*//package renderer;  
//  
//import elements.Camera;  
//import primitives.Color;  
//import primitives.Point3D;  
//import primitives.Ray;  
//import scene.Scene;  
//  
//import java.util.List;  
//import java.util.MissingResourceException;  
//  
///\*\*  
// \* this class create the image's color matrix from the scene  
// \*/  
//public class Render {  
// private ImageWriter \_imageWriter;  
// private Camera \_camera;  
// private RayTracerBase \_rayTracer;  
//  
// //chaining methods  
// public Render setImageWriter(ImageWriter imageWriter) {  
// \_imageWriter = imageWriter;  
// return this;  
// }  
//  
//  
// public Render setCamera(Camera camera) {  
// \_camera = camera;  
// return this;  
// }  
//  
// public Render setRayTracer(RayTracerBase rayTracer) {  
// \_rayTracer = rayTracer;  
// return this;  
// }  
//  
// /\*\*  
// \* this function calculate the color of each pixel, and color it  
// \*/  
// public void renderImage() {  
// //check that all the field is not null  
// if (\_imageWriter == null || \_camera == null || \_rayTracer == null) {  
// throw new MissingResourceException("one or more field in render is null", "render", "");  
// }  
// //calculate and paint all the pixels  
// int nX = \_imageWriter.getNx();  
// int nY = \_imageWriter.getNy();  
// for (int i = 0; i < nY; i++) {  
// for (int j = 0; j < nX; j++) {  
// if(i==100&&j==100) {  
// System.out.println(i + "-" + j);  
// }  
// System.out.println(i + "-" + j);  
// Ray ray = \_camera.constructRayThroughPixel(nX, nY, j, i);  
// Color color = \_rayTracer.traceRay(ray);  
// \_imageWriter.writePixel(j, i, color);  
// }  
// }  
// }  
//  
// /\*\*  
// \* print grid on the picture  
// \*  
// \* @param interval distance between the lines grid  
// \* @param intervalColor the color of the grid  
// \*/  
// public void printGrid(int interval, Color intervalColor) {  
// if (\_imageWriter == null) {  
// throw new MissingResourceException("\_imageWriter is null", "render", "");  
// }  
// int nX = \_imageWriter.getNx();  
// int nY = \_imageWriter.getNy();  
// for (int i = 0; i < nY; i++) {  
// for (int j = 0; j < nX; j++) {  
// if (i % interval == 0 || j % interval == 0) {  
// \_imageWriter.writePixel(j, i, intervalColor);  
// }  
// }  
// }  
//  
// }  
//  
// /\*\*  
// \* this methode implement the low of Demeter  
// \*/  
// public void writeToImage() {  
// if (\_imageWriter == null) {  
// throw new MissingResourceException("\_imageWriter is null", "render", "");  
// }  
// \_imageWriter.writeToImage();  
// }  
//}*

*/\*\*  
 \* this function check if there is intersection between the box and the ray  
 \*  
 \** ***@param ray*** *\** ***@return*** *\*/* **public boolean** hit1(Ray ray) {  
 Point3D p0 = ray.getP0();  
 **double** p0X = p0.getX();  
 **double** p0Y = p0.getY();  
 **double** p0Z = p0.getZ();  
 Vector direction = ray.getDir();  
 **double** directionX = direction.getX();  
 **double** directionY = direction.getY();  
 **double** directionZ = direction.getZ();  
  
 **double** maxX;  
 **double** minX;  
*//if the directionX is negative, then the min x in the box is maximal* **if** (directionX < 0) {  
 maxX = (**\_min**.getX() - p0X) / directionX;  
 *//check if the geometry is behind the camera* **if** (maxX <= 0)  
 **return false**;  
 minX = (**\_max**.getX() - p0X) / directionX;  
 } **else if** (directionX > 0) {*//* maxX = (**\_max**.getX() - p0X) / directionX;  
 **if** (maxX <= 0)  
 **return false**;  
 minX = (**\_min**.getX() - p0X) / directionX;  
 } **else** {  
 **if** (p0X >= **\_max**.getX() || p0X <= **\_min**.getX())  
 **return false**;  
 **else** {  
 maxX = Double.***POSITIVE\_INFINITY***;  
 minX = Double.***NEGATIVE\_INFINITY***;  
 }  
 }  
  
