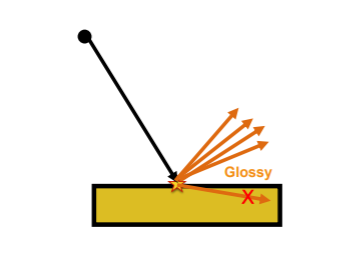
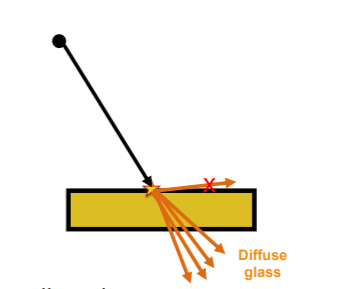
השיפורים שעשינו:

* Glossy surface
  + הבעיה - שזכוכית משקפת לחלוטין
  + הפתרון – במקום לשלוח קרן אחד של השתקפות נשלח יותר
  + תוצאה – מה שיותר קרוב יראה חד ומה שרחוק יותר יראה מטושטש בגלל שהתפזרות הקרניים מתרחבת עם המרחק



* Diffuse glass
* הבעיה – משטחים שמבריקים באופן מושלם
* הפתרון – במקום לשלוח קרן אחד של שקיפות נשלח יותר
  + תוצאה - מה שיותר קרוב יראה חד ומה שרחוק יותר יראה מטושטש בגלל שהתפזרות הקרניים מתרחבת עם המרחק



* Bounding volume hierarchy – לוקח מלא זמן למצוא נקודות חיתוך עם גיאומטריות – במיוחד עם תמונות מסובכות לכן נכין לכל הגיאומטריות קופסה שנכניס לרשימה של גיאומטריות ובמקום לבדוק כל פעם חיתוך עם גיאומטריה נבדוק קודם עם הקופסה וכך יחסוך זמן

איך ממשנו?

* קודם הוספנו פונקציה במחלקת Util שמחשבת מספר רנדומלי בתחום [min,max)

*/\*\*  
 \* generates a random number to help generate rays for ray tracing  
 \** ***@param min*** *the minimum number  
 \** ***@param max*** *the maximum number  
 \** ***@return*** *the random number that we generated  
 \*/***public static double** randomNumber(**double** min, **double** max)  
{  
 **double** random=Math.*random*() \*(max-min) +min;  
 **return** random;  
}

* הוספנו פונקציה למחלקת Vector שמחשבת את הנורמל לווקטור שקורא לפונקציה (בשיפורים שלנו זה הכיוון של הקרן הראשית)

*/\*\*  
 \* creates a vector normal to the vector that calls the function (the dot product of the new vector and the old vector equals zero)  
 \** ***@return*** *normal vector  
 \*/***public** Vector normalToVector()  
{  
 **int** min=0;  
 **double** coordinate;  
 *//finding the smallest coordinate of the vector to replace it with 0* **if**(**this**.get\_head().get\_x().get()>0)  
 {  
 coordinate = **this**.get\_head().get\_x().get();  
 }  
 **else** coordinate=-**this**.get\_head().get\_x().get();  
 **if**(Math.*abs*(**this**.get\_head().get\_y().get())<coordinate)  
 {  
 coordinate=1;  
 **if**(**this**.get\_head().get\_y().get()>0)  
 coordinate=**this**.get\_head().get\_y().get();  
 **else** coordinate=-**this**.get\_head().get\_y().get();  
 }  
 **if**(Math.*abs*(**this**.get\_head().get\_z().get())<coordinate)  
 {  
 coordinate=2;  
 *//last coordinate that we are checking so no need to reassign coordinate* }  
 **if**(coordinate==0)*//x is the smallest* **return new** Vector(0,-**this**.get\_head().get\_z().get(),**this**.get\_head().get\_y().get()).normalize();  
 **if**(coordinate==1)*//y is the smallest* **return new** Vector(-**this**.get\_head().get\_z().get(),0,**this**.get\_head().get\_x().get()).normalize();  
 *//z is the smallest* **return new** Vector(**this**.get\_head().get\_y().get(),-**this**.get\_head().get\_x().get(),0).normalize();  
}

