```
/** Author: @N8Programs https://github.com/N8python
 *    https://gist.github.com/N8python/eb42d25c7cd00d12e965ac9cba544317
 *  Inspired by: @ore_ukonpower and http://next.junni.co.jp
 *    https://github.com/junni-
inc/next.junni.co.jp/blob/master/src/ts/MainScene/World/Sections/Section2/Transparents/Transp
arent/shaders/transparent.fs
 */

import * as THREE from 'three'
import * as React from 'react'
import { extend, ThreeElements, useFrame } from '@react-three/fiber'
import { useFBO } from './Fbo'
import { DiscardMaterial } from '../materials/DiscardMaterial'
import { ForwardRefComponent } from '../helpers/ts-utils'

type MeshTransmissionMaterialType = Omit<
  ThreeElements['meshPhysicalMaterial'],
  'args' | 'roughness' | 'thickness' | 'transmission'
> & {
  /* Transmission, default: 1 */
  transmission?: number
  /* Thickness (refraction), default: 0 */
  thickness?: number
  /* Roughness (blur), default: 0 */
  roughness?: number
  /* Chromatic aberration, default: 0.03 */
  chromaticAberration?: number
  /* Anisotropy, default: 0.1 */
  anisotropy?: number
  /* AnisotropicBlur, default: 0.1 */
  anisotropicBlur?: number
  /* Distortion, default: 0 */
  distortion?: number
  /* Distortion scale, default: 0.5 */
  distortionScale?: number
  /* Temporal distortion (speed of movement), default: 0.0 */
  temporalDistortion?: number
  /** The scene rendered into a texture (use it to share a texture between materials),
default: null  */
  buffer?: THREE.Texture
  /** Internals */
  time?: number
  /** Internals */
  args?: [samples: number, transmissionSampler: boolean]
}

export type MeshTransmissionMaterialProps = Omit<MeshTransmissionMaterialType, 'ref' |
'args'> & {
  /** transmissionSampler, you can use the threejs transmission sampler texture that is
   *  generated once for all transmissive materials. The upside is that it can be faster if
you
   *  use multiple MeshPhysical and Transmission materials, the downside is that transmissive
materials
   *  using this can't see other transparent or transmissive objects, default: false */
  transmissionSampler?: boolean
  /** Render the backside of the material (more cost, better results), default: false */
  backside?: boolean
  /** Backside thickness (when backside is true), default: 0 */
  backsideThickness?: number
  backsideEnvMapIntensity?: number
  /** Resolution of the local buffer, default: undefined (fullscreen) */
  resolution?: number
  /** Resolution of the local buffer for backfaces, default: undefined (fullscreen) */
  backsideResolution?: number
  /** Refraction samples, default: 6 */
  samples?: number
  /** Buffer scene background (can be a texture, a cubetexture or a color), default: null */
  background?: THREE.Texture | THREE.Color
}

interface Uniform<T> {
  value: T
}

interface Shader {
  uniforms: { [uniform: string]: Uniform<any> }
  vertexShader: string
  fragmentShader: string
}

declare module '@react-three/fiber' {
  interface ThreeElements {
    meshTransmissionMaterial: MeshTransmissionMaterialType
  }
}

class MeshTransmissionMaterialImpl extends THREE.MeshPhysicalMaterial {
  uniforms: {
    chromaticAberration: Uniform<number>
    transmission: Uniform<number>
    transmissionMap: Uniform<THREE.Texture | null>
    _transmission: Uniform<number>
    thickness: Uniform<number>
    roughness: Uniform<number>
    thicknessMap: Uniform<THREE.Texture | null>
    attenuationDistance: Uniform<number>
    attenuationColor: Uniform<THREE.Color>
    anisotropicBlur: Uniform<number>
    time: Uniform<number>
