```
import { useFrame } from '@react-three/fiber'
import { useMemo, useRef, useEffect, useState } from 'react'
import * as THREE from 'three'
import { DracoPointCloudLoader } from '../utils/dracoLoader'

interface DracoPointCloudProps {
  url: string
  pointSize?: number
  color?: string
  position?: [number, number, number]
  rotation?: [number, number, number]
  scale?: [number, number, number]
  resolution?: number
}

export function DracoPointCloud({
  url,
  pointSize = 0.01,
  color = '#ffffff',
  position = [0, 0, 0],
  rotation = [0, 0, 0],
  scale = [1, 1, 1],
  resolution = 1.0
}: DracoPointCloudProps) {
  const [dracoData, setDracoData] = useState<THREE.BufferGeometry | null>(null)
  const pointsRef = useRef<THREE.Points>(null)
  const materialRef = useRef<THREE.PointsMaterial>(null)

  useEffect(() => {
    const loader = new DracoPointCloudLoader()

    loader.load(
      url,
      (geometry) => {
        setDracoData(geometry)
      },
      undefined,
      (error) => {
        console.error('Error loading Draco file:', error)
      }
    )

    return () => {
      loader.dispose()
    }
  }, [url])

  const { positions, colors, normals, sampledCount } = useMemo(() => {
    if (!dracoData) {
      return {
        positions: new Float32Array(0),
        colors: new Float32Array(0),
        normals: new Float32Array(0),
        sampledCount: 0
      }
    }
    const positionAttribute = dracoData.attributes.position
    const colorAttribute = dracoData.attributes.color
    const normalAttribute = dracoData.attributes.normal

    // Extract position data
    const originalPositions = new Float32Array(positionAttribute.array)

    // Apply resolution by sampling points
    let sampledPositions: Float32Array
    let currentSampledCount: number
    if (resolution !== 1.0) {
      const totalPoints = originalPositions.length / 3
      const sampleStep = Math.max(1, Math.floor(1 / resolution))
      currentSampledCount = Math.floor(totalPoints / sampleStep)
      sampledPositions = new Float32Array(currentSampledCount * 3)

      for (let i = 0; i < currentSampledCount; i++) {
```

```
        const sourceIndex = i * sampleStep
        sampledPositions[i * 3] = originalPositions[sourceIndex * 3]
        sampledPositions[i * 3 + 1] = originalPositions[sourceIndex * 3 + 1]
        sampledPositions[i * 3 + 2] = originalPositions[sourceIndex * 3 + 2]
      }
    } else {
      sampledPositions = originalPositions
      currentSampledCount = originalPositions.length / 3
    }

    // Extract color data if available
    let sampledColors = new Float32Array(0)
    if (colorAttribute) {
      const originalColors = new Float32Array(colorAttribute.array)

      // Apply same sampling to colors
      if (resolution !== 1.0) {
        sampledColors = new Float32Array(currentSampledCount * 3)
        const sampleStep = Math.max(1, Math.floor(1 / resolution))

        for (let i = 0; i < currentSampledCount; i++) {
          const sourceIndex = i * sampleStep
          sampledColors[i * 3] = originalColors[sourceIndex * 3]
          sampledColors[i * 3 + 1] = originalColors[sourceIndex * 3 + 1]
          sampledColors[i * 3 + 2] = originalColors[sourceIndex * 3 + 2]
        }
      } else {
        sampledColors = originalColors
      }
    }

    // Extract normal data if available
    let sampledNormals = new Float32Array(0)
    if (normalAttribute) {
      const originalNormals = new Float32Array(normalAttribute.array)

      // Apply same sampling to normals
      if (resolution !== 1.0) {
        sampledNormals = new Float32Array(currentSampledCount * 3)
        const sampleStep = Math.max(1, Math.floor(1 / resolution))

        for (let i = 0; i < currentSampledCount; i++) {
          const sourceIndex = i * sampleStep
          sampledNormals[i * 3] = originalNormals[sourceIndex * 3]
          sampledNormals[i * 3 + 1] = originalNormals[sourceIndex * 3 + 1]
          sampledNormals[i * 3 + 2] = originalNormals[sourceIndex * 3 + 2]
        }
      } else {
        sampledNormals = originalNormals
      }
    }

    return {
      positions: sampledPositions,
      colors: sampledColors,
      normals: sampledNormals,
      sampledCount: currentSampledCount
    }
  }, [dracoData, resolution])

  const points = useMemo(() => {
    if (sampledCount === 0) return null

    const geometry = new THREE.BufferGeometry()

    geometry.setAttribute('position', new THREE.BufferAttribute(positions, 3))

    if (colors.length > 0) {
      geometry.setAttribute('color', new THREE.BufferAttribute(colors, 3))
    }

    if (normals.length > 0) {
      geometry.setAttribute('normal', new THREE.BufferAttribute(normals, 3))
```

```
    }

    return geometry
  }, [positions, colors, normals, sampledCount])

  // Update material properties in real-time
  useFrame(() => {
    if (materialRef.current) {
      materialRef.current.size = pointSize
      materialRef.current.color.set(color)
    }
  })

  if (!points) return null

  const hasVertexColors = !!dracoData?.attributes.color

  return (
    <points
      ref={pointsRef}
      position={position}
      rotation={rotation}
      scale={scale}
    >
      <primitive object={points} />
      <pointsMaterial
        ref={materialRef}
        size={pointSize}
        vertexColors={hasVertexColors}
        color={hasVertexColors ? undefined : color}
        sizeAttenuation={true}
        transparent={false}
        alphaTest={0.5}
        depthWrite={true}
        depthTest={true}
        // Use circular points instead of squares
        map={null}
        // Make material unlit to preserve original colors
        onBeforeCompile={(shader) => {
          if (hasVertexColors) {
            // Replace the entire fragment shader to ignore lighting and use circula
r points
            shader.fragmentShader = `
              varying vec3 vColor;
              varying vec2 vUv;
              void main() {
                vec2 center = vec2(0.5, 0.5);
                float dist = distance(gl_PointCoord, center);
                if (dist > 0.5) discard;
                gl_FragColor = vec4(vColor, 1.0);
              }
            `
          } else {
            // For non-vertex colored points, still use circular shape
            shader.fragmentShader = shader.fragmentShader.replace(
              'gl_FragColor = vec4( outgoingLight, diffuseColor.a );',
              `
              vec2 center = vec2(0.5, 0.5);
              float dist = distance(gl_PointCoord, center);
              if (dist > 0.5) discard;
              gl_FragColor = vec4( outgoingLight, diffuseColor.a );
              `
            )
          }
        }}
      />
    </points>
  )
}
```