 **double** maxY;  
 **double** minY;  
  
*//if the directionY is negative, then the min Y in the box is maximal* **if** (directionY < 0) {  
 maxY = (**\_min**.getY() - p0Y) / directionY;  
 *//check if the geometry is behind the camera* **if** (maxY <= 0)  
 **return false**;  
 minY = (**\_max**.getY() - p0Y) / directionY;  
 } **else if** (directionY > 0) {  
 maxY = (**\_max**.getY() - p0Y) / directionY;  
 **if** (maxX <= 0)  
 **return false**;  
 minY = (**\_min**.getY() - p0Y) / directionY;  
 } **else** {  
 **if** (p0Y >= **\_max**.getY() || p0Y <= **\_min**.getY())  
 **return false**;  
 **else** {  
 maxY = Double.***POSITIVE\_INFINITY***;  
 minY = Double.***NEGATIVE\_INFINITY***;  
 }  
 }  
*//  
 // | |  
 // miss \*t0x |  
 // \* | |  
 //--\*t1y--|---------|-------------  
 // | |  
 // | |  
 //--------|---------|---\*t0y----------  
 // | | \* miss  
 // | \*t1x  
 // | |* **double** tempMax = maxY < maxX ? maxY : maxX;  
 **double** tempMin = minY > minX ? minY : minX;  
 tempMin = tempMin > 0 ? tempMin : 0;  
 **if** (tempMax < tempMin)  
 **return false**;  
  
 **double** maxZ;  
 **double** minZ;  
  
*//if the directionZ is negative, then the min Z in the box is maximal* **if** (directionZ < 0) {  
 maxZ = (**\_min**.getZ() - p0Z) / directionZ;  
 *//check if the geometry is behind the camera* **if** (maxZ <= 0)  
 **return false**;  
 minZ = (**\_max**.getZ() - p0Z) / directionZ;  
 } **else if** (directionZ > 0) {  
 maxZ = (**\_max**.getZ() - p0Z) / directionZ;  
 **if** (maxX <= 0)  
 **return false**;  
 minZ = (**\_min**.getZ() - p0Z) / directionZ;  
 } **else** {  
 **if** (p0Z >= **\_max**.getZ() || p0Z <= **\_min**.getZ())  
 **return false**;  
 **else** {  
 maxZ = Double.***POSITIVE\_INFINITY***;  
 minZ = Double.***NEGATIVE\_INFINITY***;  
 }  
 }  
  
 tempMax = maxZ < tempMax ? maxZ : tempMax;  
 tempMin = minZ > tempMin ? minZ : tempMin;  
 **if** (tempMax < tempMin)  
 **return false**;  
  
 **return true**;  
  
 }

*/\*\*  
 \* create BVH tree from the geometries  
 \*/***public void** createBVHTree() {  
 *//remove all the infinity geometries from \_listOfGeometries, and add them in the end* List<Intersectable> planeList = **new** LinkedList<>();  
 **for** (Intersectable geo : **\_listOfGeometries**) {  
 **if** (geo **instanceof** Plane) {  
 planeList.add(geo);  
 }  
 }  
 **\_listOfGeometries**.removeAll(planeList);  
  
 *//seek every time the closest boxes, and union them into a big box  
 //repeat until there is one box left* **double** distance = 0;  
 Intersectable son1 = **null**;  
 Intersectable son2 = **null**;  
 **while** (**\_listOfGeometries**.size() > 1) {*//* **double** minDistance = Double.***POSITIVE\_INFINITY***;  
 **for** (Intersectable geo1 : **\_listOfGeometries**) {  
 **for** (Intersectable geo2 : **\_listOfGeometries**) {  
 *//find the minimum distance between 2 boxes* **if** (geo1 != geo2 && (distance = distance(geo1, geo2)) < minDistance) {  
 minDistance = distance;  
 son1 = geo1;  
 son2 = geo2;  
 }  
 }  
 }  
 Geometries tempGeometries = **new** Geometries(son1, son2);*//union the two closest geometries* **\_listOfGeometries**.remove(son1);  
 **\_listOfGeometries**.remove(son2);  
 **\_listOfGeometries**.add(tempGeometries);  
  
 }  
  
 **\_listOfGeometries**.addAll(planeList);*//add the infinity geometries to the list*}

*/\*\*  
 \* create BVH tree from the geometries  
 \*/***public void** createBVHTree1() {  
 List<Intersectable> planeList = **new** LinkedList<>();  
 **for** (Intersectable geo : **\_listOfGeometries**) {  
 **if** (geo **instanceof** Plane) {  
 planeList.add(geo);  
 }  
 }  
 **\_listOfGeometries**.removeAll(planeList);  
 **double** distance = 0;  
 Intersectable left = **null**;  
 Intersectable right = **null**;  
 **while** (**\_listOfGeometries**.size() > 1) {  
 **double** minDistance = Double.***POSITIVE\_INFINITY***;  
 Intersectable geo1 = **\_listOfGeometries**.get(0);  
 **for** (Intersectable geo2 : **\_listOfGeometries**) {  
 **if** (geo1 != geo2 && (distance = distance(geo1, geo2)) < minDistance) {  
 minDistance = distance;  
 left = geo1;  
 right = geo2;  
 }  
 }  
  
 Geometries tempGeometries = **new** Geometries(left, right);  
 **\_listOfGeometries**.remove(left);  
 **\_listOfGeometries**.remove(right);  
 **\_listOfGeometries**.add(tempGeometries);  
  
 }  
 **\_listOfGeometries**.addAll(planeList);  
}