Determines how transparent the material is. A value of 0 means the material is completely transparent, while a value of 1 means the material is completely opaque.

Transparency: Determines the level of transparency of the material. This property is used to create materials that appear partially transparent or translucent, allowing light to pass through them to varying degrees.

Specular: Determines the specular color of the material. The specular color is the color that is reflected by the material when light hits it.

Shininess: Determines the shininess of the material. The shininess property is used to control how shiny the material appears. A higher value means the material appears shinier, while a lower value means it appears less shiny.

Ambient: Determines the ambient color of the material. The ambient color of a material is the color that it appears to be under ambient lighting conditions.

Emissive: Determines the emissive color of the material. The emissive color is the color that the material appears to be emitting. This can be used to create materials that appear to be glowing, or to create materials that have a self-illuminating effect.

These properties can be used to create a wide range of materials, from reflective metallic surfaces to translucent glass. By adjusting the properties of materials, developers can control how objects in the scene look and interact with light.

Texture Loading

In Three.js, textures are images that are wrapped onto 3D objects to give them a more realistic appearance. Textures can be loaded from various sources such as image files, videos, or even generated procedurally.

The process of loading and applying textures to 3D objects can be broken down into the following steps:

1. Creating a texture loader: A texture loader is used to load the texture image file into the scene.

```
const textureLoader = new THREE.TextureLoader();
```

2. Loading the texture: The texture image file is loaded using the texture loader. The load method of the texture loader is used for this purpose. The path to the texture image file is passed as a parameter to this method.

```
const texture = textureLoader.load('path/to/texture.jpg');
```

3. Creating a material: A material is created for the 3D object and the loaded texture is applied to it. There are several types of materials available in Three.js such as MeshBasicMaterial, MeshStandardMaterial, MeshPhysicalMaterial, etc. The map property of the material is set to the loaded texture to apply it to the 3D object.

```
const material = new THREE.MeshStandardMaterial({ map: texture });
```

4. Creating a mesh: A mesh is created using a geometry and the material created in step 3. The mesh is then added to the scene.

```
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

Once the texture is applied to the material and the material is applied to the mesh, the 3D object will have the texture mapped onto it.

Textures can also be applied to specific parts of a 3D object using UV mapping. UV mapping is a process of mapping a 2D image onto a 3D object's surface. It

involves creating a 2D representation of the 3D object's surface and mapping the texture onto this 2D representation.

Overall, the process of loading and applying textures to 3D objects is an important aspect of creating realistic and visually appealing scenes in Three.js.

Bump Mapping

Detailed analysis of Creating bump maps to add texture to materials
Bump mapping is a technique used in computer graphics to add details to a material's surface without actually changing the geometry of the object. This technique simulates bumps, wrinkles, or other small-scale features on the surface of an object by altering the shading of the material.

To create bump maps in Three.js, we use grayscale images, where black areas represent the lowest points on the material's surface, and white areas represent the highest points. The bump map image is then applied to the material's normal map property.

The normal map property in Three.js represents the surface normals of a material, which determine how light interacts with the surface. By modifying the surface normals using the bump map, we can simulate changes to the surface's depth and create the appearance of bumps and other small-scale details.

To apply bump maps to a material in Three.js, we need to create a normal map using the grayscale bump map image. We can do this using the built-in NormalMapShader in Three.js. Once the normal map is created, we can apply it to the material's normal map property, along with other material properties like color, opacity, and shininess.

Here's an example of creating and applying a bump map to a material in Three.js:

```
// Load the bump map texture
const textureLoader = new THREE.TextureLoader();
const bumpMap = textureLoader.load('bump_map.png');
```

```
// Create the normal map from the bump map using the NormalMapShader
const normalMap = new THREE.WebGLRenderTarget(bumpMap.image.width,
bumpMap.image.height);
const normalShader = THREE.NormalMapShader;
const uniforms = THREE.UniformsUtils.clone(normalShader.uniforms);
uniforms['tDepth'].value = bumpMap;
const material = new THREE.ShaderMaterial({
  uniforms: uniforms,
  vertexShader: normalShader.vertexShader,
  fragmentShader: normalShader.fragmentShader
});

// Create the material and apply the normal map
const material = new THREE.MeshPhongMaterial({
  color: 0xffffff,
  map: texture,
  bumpMap: normalMap.texture, // Apply the normal map
  bumpScale: 0.2 // Adjust the intensity of the bump map
});

// Create the mesh and add it to the scene
const geometry = new THREE.BoxGeometry(1, 1, 1);
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

In this example, we load a bump map texture and use it to create a normal map using the NormalMapShader. We then create a material with the normal map applied to the bumpMap property and adjust the intensity of the bump map using the bumpScale property. Finally, we create a mesh with the material and add it to the scene.