    distortion: Uniform<number>
    distortionScale: Uniform<number>
    temporalDistortion: Uniform<number>
    buffer: Uniform<THREE.Texture | null>
  }

  constructor(samples = 6, transmissionSampler = false) {
    super()

    this.uniforms = {
      chromaticAberration: { value: 0.05 },
      // Transmission must always be 0, unless transmissionSampler is being used
      transmission: { value: 0 },
      // Instead a workaround is used, see below for reasons why
      _transmission: { value: 1 },
      transmissionMap: { value: null },
      // Roughness is 1 in THREE.MeshPhysicalMaterial but it makes little sense in a
transmission material
      roughness: { value: 0 },
      thickness: { value: 0 },
      thicknessMap: { value: null },
      attenuationDistance: { value: Infinity },
      attenuationColor: { value: new THREE.Color('white') },
      anisotropicBlur: { value: 0.1 },
      time: { value: 0 },
      distortion: { value: 0.0 },
      distortionScale: { value: 0.5 },
      temporalDistortion: { value: 0.0 },
      buffer: { value: null },
    }

    this.onBeforeCompile = (shader: Shader & { defines: { [key: string]: string } }) => {
      shader.uniforms = {
        ...shader.uniforms,
        ...this.uniforms,
      }

      // Fix for r153-r156 anisotropy chunks
      // https://github.com/mrdoob/three.js/pull/26716
      if ((this as any).anisotropy > 0) shader.defines.USE_ANISOTROPY = ''

      // If the transmission sampler is active inject a flag
      if (transmissionSampler) shader.defines.USE_SAMPLER = ''
      // Otherwise we do use use .transmission and must therefore force USE_TRANSMISSION
      // because threejs won't inject it for us
      else shader.defines.USE_TRANSMISSION = ''

      // Head
      shader.fragmentShader =
        /*glsl*/ `
      uniform float chromaticAberration;
      uniform float anisotropicBlur;
      uniform float time;
      uniform float distortion;
      uniform float distortionScale;
      uniform float temporalDistortion;
      uniform sampler2D buffer;

      vec3 random3(vec3 c) {
        float j = 4096.0*sin(dot(c,vec3(17.0, 59.4, 15.0)));
        vec3 r;
        r.z = fract(512.0*j);
        j *= .125;
        r.x = fract(512.0*j);
        j *= .125;
        r.y = fract(512.0*j);
        return r-0.5;
      }

      uint hash( uint x ) {
        x += ( x << 10u );
        x ^= ( x >>  6u );
        x += ( x <<  3u );
        x ^= ( x >> 11u );
        x += ( x << 15u );
        return x;
      }

      // Compound versions of the hashing algorithm I whipped together.
      uint hash( uvec2 v ) { return hash( v.x ^ hash(v.y)                         ); }
      uint hash( uvec3 v ) { return hash( v.x ^ hash(v.y) ^ hash(v.z)             ); }
      uint hash( uvec4 v ) { return hash( v.x ^ hash(v.y) ^ hash(v.z) ^ hash(v.w) ); }

      // Construct a float with half-open range [0:1] using low 23 bits.
      // All zeroes yields 0.0, all ones yields the next smallest representable value below
1.0.
      float floatConstruct( uint m ) {
        const uint ieeeMantissa = 0x007FFFFFu; // binary32 mantissa bitmask
        const uint ieeeOne      = 0x3F800000u; // 1.0 in IEEE binary32
        m &= ieeeMantissa;                     // Keep only mantissa bits (fractional part)
        m |= ieeeOne;                          // Add fractional part to 1.0
        float  f = uintBitsToFloat( m );       // Range [1:2]
        return f - 1.0;                        // Range [0:1]
      }

      // Pseudo-random value in half-open range [0:1].
