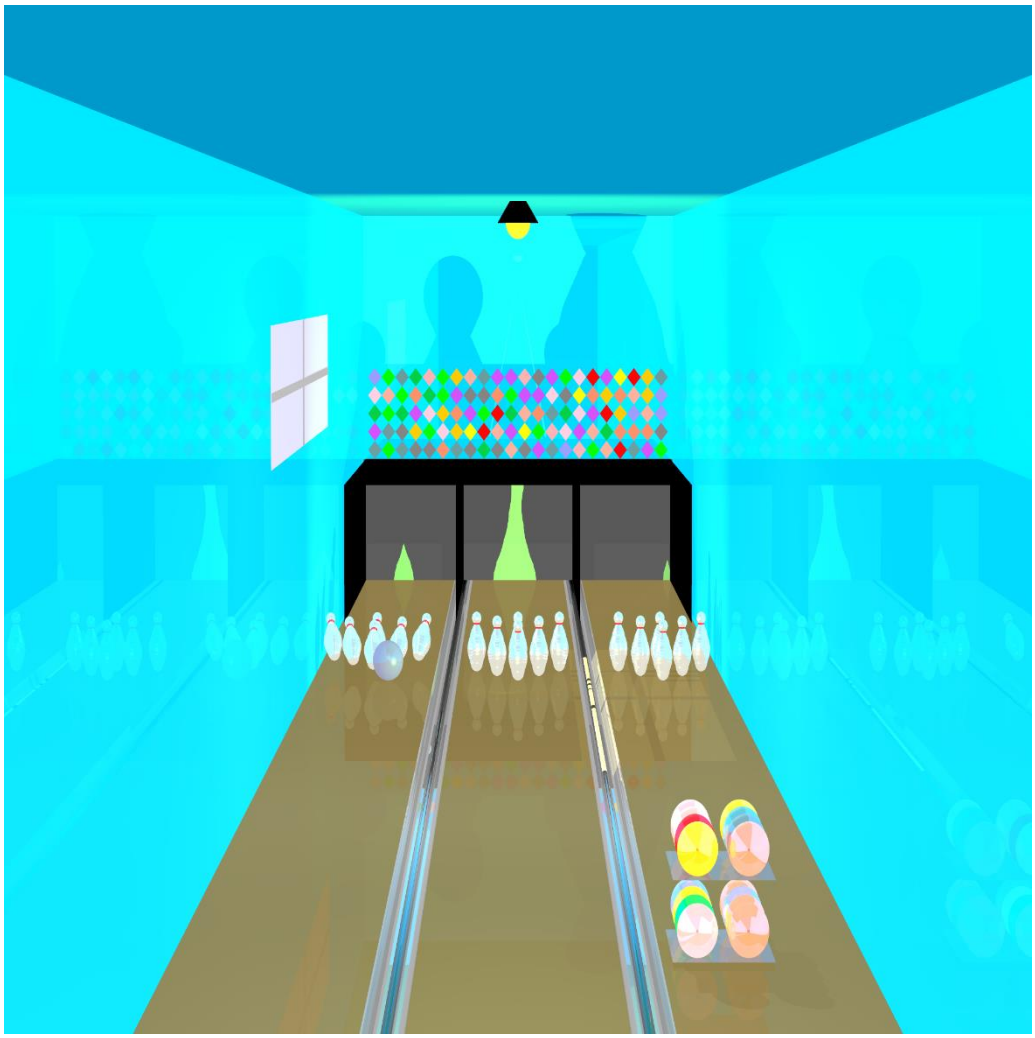
בס״ד

דו״ח שיפורים

מיני פרויקט במבוא להנדסת תוכנה



שרה עדי ציון בדנרש 209779065

ליאורה מרגלית מנדלבאום 214859845

בס"ד

תוכן עניינים

בס"ד

שיפורי תמונה

antialiasing

תיאור:

הבעיה – מכיוון שאנו מחשבים את הצבע על פי מרכז הפיקסל בלבד, נקבל שצורות שאינן ישרות יראו מחוספסות והחלוקה לפיקסלים תבלוט.

השיפור – נשלח מספר קרניים לכל פיקסל ונחשב ממוצע של הצבע, כך שבקצוות הצורה יתקבל צבע משולב והגבולות יהיו "רכים" יותר.

נשתמש ב-super sampling על פי האלגוריתם הרנדומלי (נשלח קרניים שממוקמות באופן רנדומלי בתוך הפיקסל בנוסף לקרן המקורית).

מימוש:

ראשית, נוסיף במחלקת Camera פונקציה renderImageSuperSampling() שתפקידה יהיה לרנדר את התמונה על פי Supersampling.

```
/**
 * render the image using the image writer, using super sampling in the random method
 */
public Camera renderImageSuperSampling() {
    checkExceptions();
    // for each pixel
    for (int i = 0; i < _imageWriter.getNx(); i++) {
        for (int j = 0; j < _imageWriter.getNy(); j++) {
            _imageWriter.writePixel(j, i, castBeamSuperSampling(j, i));
        }
    }
    return this;
}
```

תוכן הפונקציה זהה לזה של renderImage() הרגילה, למעט שימוש ב-castBeamSuperSampling המחשבת את הצבע הממוצע של האלומה, במקום castRay.