Complex Visual Effects

Detailed analysis of Combining materials and textures to create complex visual effects
In Three.js, it is possible to combine multiple materials and textures to create complex visual effects. This can be achieved through the use of the MultiMaterial class, which allows you to apply multiple materials to a single mesh.

To create a MultiMaterial, you can pass an array of materials to the constructor, like this:

```
var materials = [
  new THREE.MeshBasicMaterial({ color: 0xff0000 }),  // First material
  new THREE.MeshPhongMaterial({ color: 0x00ff00 })   // Second material
];
var multiMaterial = new THREE.MultiMaterial(materials);
```

Once you have a MultiMaterial, you can apply it to a mesh like any other material:

```
var mesh = new THREE.Mesh(geometry, multiMaterial);
```

In addition to combining materials, you can also combine textures to create complex visual effects. For example, you can use a normal map to add surface detail to a material, and then apply a specular map to control the specular highlights on the surface. To do this, you would create a MeshPhongMaterial with both a normal map and a specular map:

```
var material = new THREE.MeshPhongMaterial({
  map: diffuseMap,
```

```
  normalMap: normalMap,
  specularMap: specularMap,
  shininess: 50
});
```

By combining multiple materials and textures in this way, you can create a wide range of visual effects in your Three.js scenes.

Basics of Three.js Renderer

Review the basics of Three.js renderer please
Sure, here's a brief overview of the Three.js renderer:

The renderer is an essential component of any Three.js scene, responsible for rendering the scene and its objects onto the HTML5 canvas. It takes in a scene and a camera, and outputs a 2D image that represents the view of the 3D scene from the perspective of the camera.

In Three.js, the WebGLRenderer is the most commonly used renderer. It uses the WebGL API to create hardware-accelerated 3D graphics, making it fast and efficient. However, there are other renderers available as well, such as the CanvasRenderer and SVGRenderer, which use different rendering technologies and may have different capabilities and performance characteristics.

The WebGLRenderer is created with a call to the THREE.WebGLRenderer constructor, which takes in an options object that can specify various settings such as the canvas to use for rendering, the antialiasing mode, and the gamma correction factor. Once the renderer is created, it needs to be added to the HTML document using the renderer.domElement property.

Before rendering the scene, the renderer needs to be told to update its size and aspect ratio based on the size of the viewport. This can be done using the renderer.setSize(width, height) method. Additionally, the renderer needs to be told to clear the canvas before each render, using the renderer.clear() method.

Finally, to render the scene, the renderer.render(scene, camera) method is called, with the scene and camera as parameters. The renderer will then iterate

over all the objects in the scene and render them onto the canvas, taking into account the position, rotation, and other properties of each object.

Overall, the renderer is a critical component of any Three.js scene, responsible for transforming the 3D scene into a 2D image that can be displayed on a screen. Understanding its basic concepts and usage is essential for working with Three.js.

Module 1 Section 4

Section 4: Advanced Topics in Three.js

- Using particle systems to create special effects
- Animating 3D objects in a scene
- Importing 3D models from external sources
- Optimizing performance in Three.js applications

Particle Systems

Particle systems are a powerful tool in Three.js that allows developers to create special effects such as smoke, fire, explosions, and more.

In Three.js, particle systems are created using a combination of geometries, materials, and textures. The particles themselves are represented as small points in space, with each point having its own unique position, velocity, and lifespan.

To create a particle system, developers first define a geometry that determines the shape and position of the particles. This geometry can be any of the built-in geometries in Three.js or a custom geometry created by the developer.

Next, a material is created that defines the appearance of the particles. This material can be a simple color, an image-based texture, or a combination of both. To create more complex effects, developers can use shaders to create custom materials that respond to lighting and other environmental factors.