      float randomBase( float x ) { return floatConstruct(hash(floatBitsToUint(x))); }
      float randomBase( vec2  v ) { return floatConstruct(hash(floatBitsToUint(v))); }
      float randomBase( vec3  v ) { return floatConstruct(hash(floatBitsToUint(v))); }
      float randomBase( vec4  v ) { return floatConstruct(hash(floatBitsToUint(v))); }
      float rand(float seed) {
        float result = randomBase(vec3(gl_FragCoord.xy, seed));
        return result;
      }

      const float F3 =  0.3333333;
      const float G3 =  0.1666667;

      float snoise(vec3 p) {
        vec3 s = floor(p + dot(p, vec3(F3)));
        vec3 x = p - s + dot(s, vec3(G3));
        vec3 e = step(vec3(0.0), x - x.yzx);
        vec3 i1 = e*(1.0 - e.zxy);
        vec3 i2 = 1.0 - e.zxy*(1.0 - e);
        vec3 x1 = x - i1 + G3;
        vec3 x2 = x - i2 + 2.0*G3;
        vec3 x3 = x - 1.0 + 3.0*G3;
        vec4 w, d;
        w.x = dot(x, x);
        w.y = dot(x1, x1);
        w.z = dot(x2, x2);
        w.w = dot(x3, x3);
        w = max(0.6 - w, 0.0);
        d.x = dot(random3(s), x);
        d.y = dot(random3(s + i1), x1);
        d.z = dot(random3(s + i2), x2);
        d.w = dot(random3(s + 1.0), x3);
        w *= w;
        w *= w;
        d *= w;
        return dot(d, vec4(52.0));
      }

      float snoiseFractal(vec3 m) {
        return 0.5333333* snoise(m)
              +0.2666667* snoise(2.0*m)
              +0.1333333* snoise(4.0*m)
              +0.0666667* snoise(8.0*m);
      }\n` + shader.fragmentShader

      // Remove transmission
      shader.fragmentShader = shader.fragmentShader.replace(
        '#include <transmission_pars_fragment>',
        /*glsl*/ `
        #ifdef USE_TRANSMISSION
          // Transmission code is based on glTF-Sampler-Viewer
          // https://github.com/KhronosGroup/glTF-Sample-Viewer
          uniform float _transmission;
          uniform float thickness;
          uniform float attenuationDistance;
          uniform vec3 attenuationColor;
          #ifdef USE_TRANSMISSIONMAP
            uniform sampler2D transmissionMap;
          #endif
          #ifdef USE_THICKNESSMAP
            uniform sampler2D thicknessMap;
          #endif
          uniform vec2 transmissionSamplerSize;
          uniform sampler2D transmissionSamplerMap;
          uniform mat4 modelMatrix;
          uniform mat4 projectionMatrix;
          varying vec3 vWorldPosition;
          vec3 getVolumeTransmissionRay( const in vec3 n, const in vec3 v, const in float
thickness, const in float ior, const in mat4 modelMatrix ) {
```

```
            // Direction of refracted light.
            vec3 refractionVector = refract( - v, normalize( n ), 1.0 / ior );
            // Compute rotation-independant scaling of the model matrix.
            vec3 modelScale;
            modelScale.x = length( vec3( modelMatrix[ 0 ].xyz ) );
            modelScale.y = length( vec3( modelMatrix[ 1 ].xyz ) );
            modelScale.z = length( vec3( modelMatrix[ 2 ].xyz ) );
            // The thickness is specified in local space.
            return normalize( refractionVector ) * thickness * modelScale;
          }
          float applyIorToRoughness( const in float roughness, const in float ior ) {
            // Scale roughness with IOR so that an IOR of 1.0 results in no microfacet
refraction and
            // an IOR of 1.5 results in the default amount of microfacet refraction.
            return roughness * clamp( ior * 2.0 - 2.0, 0.0, 1.0 );
          }
          vec4 getTransmissionSample( const in vec2 fragCoord, const in float roughness,
const in float ior ) {
            float framebufferLod = log2( transmissionSamplerSize.x ) * applyIorToRoughness(
roughness, ior );
            #ifdef USE_SAMPLER
              #ifdef texture2DLodEXT
                return texture2DLodEXT(transmissionSamplerMap, fragCoord.xy, framebufferLod);
              #else
                return texture2D(transmissionSamplerMap, fragCoord.xy, framebufferLod);
              #endif
            #else
              return texture2D(buffer, fragCoord.xy);
            #endif
          }
          vec3 applyVolumeAttenuation( const in vec3 radiance, const in float
transmissionDistance, const in vec3 attenuationColor, const in float attenuationDistance ) {
            if ( isinf( attenuationDistance ) ) {
              // Attenuation distance is +∞, i.e. the transmitted color is not attenuated at
all.
              return radiance;
            } else {
              // Compute light attenuation using Beer's law.
              vec3 attenuationCoefficient = -log( attenuationColor ) / attenuationDistance;
              vec3 transmittance = exp( - attenuationCoefficient * transmissionDistance ); //
Beer's law
              return transmittance * radiance;
            }
          }
          vec4 getIBLVolumeRefraction( const in vec3 n, const in vec3 v, const in float
roughness, const in vec3 diffuseColor,
            const in vec3 specularColor, const in float specularF90, const in vec3 position,
const in mat4 modelMatrix,
            const in mat4 viewMatrix, const in mat4 projMatrix, const in float ior, const in
float thickness,
            const in vec3 attenuationColor, const in float attenuationDistance ) {
            vec3 transmissionRay = getVolumeTransmissionRay( n, v, thickness, ior,
modelMatrix );
            vec3 refractedRayExit = position + transmissionRay;
            // Project refracted vector on the framebuffer, while mapping to normalized
device coordinates.
