```
import * as React from 'react'
import {
  Plane,
  Vector3,
  Vector4,
  Matrix4,
  PerspectiveCamera,
  LinearFilter,
  WebGLRenderTarget,
  DepthTexture,
  DepthFormat,
  UnsignedShortType,
  HalfFloatType,
} from 'three'
import { useFrame, useThree, extend, ThreeElements, ThreeElement } from '@react-three/fiber'

import { BlurPass } from '../materials/BlurPass'
import { MeshReflectorMaterial as MeshReflectorMaterialImpl } from
'../materials/MeshReflectorMaterial'
import { ForwardRefComponent } from '../helpers/ts-utils'

declare module '@react-three/fiber' {
  interface ThreeElements {
    meshReflectorMaterialImpl: ThreeElement<typeof MeshReflectorMaterialImpl>
  }
}

export type MeshReflectorMaterialProps = ThreeElements['meshReflectorMaterialImpl'] & {
  resolution?: number
  blur?: [number, number] | number
  reflectorOffset?: number
}

export const MeshReflectorMaterial: ForwardRefComponent<MeshReflectorMaterialProps,
MeshReflectorMaterialImpl> =
  /* @__PURE__ */ React.forwardRef<MeshReflectorMaterialImpl, MeshReflectorMaterialProps>(
    (
      {
        mixBlur = 0,
        mixStrength = 1,
        resolution = 256,
        blur = [0, 0],
        minDepthThreshold = 0.9,
        maxDepthThreshold = 1,
        depthScale = 0,
        depthToBlurRatioBias = 0.25,
        mirror = 0,
        distortion = 1,
        mixContrast = 1,
        distortionMap,
        reflectorOffset = 0,
        ...props
      },
      ref
    ) => {
      extend({ MeshReflectorMaterialImpl })
      const gl = useThree(({ gl }) => gl)
      const camera = useThree(({ camera }) => camera)
      const scene = useThree(({ scene }) => scene)
      blur = Array.isArray(blur) ? blur : [blur, blur]
      const hasBlur = blur[0] + blur[1] > 0
      const blurX = blur[0]
      const blurY = blur[1]
      const materialRef = React.useRef<MeshReflectorMaterialImpl>(null!)
      React.useImperativeHandle(ref, () => materialRef.current, [])
```

```
      const [reflectorPlane] = React.useState(() => new Plane())
      const [normal] = React.useState(() => new Vector3())
      const [reflectorWorldPosition] = React.useState(() => new Vector3())
      const [cameraWorldPosition] = React.useState(() => new Vector3())
      const [rotationMatrix] = React.useState(() => new Matrix4())
      const [lookAtPosition] = React.useState(() => new Vector3(0, 0, -1))
      const [clipPlane] = React.useState(() => new Vector4())
      const [view] = React.useState(() => new Vector3())
      const [target] = React.useState(() => new Vector3())
      const [q] = React.useState(() => new Vector4())
      const [textureMatrix] = React.useState(() => new Matrix4())
      const [virtualCamera] = React.useState(() => new PerspectiveCamera())

      const beforeRender = React.useCallback(() => {
        const parent = (materialRef.current as any).parent || (materialRef.current as
any)?.__r3f.parent?.object
        if (!parent) return

        reflectorWorldPosition.setFromMatrixPosition(parent.matrixWorld)
        cameraWorldPosition.setFromMatrixPosition(camera.matrixWorld)
        rotationMatrix.extractRotation(parent.matrixWorld)
        normal.set(0, 0, 1)
        normal.applyMatrix4(rotationMatrix)
        reflectorWorldPosition.addScaledVector(normal, reflectorOffset)
        view.subVectors(reflectorWorldPosition, cameraWorldPosition)
        // Avoid rendering when reflector is facing away
        if (view.dot(normal) > 0) return
        view.reflect(normal).negate()
        view.add(reflectorWorldPosition)
        rotationMatrix.extractRotation(camera.matrixWorld)
        lookAtPosition.set(0, 0, -1)
        lookAtPosition.applyMatrix4(rotationMatrix)
        lookAtPosition.add(cameraWorldPosition)
        target.subVectors(reflectorWorldPosition, lookAtPosition)
        target.reflect(normal).negate()
        target.add(reflectorWorldPosition)
        virtualCamera.position.copy(view)
        virtualCamera.up.set(0, 1, 0)
        virtualCamera.up.applyMatrix4(rotationMatrix)
        virtualCamera.up.reflect(normal)
        virtualCamera.lookAt(target)
        virtualCamera.far = camera.far // Used in WebGLBackground
        virtualCamera.updateMatrixWorld()
        virtualCamera.projectionMatrix.copy(camera.projectionMatrix)
        // Update the texture matrix
        textureMatrix.set(0.5, 0.0, 0.0, 0.5, 0.0, 0.5, 0.0, 0.5, 0.0, 0.0, 0.5, 0.5, 0.0,
0.0, 0.0, 1.0)
        textureMatrix.multiply(virtualCamera.projectionMatrix)
        textureMatrix.multiply(virtualCamera.matrixWorldInverse)