Once the geometry and material have been defined, the particles themselves are generated using a particle system object. This object controls the behavior of the particles, including their movement, lifespan, and interaction with other objects in the scene.

One of the key benefits of using particle systems in Three.js is that they can be easily animated and manipulated in real-time. Developers can use code to change the position, velocity, and other properties of individual particles, allowing for a wide range of creative possibilities.

Overall, particle systems are a powerful tool in Three.js that can be used to create a wide range of special effects and visualizations. By combining geometries, materials, and textures, developers can create complex and dynamic particle systems that add a new level of interactivity and realism to their projects.

Position, Velocity, Lifespan

In the context of particle systems, position, velocity, and lifespan are important concepts to understand.

Position refers to the location of a particle in 3D space at a given moment in time. It can be set initially to a specific location or randomized within a certain range.

Velocity is the rate of change of position over time, which determines how fast a particle moves and in what direction. It can also be set initially or randomized within a range.

Lifespan refers to the duration of time a particle remains visible in the scene before it disappears. It can be set to a specific value or randomized within a range.

By manipulating these parameters, we can create a wide variety of effects such as fireworks, smoke, or rain. For example, to create a realistic smoke effect, we would set the initial position of the particles to a specific location, give them an

initial velocity to move upwards, and set the lifespan to a certain value that would fade out the particles over time.

Particle systems can also be combined with other effects such as textures or materials to create even more complex and visually appealing results.

Demo: Smoke

Here's an advanced code example and breakdown of creating smoke in Angular Three.js:

First, we need to create a geometry for our smoke particles. We can use THREE.BufferGeometry to create the geometry and THREE.BufferAttribute to specify the position, velocity, and lifespan attributes for each particle:

```
const geometry = new THREE.BufferGeometry();

// Create arrays to hold particle data
const positions = [];
const velocities = [];
const lifespans = [];

// Add particles to the geometry
for (let i = 0; i < numParticles; i++) {
  // Randomly distribute particles within a cube
  const x = Math.random() * cubeSize - cubeSize / 2;
  const y = Math.random() * cubeSize - cubeSize / 2;
  const z = Math.random() * cubeSize - cubeSize / 2;

  // Add position data for this particle
  positions.push(x, y, z);

  // Add velocity data for this particle
  velocities.push(0, 0, 0);

  // Add lifespan data for this particle
  const lifespan = Math.random() * maxLifespan;
```

```
  lifespans.push(lifespan);
}

// Add the attributes to the geometry
geometry.setAttribute('position', new
THREE.Float32BufferAttribute(positions, 3));
geometry.setAttribute('velocity', new
THREE.Float32BufferAttribute(velocities, 3));
geometry.setAttribute('lifespan', new
THREE.Float32BufferAttribute(lifespans, 1));
```

Next, we can create a material for our smoke particles. We can use THREE.ShaderMaterial to create a custom material that will animate the particles over time:

```
const material = new THREE.ShaderMaterial({
  uniforms: {
    time: { value: 0.0 },
  },
  vertexShader: `
    attribute vec3 velocity;
    attribute float lifespan;

    uniform float time;

    void main() {
      // Calculate the new position of this particle
      vec3 newPosition = position + velocity * time;

      // If the particle has exceeded its lifespan, reset it to its
initial position
      if (time >= lifespan) {
        newPosition = vec3(0.0);
      }

      // Set the position of this vertex
      gl_Position = projectionMatrix * modelViewMatrix *
vec4(newPosition, 1.0);
```

```
    }
  `,
  fragmentShader: `
    void main() {
      // Set the color of this fragment to white
      gl_FragColor = vec4(1.0);
    }
  `,
});
```

In the vertex shader, we use the velocity and lifespan attributes to calculate the new position of each particle based on its current velocity and the amount of time that has elapsed. If a particle has exceeded its lifespan, we reset it to its initial position. In the fragment shader, we simply set the color of each particle to white.