* הוספנו למחלקת Ray פונקציה שמייצרת אלומת קרניים לפי הקרן הראשית (this) , הווקטור נורמל בין הגיאומטריה וה geopoint, והמרחק שאנחנו מחליטות – מיצרים מעגל וירטואלי שאנך לקרן שקורא לפונקציה. הרדיוס של המעגל זה ה tan של הזווית בין שתי הווקטורים הניצבים שמצאנו , מפזרים נקודות רנדומליות בתוך המעגל ולפי זה מוצאים את הקרניים החדשות.

*/\*\*  
 \* creates a beam of rays(in a list of rays)  
 \*  
 \** ***@param n*** *Vector - normal vector where the rays start  
 \** ***@param distance*** *double - the distance between the point and the circle we are creating to find the beam  
 \** ***@param num*** *int - the number of rays that will be in the beam  
 \** ***@return*** *list that includes all the rays that make up the beam  
 \*/***public** List<Ray> createBeamOfRays(Vector n, **double** distance, **int** num) {  
 List<Ray> beam = **new** LinkedList<Ray>();  
 beam.add(**this**);*//the original ray that calls the function - there has to be at least one beam* **if** (num == 1)*//if no additional rays were requested here there is nothing else to do in this function* **return** beam;  
 Vector w = **this**.get\_dir().normalToVector();*//finds a vector that is normal to the direction on the ray* Vector v = **this**.get\_dir().crossProduct(w).normalize();*//the cross product between the normal and the direction* Point3D center = **this**.getTargetPoint(distance);*//the center of our circle is the distance requested from p0* Point3D randomP = Point3D.***ZERO***;  
 **double** xRandom, yRandom, random;  
 **double** nDotDirection = *alignZero*(n.dotProduct(**this**.get\_dir()));  
 **double** r = Math.*abs*(Math.*tan*(Math.*acos*(w.dotProduct(v))));  
 **for** (**int** i = 1; i < num; i++)*//starts from 1 because there has to be at least one ray(the original)and we already dealt with it* {  
 xRandom = *randomNumber*(-1, 1);*//random number [-1,1)* yRandom = Math.*sqrt*(1 - Math.*pow*(xRandom, 2));  
 random = *randomNumber*(-r, r);*//random number[-r,r)* **if** (xRandom != 0)*//vector cannot be scaled with zero* randomP = center.add(w.scale(random));  
 **if** (yRandom != 0)*//vector cannot be scaled with zero* randomP = center.add(v.scale(random));  
 Vector t = randomP.subtract(**this**.get\_p0());*//vector between the random point and the start of the original ray* **double** normalDotT = *alignZero*(n.dotProduct(t));  
 **if** (nDotDirection \* normalDotT > 0)*//if they are both positive or both negative then we need to create a ray with the original p0 and t* beam.add(**new** Ray(**this**.get\_p0(), t));  
 }  
 **return** beam;  
}

* הוספנו למחלקת Render את השדות הבאים ו get and set בהתאם – שניהם בהתחלה עם ערך ברירת מחדל אפס כדי שלא נצטרך לחזור ולתקן את כל הטסטים הקודמים

*/\*\*  
 \* parameters for ray tracing- glossy surface and diffuse glass - they are in class render because this class takes care of ray tracing  
 \*/***private int \_numOfRays**;  
  
**private double \_rayDistance**;

*/\*\*  
 \* gets the distance we want between the ray point and the circle  
 \*  
 \** ***@return*** *double - distance  
 \*/***public double** get\_rayDistance() {  
 **return \_rayDistance**;  
}  
  
*/\*\*  
 \* sets the distance between the ray point and the circle  
 \*  
 \** ***@param \_rayDistance*** *\*/***public void** set\_rayDistance(**double** \_rayDistance) {  
 **if** (\_rayDistance < 0)  
 **throw new** IllegalArgumentException(**"distance cant be negative"**);  
 **this**.**\_rayDistance** = \_rayDistance;  
}  
  