            vec4 ndcPos = projMatrix * viewMatrix * vec4( refractedRayExit, 1.0 );
            vec2 refractionCoords = ndcPos.xy / ndcPos.w;
            refractionCoords += 1.0;
            refractionCoords /= 2.0;
            // Sample framebuffer to get pixel the refracted ray hits.
            vec4 transmittedLight = getTransmissionSample( refractionCoords, roughness, ior
);
            vec3 attenuatedColor = applyVolumeAttenuation( transmittedLight.rgb, length(
transmissionRay ), attenuationColor, attenuationDistance );
            // Get the specular component.
            vec3 F = EnvironmentBRDF( n, v, specularColor, specularF90, roughness );
            return vec4( ( 1.0 - F ) * attenuatedColor * diffuseColor, transmittedLight.a );
          }
        #endif\n`
      )

      // Add refraction
      shader.fragmentShader = shader.fragmentShader.replace(
        '#include <transmission_fragment>',
        /*glsl*/ `
        // Improve the refraction to use the world pos
        material.transmission = _transmission;
        material.transmissionAlpha = 1.0;
        material.thickness = thickness;
        material.attenuationDistance = attenuationDistance;
        material.attenuationColor = attenuationColor;
        #ifdef USE_TRANSMISSIONMAP
          material.transmission *= texture2D( transmissionMap, vUv ).r;
        #endif
        #ifdef USE_THICKNESSMAP
          material.thickness *= texture2D( thicknessMap, vUv ).g;
        #endif

        vec3 pos = vWorldPosition;
        float runningSeed = 0.0;
        vec3 v = normalize( cameraPosition - pos );
        vec3 n = inverseTransformDirection( normal, viewMatrix );
        vec3 transmission = vec3(0.0);
        float transmissionR, transmissionB, transmissionG;
        float randomCoords = rand(runningSeed++);
        float thickness_smear = thickness * max(pow(roughnessFactor, 0.33), anisotropicBlur);
        vec3 distortionNormal = vec3(0.0);
        vec3 temporalOffset = vec3(time, -time, -time) * temporalDistortion;
        if (distortion > 0.0) {
          distortionNormal = distortion * vec3(snoiseFractal(vec3((pos * distortionScale +
temporalOffset))), snoiseFractal(vec3(pos.zxy * distortionScale - temporalOffset)),
snoiseFractal(vec3(pos.yxz * distortionScale + temporalOffset)));
        }
        for (float i = 0.0; i < ${samples}.0; i ++) {
          vec3 sampleNorm = normalize(n + roughnessFactor * roughnessFactor * 2.0 *
normalize(vec3(rand(runningSeed++) - 0.5, rand(runningSeed++) - 0.5, rand(runningSeed++) -
0.5)) * pow(rand(runningSeed++), 0.33) + distortionNormal);
          transmissionR = getIBLVolumeRefraction(
            sampleNorm, v, material.roughness, material.diffuseColor, material.specularColor,
material.specularF90,
            pos, modelMatrix, viewMatrix, projectionMatrix, material.ior, material.thickness
+ thickness_smear * (i + randomCoords) / float(${samples}),
            material.attenuationColor, material.attenuationDistance
          ).r;
          transmissionG = getIBLVolumeRefraction(
            sampleNorm, v, material.roughness, material.diffuseColor, material.specularColor,
material.specularF90,
            pos, modelMatrix, viewMatrix, projectionMatrix, material.ior  * (1.0 +
chromaticAberration * (i + randomCoords) / float(${samples})) , material.thickness +
thickness_smear * (i + randomCoords) / float(${samples}),
            material.attenuationColor, material.attenuationDistance
          ).g;
          transmissionB = getIBLVolumeRefraction(
            sampleNorm, v, material.roughness, material.diffuseColor, material.specularColor,
material.specularF90,
            pos, modelMatrix, viewMatrix, projectionMatrix, material.ior * (1.0 + 2.0 *
chromaticAberration * (i + randomCoords) / float(${samples})), material.thickness +
thickness_smear * (i + randomCoords) / float(${samples}),
            material.attenuationColor, material.attenuationDistance
          ).b;
          transmission.r += transmissionR;
          transmission.g += transmissionG;
          transmission.b += transmissionB;
        }
        transmission /= ${samples}.0;
        totalDiffuse = mix( totalDiffuse, transmission.rgb, material.transmission );\n`