Finally, we can create a THREE.Points object to display our smoke particles:

```
const points = new THREE.Points(geometry, material);
scene.add(points);
```

To animate the particles, we can update the time uniform in the material every frame:

```
function animate() {
  requestAnimationFrame(animate);

  const time = performance.now() / 1000; // Convert to seconds
  material.uniforms.time.value = time;

  renderer.render(scene, camera);
}
```

This will cause the particles to move and fade out over time, creating a realistic smoke effect.

Potential Errors

There are several potential errors that can occur when working with particle systems in Three.js, including:

Performance issues: Particle systems can be very computationally expensive, especially when there are a large number of particles. This can lead to performance issues on lower-end devices or when rendering complex scenes.

Memory leaks: When working with particle systems, it's important to properly manage memory allocation and deallocation to prevent memory leaks. This can be a complex task, especially when dealing with large numbers of particles.

Z-fighting: When particles are rendered very close together, they can "fight" for the same depth buffer values, leading to visual artifacts and flickering.

Alpha blending issues: When rendering transparent particles, alpha blending can lead to issues with depth sorting and visual artifacts. This can be especially problematic when particles overlap or are rendered in front of other objects.

Scaling issues: Scaling the particle system can cause issues with particle sizes and positions, and can affect the overall look of the effect. It's important to test the particle system at different scales to ensure it works as intended.

Animating 3D objects

Lecture Outline

Animation is the process of creating the illusion of motion and change by rapidly displaying a sequence of static images that minimally differ from each other.

Types of Animations

In Three.js, there are different types of animations that we can use to animate 3D objects in a scene. These include:

Translation animation: This type of animation involves moving an object from one position to another position over a specific period of time.
Rotation animation: This type of animation involves rotating an object around a specific axis over a specific period of time.
Scaling animation: This type of animation involves changing the size of an object over a specific period of time.
Morph animation: This type of animation involves morphing one shape into another shape over a specific period of time.

Animated Scene in Three.js

A. Setting up the scene and camera

To start animating objects in Three.js, we first need to create a scene and a camera. The scene is where all the 3D objects will be placed, while the camera is used to view the scene. We can create a scene using the following code:

```
const scene = new THREE.Scene();
```

We can also create a camera using the following code:

```
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000 );
camera.position.z = 5;
```

B. Adding an object to the scene

After creating a scene and a camera, we can now add an object to the scene. In this example, we'll add a cube to the scene using the following code:

```
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
const cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```

C. Animating the object

Now that we have an object in the scene, we can start animating it. In this example, we'll create a rotation animation for the cube using the following code:

```
function animate() {
  requestAnimationFrame( animate );
  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;
  renderer.render( scene, camera );
}
animate();
```

This code will continuously rotate the cube on the x and y axis, giving the illusion of motion.

Animation Loop

Intro

I. Introduction to the animation loop
A. Definition and explanation of animation loop

Animation loop is a process in Three.js that updates the scene and renders it at a certain frame rate. It involves a continuous cycle of updating the state of the objects in the scene, rendering them, and then repeating the process.
B. Code demo to create an animation loop

Let's start by creating a basic scene with a cube and a camera:

```
// create scene
const scene = new THREE.Scene();

// create camera
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
camera.position.z = 5;

// create cube
```

```
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

// create renderer
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

Now, we can create an animation loop by using the requestAnimationFrame() method:

```
function animate() {
  requestAnimationFrame(animate);

  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;

  renderer.render(scene, camera);
}

animate();
```

C. Explanation of how the animation loop works

animate() function

The animate() function is called repeatedly using the requestAnimationFrame() method, which requests that the browser calls the animate() function before the next repaint.
In the animate() function, we update the state of the cube by incrementing its rotation around the x and y axes.
We then call the renderer.render() method to render the updated scene with the camera.
This process repeats continuously, resulting in a smooth animation of the rotating cube.

II. Advanced animation techniques

A. Using keyframe animations

Keyframe animation involves defining specific positions and orientations of an object at different points in time, and then using interpolation to smoothly animate the object between those keyframes.
Three.js provides the KeyframeTrack and AnimationClip classes for defining keyframe animations.

B. Using morph targets

Morph targets involve defining a series of target meshes, and then interpolating between them to create an animation.
Three.js provides the MorphTarget and MorphTargetGeometry classes for working with morph targets.