*/\*\*  
 \* get number of rays function  
 \*  
 \** ***@return*** *number of rays that will be part of the beam  
 \*/***public int** get\_numOfRays() {  
 **return \_numOfRays**;  
}  
  
*/\*\*  
 \* sets the number of rays that will be part of the beam  
 \*  
 \** ***@param \_numOfRays*** *int - amount of rays that will be part of the beam  
 \*/***public void** set\_numOfRays(**int** \_numOfRays) {  
 **if** (\_numOfRays < 1)  
 **throw new** IllegalArgumentException(**"there has to be at least one ray"**);  
 **this**.**\_numOfRays** = \_numOfRays;  
}

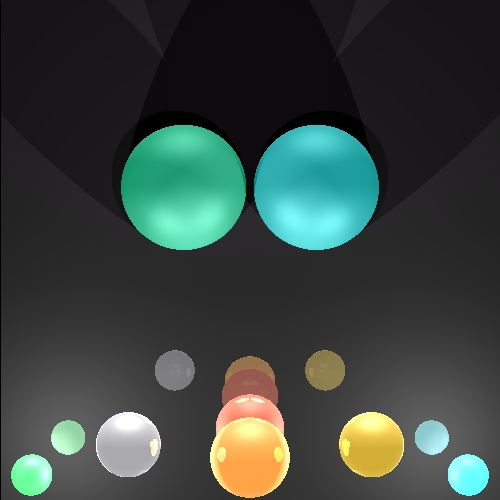
* תיקנו את calcColor הרקורסיבית שיתמוך בשיפורים שלנו כי שמה אנחנו מחשבים את השקיפות והשתקפות – הוספנו את היצירה של האלומת קרניים על שקיפות והשתקפות ובצענו קריאה רקורסיבית עבור כל קרן ששייך לאלומה והכנסנו את הצבע למשתנה זמני ואחרי שסיימנו את הקריאות הרקורסיביות חלקנו במספר הקרניים (תמיד יש לפחות אחד) ואז הוספנו לצבע.

*/\*\*  
 \* Calculate the color intensity in a point  
 \** ***@param gp*** *the point for which the color is required  
 \** ***@param in*** *Ray  
 \** ***@param level*** *- the recursion level  
 \** ***@param k*** *- double - helps with recursion  
 \** ***@return*** *the color intensity  
 \*/***private** primitives.Color calcColor(Intersectable.GeoPoint gp,Ray in,**int** level,**double** k) {  
 **if** (level == 1 || k < ***MIN\_CALC\_COLOR\_K***)  
 **return** primitives.Color.***BLACK***;  
 List<Ray>beam=**new** LinkedList<>();  
 primitives.Color color = gp.getGeometry().get\_emission();*//the geometries emission light* Vector v = gp.**point**.subtract(**\_scene**.getCamera().get\_p0()).normalize();*//subtracts the camera starting point from the geopoint and normalizes the vector* **double** kr = k \* gp.getGeometry().get\_material().get\_kR();*//reflection* **double** kt = k \* gp.getGeometry().get\_material().get\_kT();*//refraction* **double** transparencyAmount=0;*//transparency* **for** (LightSource : **\_scene**.getLightSources())*//for each light source in the scene's light sources* {  
 Vector l = lightSource.getL(gp.**point**);*//the lights direction from geopoint* **if** (*alignZero*( gp.**geometry**.getNormal(gp.getPoint()).dotProduct(l)) \* *alignZero*( gp.**geometry**.getNormal(gp.getPoint()).dotProduct(v)) > 0)*//if the dot product between the normal and the light direction times the dot product btween the normal and the normal vector between the camera and geopoint* {  
 *// if (unshaded(lightSource, l, n, gp))//if the geopoint isnt shaded by the light* transparencyAmount = transparency(lightSource, l, gp.**geometry**.getNormal(gp.getPoint()), gp);  
 **if** (transparencyAmount \* k > ***MIN\_CALC\_COLOR\_K***) {  
 Material =gp.**geometry**.get\_material();  
 **double** ln =l.dotProduct( gp.**geometry**.getNormal(gp.getPoint()));  
 color = color.add(calcDiffusive(  
 material.getKd(),  
 ln,  
 lightSource.getIntensity(gp.getPoint())),  
 calcSpecular(  
 material.getKs(), l,  
 gp.**geometry**.getNormal(gp.getPoint()),  
 ln,  
 v,  
 material.getnShininess(),  
 lightSource.getIntensity(gp.getPoint())));  
 }  
 }  
 }  
  