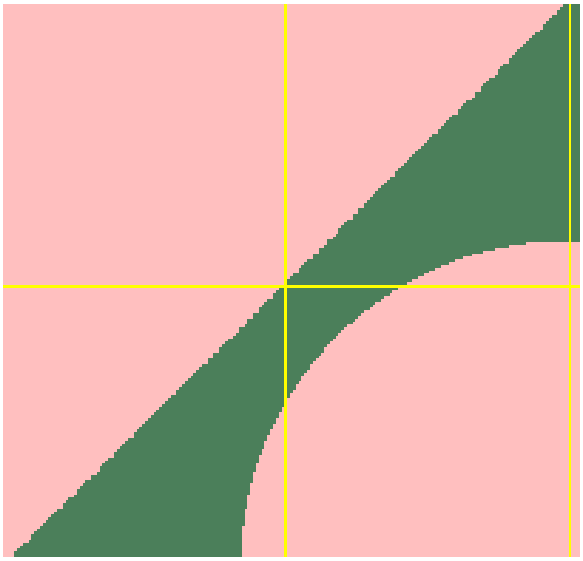
```
/**
 * casts beam of rays around the center ray of pixel
 *
 * @param j col index
 * @param i row index
 * @return Color for a certain pixel
 */
private Color castBeamSuperSampling(int j, int i) {
    List<Ray> beam = constructBeamSuperSampling(_imageWriter.getNx(),
_imageWriter.getNy(), j, i);
    Color color = Color.BLACK;
    // calculate average color of rays traced
    for (Ray ray : beam) {
```

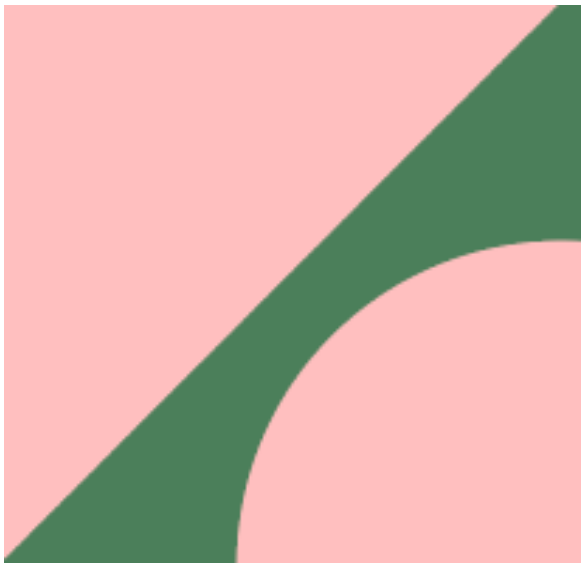
```
        color = color.add(_rayTracerBase.traceRay(ray));
    }
    return color.reduce(_nSS);
}
```

```
/**
 * creates beam of rays around center of pixel randomly
 *
 * @param nX num of row pixels
 * @param nY num of col pixels
 * @param j  col index
 * @param i  row index
 * @return list of rays in beam
 */
private List<Ray> constructBeamSuperSampling(int nX, int nY, int j, int i) {
    List<Ray> beam = new LinkedList<>();
    beam.add(constructRay(nX, nY, j, i));
    double ry = _height / nY;
    double rx = _width / nX;
    double yScale = alignZero((j - nX / 2d) * rx + rx / 2d);
    double xScale = alignZero((i - nY / 2d) * ry + ry / 2d);
    Point pixelCenter = _location.add(_vto.scale(_distance)); // center
    if (!isZero(yScale))
        pixelCenter = pixelCenter.add(_vright.scale(yScale));
    if (!isZero(xScale))
        pixelCenter = pixelCenter.add(_vup.scale(-1 * xScale));
    Random rand = new Random();
    // create rays randomly around the center ray
    for (int c = 0; c < _nSS; c++) {
        // move randomly in the pixel
        double dxfactor = rand.nextBoolean() ? rand.nextDouble() : -1 *
rand.nextDouble();
        double dyfactor = rand.nextBoolean() ? rand.nextDouble() : -1 *
rand.nextDouble();
        double dx = rx * dxfactor;
        double dy = ry * dyfactor;
        Point randomPoint = pixelCenter;
        if (!isZero(dx))
            randomPoint = randomPoint.add(_vright.scale(dx));
        if (!isZero(dy))
            randomPoint = randomPoint.add(_vup.scale(-1 * dy));
        beam.add(new Ray(_location, randomPoint.subtract(_location)));
    }
    return beam;
}
```

בס״ד

תוצאות:





glossy surface and diffuse glass

תיאור:

הבעיה – במשטח משקף או שקוף, נקבל השתקפות או מראה מדויק בקצוותיו, מכיוון שהתפזרות הקרניים אחידה. המראה המתואר אינו מציאותי.