C. Using skeletal animations

Skeletal animation involves defining a skeleton of bones for an object, and then animating the object by manipulating the bones.
Three.js provides the SkinnedMesh and Bone classes for working with skeletal animations.

III. Conclusion

In conclusion, the animation loop is a critical component of creating dynamic and engaging 3D scenes in Three.js.
By understanding the basic structure of the animation loop and exploring advanced animation techniques such as keyframe animations, morph targets, and skeletal animations, we can create complex and compelling animations in our Three.js projects.

Manipulating Models

Lecture Outline

I. Introduction

A. Definition and explanation of importing 3D models:
Importing 3D models refers to the process of bringing external 3D models into a Three.js scene. This allows for more complex and visually appealing 3D scenes, as designers can take advantage of pre-built models created in other software applications.

B. Advantages of importing 3D models:
Importing 3D models can save time and resources, as designers do not have to create every model from scratch. It also allows for more intricate designs, as complex models can be imported and manipulated in Three.js. Additionally, importing 3D models can increase the realism and detail of a 3D scene.

C. Supported file formats for importing 3D models:
Three.js supports various file formats for 3D models, including .obj, .fbx, .glTF, and .dae.

II. Preparing for 3D model import

A. Understanding file formats and conversion tools:
It is important to understand the file format of the 3D model to be imported, as some file formats may not be compatible with Three.js. Conversion tools may be required to convert the 3D model to a compatible format.

B. Preparing the 3D model file for import:
The 3D model file should be optimized for use in a Three.js scene. This includes reducing the polygon count and minimizing file size.

III. Loading a 3D model in Three.js

A. Overview of the loading process:
The Three.js loader is used to load the 3D model file into the scene. The loader options may need to be adjusted depending on the file format and model.

B. Code demo to load a 3D model in Three.js:

```
// instantiate a loader
var loader = new THREE.OBJLoader();

// load a resource
loader.load(
      // resource URL
      'model.obj',
      // called when resource is loaded
      function ( object ) {
            scene.add( object );
      },
      // called when loading is in progress
      function ( xhr ) {
            console.log( ( xhr.loaded / xhr.total
```

Let's continue with the next section:

C. Manipulating 3D models
Once we have loaded a 3D model into our scene, we can manipulate it in various ways. Three.js provides us with a number of methods to transform, rotate, and scale objects in the scene.

To manipulate a 3D model, we first need to access it. We can do this using the getObjectByName method of the Scene object. This method takes a string argument that corresponds to the name of the object we want to retrieve. We can set the name of the object when we load the 3D model.

Once we have a reference to the object, we can manipulate it using the position, rotation, and scale properties. These properties are instances of the Vector3, Euler, and Vector3 classes respectively.

Let's look at an example of how to manipulate a 3D model:

```
const loader = new GLTFLoader();

loader.load(
  'model.gltf',
  (gltf) => {
    // Set the name of the object
    gltf.scene.name = 'MyModel';

    // Add the model to the scene
    scene.add(gltf.scene);

    // Retrieve the object by name
    const myModel = scene.getObjectByName('MyModel');

    // Set the position of the model
    myModel.position.set(0, 0, 0);

    // Set the rotation of the model
    myModel.rotation.set(0, Math.PI / 2, 0);

    // Set the scale of the model
    myModel.scale.set(2, 2, 2);
  },
  (xhr) => {
    console.log((xhr.loaded / xhr.total * 100) + '% loaded');
  },
  (error) => {
    console.error('Error loading model', error);
  }
);
```

C. Explanation of the loader and its options

The loader in Three.js is responsible for importing and loading external 3D models into the scene. There are several types of loaders available, depending on the file format of the model being loaded.

The most commonly used loader is the GLTFLoader, which is used to load models in the .gltf or .glb format. This loader supports features like animations, materials, and textures, and is the recommended loader for most use cases.

Other loaders available in Three.js include OBJLoader, FBXLoader, and ColladaLoader, which support loading models in the .obj, .fbx, and .dae formats, respectively. These loaders may have limitations in terms of the features they support, so it's important to choose the appropriate loader for the specific model being loaded.