        textureMatrix.multiply(parent.matrixWorld)
        // Now update projection matrix with new clip plane, implementing code from:
http://www.terathon.com/code/oblique.html
        // Paper explaining this technique: http://www.terathon.com/lengyel/Lengyel-
Oblique.pdf
        reflectorPlane.setFromNormalAndCoplanarPoint(normal, reflectorWorldPosition)
        reflectorPlane.applyMatrix4(virtualCamera.matrixWorldInverse)
        clipPlane.set(
          reflectorPlane.normal.x,
          reflectorPlane.normal.y,
          reflectorPlane.normal.z,
          reflectorPlane.constant
        )
        const projectionMatrix = virtualCamera.projectionMatrix
        q.x = (Math.sign(clipPlane.x) + projectionMatrix.elements[8]) /
projectionMatrix.elements[0]
        q.y = (Math.sign(clipPlane.y) + projectionMatrix.elements[9]) /
```

```
projectionMatrix.elements[5]
        q.z = -1.0
        q.w = (1.0 + projectionMatrix.elements[10]) / projectionMatrix.elements[14]
        // Calculate the scaled plane vector
        clipPlane.multiplyScalar(2.0 / clipPlane.dot(q))
        // Replacing the third row of the projection matrix
        projectionMatrix.elements[2] = clipPlane.x
        projectionMatrix.elements[6] = clipPlane.y
        projectionMatrix.elements[10] = clipPlane.z + 1.0
        projectionMatrix.elements[14] = clipPlane.w
      }, [camera, reflectorOffset])

      const [fbo1, fbo2, blurpass, reflectorProps] = React.useMemo(() => {
        const parameters = {
          minFilter: LinearFilter,
          magFilter: LinearFilter,
          type: HalfFloatType,
        }
        const fbo1 = new WebGLRenderTarget(resolution, resolution, parameters)
        fbo1.depthBuffer = true
        fbo1.depthTexture = new DepthTexture(resolution, resolution)
        fbo1.depthTexture.format = DepthFormat
        fbo1.depthTexture.type = UnsignedShortType
        const fbo2 = new WebGLRenderTarget(resolution, resolution, parameters)
        const blurpass = new BlurPass({
          gl,
          resolution,
          width: blurX,
          height: blurY,
          minDepthThreshold,
          maxDepthThreshold,
          depthScale,
          depthToBlurRatioBias,
        })
        const reflectorProps = {
          mirror,
          textureMatrix,
          mixBlur,
          tDiffuse: fbo1.texture,
          tDepth: fbo1.depthTexture,
          tDiffuseBlur: fbo2.texture,
          hasBlur,
          mixStrength,
          minDepthThreshold,
          maxDepthThreshold,
          depthScale,
          depthToBlurRatioBias,
          distortion,
          distortionMap,
          mixContrast,
          'defines-USE_BLUR': hasBlur ? '' : undefined,
          'defines-USE_DEPTH': depthScale > 0 ? '' : undefined,
          'defines-USE_DISTORTION': distortionMap ? '' : undefined,
        }
        return [fbo1, fbo2, blurpass, reflectorProps]
      }, [
        gl,
        blurX,
        blurY,
        textureMatrix,
        resolution,
        mirror,
        hasBlur,
        mixBlur,
        mixStrength,
        minDepthThreshold,
```

```
        maxDepthThreshold,
        depthScale,
        depthToBlurRatioBias,
        distortion,
        distortionMap,
        mixContrast,
      ])

      useFrame(() => {
        const parent = (materialRef.current as any).parent || (materialRef.current as
any)?.__r3f.parent?.object
        if (!parent) return

        parent.visible = false
        const currentXrEnabled = gl.xr.enabled
        const currentShadowAutoUpdate = gl.shadowMap.autoUpdate
        beforeRender()
        gl.xr.enabled = false
        gl.shadowMap.autoUpdate = false
        gl.setRenderTarget(fbo1)
        gl.state.buffers.depth.setMask(true)
        if (!gl.autoClear) gl.clear()
        gl.render(scene, virtualCamera)
        if (hasBlur) blurpass.render(gl, fbo1, fbo2)
        gl.xr.enabled = currentXrEnabled
        gl.shadowMap.autoUpdate = currentShadowAutoUpdate
        parent.visible = true
        gl.setRenderTarget(null)
      })

      return (
        <meshReflectorMaterialImpl
          attach="material"
          // Defines can't be updated dynamically, so we need to recreate the material
          key={
            'key' +
            reflectorProps['defines-USE_BLUR'] +
            reflectorProps['defines-USE_DEPTH'] +
            reflectorProps['defines-USE_DISTORTION']
          }
          ref={materialRef}
          {...reflectorProps}
          {...props}
        />
      )
    }
  )
```