 **if** (kr > ***MIN\_CALC\_COLOR\_K***)*//if the reflection is bigger than the minimum of calc color* {  
 Ray reflection= constructReflectedRay(gp.getGeometry().getNormal(gp.getPoint()), gp.getPoint(), in);  
 **if**(**this**.**\_numOfRays**==0 ||**this**.**\_rayDistance**<0)  
 beam.add(reflection);  
 **else** beam= reflection.createBeamOfRays(gp.getGeometry().getNormal(gp.getPoint()),**this**.get\_rayDistance(),**this**.get\_numOfRays());  
 primitives.Color tempColorReflection = primitives.Color.***BLACK***;  
 **for**(Ray r :beam)  
 {  
 Intersectable.GeoPoint reflectedGp = findClosestIntersection(r);*//find the closest point to the reflection ray's p0* **if** (reflectedGp != **null**)*//if such a point exists* {  
 tempColorReflection = tempColorReflection.add(calcColor(reflectedGp, r, level - 1, kr).scale(kr));*//calls the recursion th find the rest of the color and then scales it with the reflection* }  
 }  
 color = color.add(tempColorReflection.reduce(beam.size()));  
  
  
 }  
 **if** (kt > ***MIN\_CALC\_COLOR\_K***)*//if the refraction is bigger than the minimum of calc color* {  
  
  
  
 Ray refraction = constructRefractedRay(gp.getPoint(), in, gp.getGeometry().getNormal(gp.getPoint()));*//constructs a refracted ray* **if**(**this**.**\_numOfRays**==0 ||**this**.**\_rayDistance**<0)  
 beam.add(refraction);  
 **else** beam= refraction.createBeamOfRays(gp.getGeometry().getNormal(gp.getPoint()),**this**.get\_rayDistance(),**this**.get\_numOfRays());  
 primitives.Color tempColorRefraction = primitives.Color.***BLACK***;  
 **for**(Ray r :beam) {  
 Intersectable.GeoPoint refractedGp = findClosestIntersection(r);*//find the closest point to the refracted ray's p0* **if** (refractedGp != **null**)*//if such a point exists* {  
 tempColorRefraction = tempColorRefraction.add(calcColor(refractedGp, r, level - 1, kt).scale(kt));*//calls the recursion to find the rest of the color and then scales it with the refracted* }  
 }  
 color = color.add(tempColorRefraction.reduce(beam.size()));  
 }  
  
 **return** color;  
  
  
}

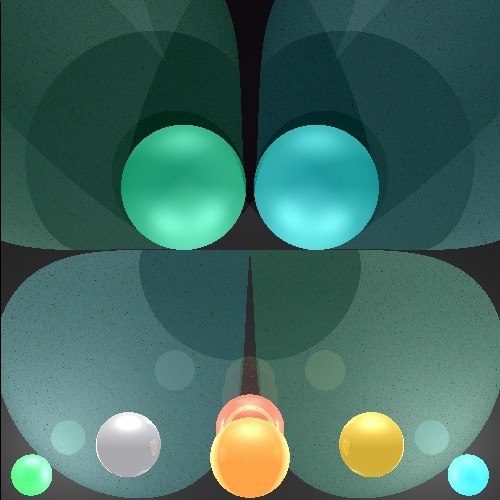
* הטסט שכתבנו להראות את השיפורים כל פעם שרצינו להריץ על מספר שונה של קרניים שיננו את זה דרך ה set ושיננו את השם של התמונה כדי שזה לא ידרוס תמונה אחרת