      )
    }

    Object.keys(this.uniforms).forEach((name) =>
      Object.defineProperty(this, name, {
        get: () => this.uniforms[name].value,
        set: (v) => (this.uniforms[name].value = v),
      })
    )
  }
}

export const MeshTransmissionMaterial: ForwardRefComponent<
  MeshTransmissionMaterialProps,
  ThreeElements['meshTransmissionMaterial']
> = /* @__PURE__ */ React.forwardRef(
  (
    {
      buffer,
      transmissionSampler = false,
      backside = false,
      side = THREE.FrontSide,
      transmission = 1,
      thickness = 0,
      backsideThickness = 0,
      backsideEnvMapIntensity = 1,
      samples = 10,
      resolution,
      backsideResolution,
      background,
      anisotropy,
      anisotropicBlur,
      ...props
    }: MeshTransmissionMaterialProps,
    fref
  ) => {
    extend({ MeshTransmissionMaterial: MeshTransmissionMaterialImpl })

    const ref = React.useRef<ThreeElements['meshTransmissionMaterial']>(null!)
    const [discardMaterial] = React.useState(() => new DiscardMaterial())
    const fboBack = useFBO(backsideResolution || resolution)
    const fboMain = useFBO(resolution)

    let oldBg
    let oldEnvMapIntensity
    let oldTone
    let parent
    useFrame((state) => {
      ref.current.time = state.clock.elapsedTime
      // Render only if the buffer matches the built-in and no transmission sampler is set
      if (ref.current.buffer === fboMain.texture && !transmissionSampler) {
        parent = (ref.current as any).__r3f.parent?.object as THREE.Object3D | undefined
        if (parent) {
          // Save defaults
          oldTone = state.gl.toneMapping
          oldBg = state.scene.background
          oldEnvMapIntensity = ref.current.envMapIntensity

          // Switch off tonemapping lest it double tone maps
          // Save the current background and set the HDR as the new BG
          // Use discardmaterial, the parent will be invisible, but it's shadows will still
be cast
          state.gl.toneMapping = THREE.NoToneMapping
          if (background) state.scene.background = background
          parent.material = discardMaterial

          if (backside) {
            // Render into the backside buffer
            state.gl.setRenderTarget(fboBack)
            state.gl.render(state.scene, state.camera)
            // And now prepare the material for the main render using the backside buffer
            parent.material = ref.current
            parent.material.buffer = fboBack.texture
            parent.material.thickness = backsideThickness
            parent.material.side = THREE.BackSide
            parent.material.envMapIntensity = backsideEnvMapIntensity
          }

          // Render into the main buffer
          state.gl.setRenderTarget(fboMain)
          state.gl.render(state.scene, state.camera)

          parent.material = ref.current
          parent.material.thickness = thickness
          parent.material.side = side
          parent.material.buffer = fboMain.texture
          parent.material.envMapIntensity = oldEnvMapIntensity

          // Set old state back
          state.scene.background = oldBg
          state.gl.setRenderTarget(null)
          state.gl.toneMapping = oldTone
        }
      }
    })

    // Forward ref
    React.useImperativeHandle(fref, () => ref.current, [])

    return (
      <meshTransmissionMaterial
        // Samples must re-compile the shader so we memoize it
        args={[samples, transmissionSampler]}
        ref={ref as any}
        {...props}
        buffer={buffer || fboMain.texture}
        // @ts-ignore
        _transmission={transmission}
        // In order for this to not incur extra cost "transmission" must be set to 0 and
treated as a reserved prop.
        // This is because THREE.WebGLRenderer will check for transmission > 0 and execute
extra renders.
        // The exception is when transmissionSampler is set, in which case we are using
three's built in sampler.
        anisotropicBlur={anisotropicBlur ?? anisotropy}
        transmission={transmissionSampler ? transmission : 0}
        thickness={thickness}
        side={side}
      />
    )
  }
)
```