When using the loader, there are several options that can be configured to customize the loading process. Some common options include:

onLoad: A function that is called when the model finishes loading.
onError: A function that is called if an error occurs during loading.
onProgress: A function that is called as the model is loading, providing progress updates.
loadingManager: An instance of LoadingManager that manages the loading process, including showing loading indicators and controlling the order in which assets are loaded.
Now let's move on to section IV.

IV. Positioning and scaling the 3D model

A. Overview of positioning and scaling in Three.js

In Three.js, positioning and scaling are used to place and resize 3D objects in the scene.

Positioning is done using the position property of the object, which is a vector representing the object's position in 3D space. Scaling is done using the scale

property, which is also a vector but represents the object's size in each dimension.

B. Code demo to position and scale a 3D model in Three.js

Let's continue with the previous code example, and add some code to position and scale the loaded model:

```
// Create a mesh object from the loaded model
const modelMesh = new THREE.Mesh(model.geometry, model.material);

// Position the model at the origin
modelMesh.position.set(0, 0, 0);

// Scale the model by a factor of 2
modelMesh.scale.set(2, 2, 2);

// Add the mesh to the scene
scene.add(modelMesh);
```

In this example, we first create a new Mesh object using the geometry and material of the loaded model. We then set the position of the mesh to the origin (0, 0, 0), and scale it by a factor of 2 in each dimension. Finally, we add the mesh to the scene.

C. Explanation of the positioning and scaling properties

As mentioned earlier, positioning and scaling are done using the position and scale properties of 3D objects in Three.js. These properties are instances of THREE.Vector3, which represents a 3D vector.

To set the position of an object, you can use the position.set() method, which takes three arguments representing the x, y, and z coordinates of the position vector. Alternatively, you can set each coordinate individually using the position.x, position.y, and position.z properties.

Similarly, to set the scale of an object, you can use the scale.set() method, which takes three arguments representing the scaling factors in each dimension. Alternatively, you can set each scaling factor individually using the scale.x, scale.y, and scale.z properties

*Using this outline [Insert outline from where it left off] Please continue diving deeper into each section starting from V. Applying materials to the 3D model

V. Applying materials to the 3D model

A. Overview of materials in Three.js

Materials in Three.js are used to add colors, textures, and other visual effects to 3D models. There are several built-in materials that can be used, such as MeshBasicMaterial, MeshStandardMaterial, and MeshPhongMaterial. These materials have different properties that can be adjusted to create different effects. Materials can also be customized by creating new instances of Material and adjusting their properties.