@Test  
**public void** rayTracingTest2()  
{  
 Scene = **new** Scene(**"Test scene"**);  
 scene.set\_camera(**new** Camera(**new** Point3D(0, 0, -1000), **new** Vector(0, 0, 1), **new** Vector(0, -1, 0)));  
 scene.set\_distance(600);  
 scene.set\_background(**new** Color (0,0,0));  
 scene.set\_ambientLight(**new** AmbientLight(**new** Color(255, 191, 191), 1));  
  
 scene.addGeometries(  
 **new** Plane(Color.***BLACK***,**new** Material(0.2,0.2,5,0,0.4),**new** Point3D(0,0,210),  
 **new** Vector(0,0,-1)),  
 **new** Sphere(**new** Color(212,175,55),*//gold* **new** Material(0.3, 0.4, 5, 0, 0.4),20,  
 **new** Point3D(75, 120, -75)),  
 **new** Sphere(**new** Color(53,187,202) ,*//blue* **new** Material(0.25, 0.3, 5, 0.22, 0),15,  
 **new** Point3D(160, 165, 100)),  
 **new** Sphere(**new** Color(255,99,71),*//red* **new** Material(0.25, 0.3, 5, 0,0.4),25,  
 **new** Point3D(0, 130, 90)),  
 **new** Sphere(**new** Color(255,164,71),*//orange* **new** Material(0.25, 0.3, 5, 0, 0.4),25,  
 **new** Point3D(0, 130, -60)),  
 **new** Sphere(**new** Color(0,128,85), *// Biggest Blue* **new** Material(0.3, 0.2, 5, 0,0),50,  
 **new** Point3D(-53, -50, 200)),  
 **new** Sphere(**new** Color (0,128,128),*//biggest teal* **new** Material(0.3, 0.2, 5, 0,0),50,  
 **new** Point3D(53, -50, 200)),  
 **new** Sphere(**new** Color(170,169,173),*//silver* **new** Material(0.3, 0.4, 5, 0, 0.4),20,  
 **new** Point3D(-75, 120, -75)),  
 **new** Sphere(**new** Color (53,202,93),*//green* **new** Material(0.3, 0.4, 5, 0.22, 0),15,  
 **new** Point3D(-160, 165, 100))  
  
  
 );  
  
 scene.addLights(  
 **new** DirectionalLight(**new** Color(210,210,210  
 ),**new** Vector(0,1,0)),  
 **new** SpotLight(**new** Color(130, 100, 130),**new** Point3D(0, 30, -50),  
 **new** Vector(0,-1,0),1, 4E-5, 2E-7),  
 **new** PointLight(**new** Color(210,210,210),**new** Point3D(-160,165,100))  
 ,**new** PointLight(**new** Color(210,210,210),**new** Point3D(160, 165, 100))  
 );  
  
 ImageWriter = **new** ImageWriter(**"ray tracing 2 - 50"**, 200, 200, 500, 500);  
 Render = **new** Render(imageWriter, scene);  
 render.set\_numOfRays(50);  
  
 render.set\_rayDistance(1);  
 render.set\_threads(3).setPrint();  
 render.renderImage();  
 render.writeToImage();  
}

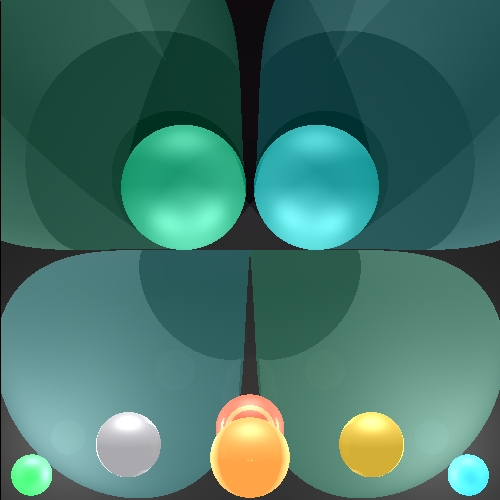
* תוצאות
  + בלי שיפור (קרן 1)



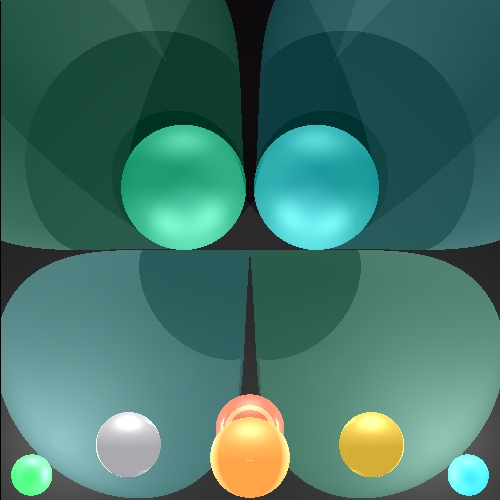
* + 10 קרניים



* + 50 קרניים



* + 100 קרניים



* BVH- שינוי של הממשק intersectable למחלקה abstract והוספנו מחלקה פנימית של Bounding Box
* **public class** BoundingBox {  
   **public double x1** = Double.***NEGATIVE\_INFINITY***;  
   **public double x2** = Double.***POSITIVE\_INFINITY***;  
   **public double y1** = Double.***NEGATIVE\_INFINITY***;  
   **public double y2** = Double.***POSITIVE\_INFINITY***;  
   **public double z1** = Double.***NEGATIVE\_INFINITY***;  
   **public double z2** = Double.***POSITIVE\_INFINITY***;  
    
  }

כאשר 1 מציין שזה מינימום ו 2 מקסימום

הוספנו פונקציה בוליאנית שבודקת אם קרן חותכת קופסה

**public boolean** intersects(Ray r) {  
 **double** xP=r.get\_p0().get\_x().get();  
 **double** xD=r.get\_dir().get\_head().get\_x().get();  
 **double** yP=r.get\_p0().get\_y().get();  
 **double** yD=r.get\_dir().get\_head().get\_y().get();  
 **double** zP=r.get\_p0().get\_z().get();  
 **double** zD=r.get\_dir().get\_head().get\_z().get();  
 **double** tmin = (**box**.**x1** - xP / xD);  
 **double** tmax = (**box**.**x2** - xP /xD);  
 **double** temp;  
 **if** (tmin > tmax) {  
 temp = tmin;  
 tmin = tmax;  
 tmax = temp;  
 }  
  
 **double** tymin = (**box**.**y1** - yP / yD);  
 **double** tymax = (**box**.**y2** - yP/ yD);  
  
 **if** (tymin > tymax) {  
 temp = tymin;  
 tymin = tymax;  
 tymax = temp;  
 }  
 **if** ((tmin > tymax) || (tymin > tmax))  
 **return false**;  
  