B. Code demo to apply materials to a 3D model in Three.js

To apply a material to a 3D model in Three.js, you need to create a new instance of the material and assign it to the mesh's material property. Here is an example of how to apply a basic material to a 3D model:

```
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 });
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

This code creates a new MeshBasicMaterial with a red color and applies it to a new mesh created from a geometry object. The mesh is then added to the scene.

C. Explanation of the material properties

There are many material properties that can be adjusted to create different visual effects. Here are some of the most common properties:

Color: Sets the base color of the material
Map: Sets a texture to use for the material
Opacity: Sets the transparency of the material

Transparent: Specifies whether the material is transparent or not
Side: Specifies which side of the mesh the material should be applied to
Wireframe: Specifies whether the material should be displayed as a wireframe or not
Shininess: Sets the shininess of the material
Specular: Sets the color of the specular highlights on the material
Emissive: Sets the emissive color of the material
EnvMap: Sets an environment map for the material

Optimizing Performance

Optimizing performance is an important consideration when developing Three.js applications, especially when dealing with complex scenes and high polygon models. There are several strategies that can be employed to improve the performance of Three.js applications.

Minimize the number of draw calls: In Three.js, a draw call occurs every time a mesh is rendered. Minimizing the number of draw calls can significantly improve the performance of the application. This can be achieved by grouping meshes that share the same material and creating a single mesh for them.

Use efficient geometry: Using efficient geometry can also improve performance. One way to achieve this is by reducing the number of polygons in a mesh using tools like Blender or Maya. Another approach is to use Three.js built-in geometries like BoxGeometry, SphereGeometry, and PlaneGeometry, which are optimized for performance.

Use LOD (Level of Detail): LOD is a technique that involves using different versions of a model with varying levels of detail, depending on the distance from the camera. This can improve performance by reducing the number of polygons that need to be rendered.

Use frustum culling: Frustum culling is a technique that involves checking if a mesh is within the camera's view frustum before rendering it. This can save resources by not rendering meshes that are not visible.

Use texture atlases: Texture atlases are a technique that involves combining multiple textures into a single texture. This can reduce the number of draw calls and improve performance.

Use WebGLRenderer instead of CanvasRenderer: WebGLRenderer is faster than CanvasRenderer because it uses WebGL, which is a hardware-accelerated graphics API. It can also support more advanced features like shadows and post-processing effects.

Use requestAnimationFrame: requestAnimationFrame is a built-in JavaScript function that can be used to synchronize animations with the browser's refresh rate. This can prevent animations from becoming choppy or stuttery.

Use Object3D as containers: Using Object3D as containers can help group and manage objects, reducing the number of draw calls.

Overall, optimizing performance in Three.js applications requires a combination of efficient use of geometry, textures, and materials, as well as utilizing techniques like LOD, frustum culling, and requestAnimationFrame.

Custom Animation Controllers using JavaScript

Creating custom animation controllers using JavaScript is a powerful way to control the playback of GLTF animations in your web app. Here are some general steps to help you get started:

Load the GLTF file

You can use a GLTF loader library such as Three.js or Babylon.js to load the GLTF file in your web app.

Create an animation mixer

Once the GLTF file is loaded, you can create an animation mixer object using the library. The mixer object is responsible for playing and blending between different animation clips.

An animation mixer is a key component in controlling the playback of animations in a 3D scene. It is responsible for blending between different animation clips and controlling the timing and speed of the animations.

Animation Mixer

Here's how an animation mixer works and how it can be implemented in JavaScript using Three.js as an example:

Loading the GLTF file: First, you need to load the GLTF file using a loader, such as the GLTFLoader from Three.js. This returns a 3D object that contains the mesh and skeleton of the model.

```
const loader = new GLTFLoader();
loader.load('model.gltf', (gltf) => {
  scene.add(gltf.scene);
  mixer = new THREE.AnimationMixer(gltf.scene);
});
Creating an animation mixer: Once the GLTF file is loaded, you can
create an instance of the AnimationMixer class from Three.js. This
mixer is responsible for controlling the playback of animations.
javascript
Copy code
let mixer;

function init() {
  // create a new animation mixer
  mixer = new THREE.AnimationMixer(scene);
```

```
}
```

Creating animation clips: You can create animation clips using a 3D modeling software such as Blender or Maya. Alternatively, you can also create them programmatically in Three.js using the AnimationClip class.

```
const clip =
THREE.AnimationClip.createFromMorphTargetSequence('animation',
morphTargets, 30);
```

Setting up animation triggers: You can set up triggers to play and stop different animation clips. For example, you can use the click event to play an animation clip when the user clicks on an object.

```
function onMouseClick() {
  const animation = mixer.clipAction(clip);
  animation.play();
}
```

Updating the mixer: Finally, you need to update the mixer object in the render loop of your app to ensure that the animations are played smoothly and accurately.

```
function animate() {
  requestAnimationFrame(animate);

  // update the mixer object
  mixer.update(deltaTime);

  renderer.render(scene, camera);
}
```

In summary, the animation mixer is a key component in controlling the playback of animations in a 3D scene. It allows you to create and blend between different animation clips and control the timing and speed of the animations. With the right implementation, you can create complex and interactive animations that enhance the user experience in your app.

Create animation clips

You can create animation clips for each state of your object using a 3D modeling software or code. These clips can include different poses, movements, and other animations.

Set up animation triggers

You can set up triggers in your app to switch between different animation clips. These triggers can be based on user interaction or other events in your app.

Control the mixer

In your JavaScript code, you can control the playback of animations by manipulating the mixer object. For example, you can use the mixer's play() method to start playing an animation clip, or use the crossFadeTo() method to blend between different clips.

Update the mixer

Finally, you can update the mixer object in your app's render loop to ensure that the animations are playing smoothly and accurately.

Overall, creating custom animation controllers using JavaScript gives you a lot of flexibility and control over the playback of GLTF animations in your web app. With the right tools and techniques, you can create complex and interactive animations that enhance the user experience.