 **if** (tymin > tmin)  
 tmin = tymin;  
  
 **if** (tymax < tmax)  
 tmax = tymax;  
  
 **double** tzmin = (**box**.**z1** - zP / zD);  
 **double** tzmax = (**box**.**z2** - zP / zD);  
  
 **if** (tzmin > tzmax) {  
 temp = tzmin;  
 tzmin = tzmax;  
 tzmax = temp;  
 }  
  
 **if** ((tmin > tzmax) || (tzmin > tmax))  
 **return false**;  
  
  
 **return true**;  
  
}

הוספנו גם קריאה לבנאי ברירת מחדל בשיבל שכל צורה יקבל קופסה

**protected** BoundingBox **box** = **new** BoundingBox();

שיננו את ההוספה של גיאומטריות כך שזה גם יחשב את הקופסאות על יותר מצורה אחת

**public void** add(Intersectable... geometries)  
{  
  
 **for** (Intersectable geo : geometries )*// adds each of the geometries the function recieved to the list of intersectable geometries* {  
 **\_geometries**.add(geo);  
 **if**(**this**.**setBoxes**==**true**)  
 {  
 **if** (geo.**box**.**x2** > **this**.**box**.**x2**)  
 **this**.**box**.**x2** = geo.**box**.**x2**;  
 **if** (geo.**box**.**x1** < **this**.**box**.**x1**)  
 **this**.**box**.**x1** = geo.**box**.**x1**;  
 **if** (geo.**box**.**y2** > **this**.**box**.**y2**)  
 **this**.**box**.**y2** = geo.**box**.**y2**;  
 **if** (geo.**box**.**y1** < **this**.**box**.**y1**)  
 **this**.**box**.**y1** = geo.**box**.**y1**;  
 **if** (geo.**box**.**z2** > **this**.**box**.**z2**)  
 **this**.**box**.**z2** = geo.**box**.**z2**;  
 **if** (geo.**box**.**z1** < **this**.**box**.**z1**)  
 **this**.**box**.**z1** = geo.**box**.**z1**;  
  
 }  
}

הוספנו לבנאי של כדור

**if**(**this**.**setBoxes** ==**true**) {  
 **this**.**box**.**x1** = center.get\_x().get() - radius;  
 **this**.**box**.**x2** = center.get\_x().get() + radius;  
 **this**.**box**.**y1** = center.get\_y().get() - radius;  
 **this**.**box**.**y2** = center.get\_y().get() + radius;  
 **this**.**box**.**z1** = center.get\_z().get() - radius;  
 **this**.**box**.**z2** = center.get\_z().get() + radius;  
}

ולבנאי של polygon

**if**(**this**.**setBoxes**==**true**) {  
 **box**.**x1** = **this**.**\_vertices**.get(0).get\_x().get();  
 **box**.**x2** = **this**.**\_vertices**.get(0).get\_x().get();  
 **box**.**y1** = **this**.**\_vertices**.get(0).get\_y().get();  
 **box**.**y2** = **this**.**\_vertices**.get(0).get\_y().get();  
 **box**.**z1** = **this**.**\_vertices**.get(0).get\_z().get();  
 **box**.**z2** = **this**.**\_vertices**.get(0).get\_z().get();  
  
 **for** (**int** i = 1; i < **this**.**\_vertices**.size(); i++) {  
 **if** (**this**.**\_vertices**.get(i).get\_x().get() < **box**.**x1**)  
 **box**.**x1** = **this**.**\_vertices**.get(i).get\_x().get();  
 **if** (**this**.**\_vertices**.get(i).get\_y().get() < **box**.**y1**)  
 **box**.**y1** = **this**.**\_vertices**.get(i).get\_y().get();  
 **if** (**this**.**\_vertices**.get(i).get\_z().get() < **box**.**z1**)  
 **box**.**z1** = **this**.**\_vertices**.get(i).get\_z().get();  
 **if** (**this**.**\_vertices**.get(i).get\_x().get() > **box**.**x2**)  
 **box**.**x2** = **this**.**\_vertices**.get(i).get\_x().get();  
 **if** (**this**.**\_vertices**.get(i).get\_y().get() > **box**.**y2**)  
 **box**.**y2** = **this**.**\_vertices**.get(i).get\_y().get();  
 **if** (**this**.**\_vertices**.get(i).get\_z().get() > **box**.**z2**)  
 **box**.**z2** = **this**.**\_vertices**.get(i).get\_z().get();  
 }  
}

הוספנו את הפונקציה הבאה בשביל לעבור על העץ שלנו למצוא את הנקודות חיתוך

**public** List<GeoPoint> bvhTree(Ray ray)  
{  
 **if**(**this**.**\_geoList**.size()==0)  
 {  
 **return this**.findIntersections(ray);  
 }  
 **else** {  
 **if**(**this**.intersects(ray))  
 {  
 **for**(Geometries geo : **this**.**\_geoList**)  
 {  
 geo.bvhTree(ray);  
 }  
 }  
}  
 **return null**;  
}

גם הוספנו פעולת set מתאים בשביל להדליק את השיפור מהטסט

* תוצאה : טסטים מורכבים יותר אמורים לרוץ יותר מהר כי יש פחות בדיקות של נקודות חיתוך בין גיאומטריות לקרניים. בטסטים עם פחות צורות לא נראה שיפור או אפילו יקח יותר זמן כי לוקח זמן בשביל לבנות את העץ.