השיפור – בחישוב של קרן שקיפות או קרן השתקפות, נוסיף אלומה של פיזור קרניים סביב הקרן שרוחבה יקבע על פי גודל מקדם glossiness/blurriness, ונעשה ממוצע על צבע הקרניים המתקבל. כך נקבל מראה מטושטש יותר או פחות בהתאם ל"חלביות" של המשטח.

מימוש:

נשנה במחלקת RayTracerBasic את הפונקציה calcGlobalEffects כך שתתחשב באלומת קרניים עבור שקיפות והשתקפות, במקום קרן בודדת. הפונקציה תקרא לפונקציה אחרת שתבנה את האלומה בהתאם לקרן המקורית:

```
/**
 * calculates global lighting effects recursively
 *
 * @param gp    point to calculate color for
 * @param v     direction vector of ray intersection the point
 * @param level depth left for recursive calls
 * @param k     current k
 * @return color calculated for the given point
 */
private Color calcGlobalEffects(GeoPoint gp, Vector v, int level, Double3 k) {
    Color color = Color.BLACK;
    Vector n = gp._geometry.getNormal(gp._point);
    Material material = gp._geometry.getMaterial();
    Double3 kkr = k.product(material._kr);
    if (kkr.greaterThan(MIN_CALC_COLOR_K)) {
        List<Ray> reflectedBeam =
constructBeamAroundRay(constructReflectedRay(gp._point, v, n), n,
                material._kg);
        Color temp = Color.BLACK;
        for (Ray reflected : reflectedBeam) {
            temp = temp.add(calcGlobalEffect(reflected, level, material._kr, kkr));
        }
        color = color.add(temp.reduce(reflectedBeam.size()));
    }
    Double3 kkt = k.product(material._kt);
    if (kkt.greaterThan(MIN_CALC_COLOR_K)) {
        List<Ray> refractedBeam =
constructBeamAroundRay(constructRefractedRay(gp._point, v, n), n,
                1 - material._kb);
        Color temp = Color.BLACK;
        for (Ray refracted : refractedBeam) {
            temp = temp.add(calcGlobalEffect(refracted, level, material._kt, kkt));
```

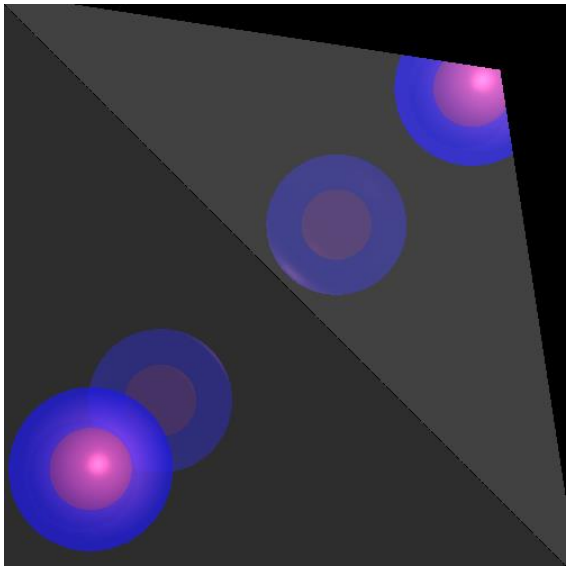
```
        }
        color = color.add(temp.reduce(refractedBeam.size()));
    }
    return color;
}
```

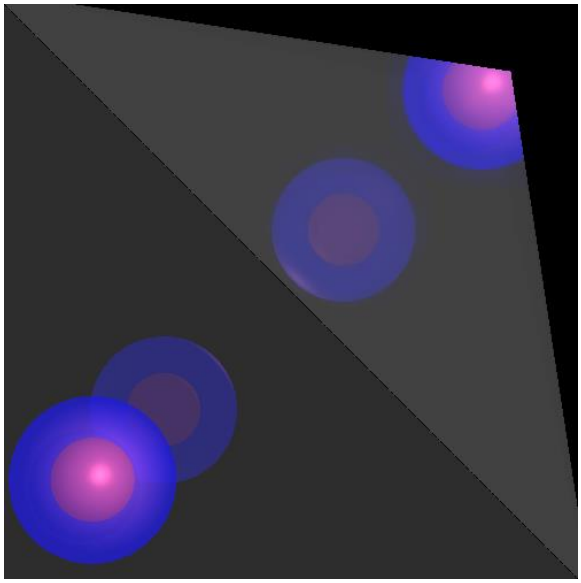
```
/**
 * constructs a beam around a ray according to coefficient
 *
 * @param ray         ray to construct beam around
 * @param n           normal to geometry
 * @param coefficient precision factor
 * @return list of rays in beam
 */
private List<Ray> constructBeamAroundRay(Ray ray, Vector n, double coefficient) {

    List<Ray> beam = new LinkedList<>();
    beam.add(ray);
    double widthFactor = 1 - coefficient;
    Vector ortho;
    // ortho = dir - (dir * n) * n
    if (isZero(ray.getDir().dotProduct(n)))
        ortho = n;
    else
        ortho =
ray.getDir().subtract(n.scale(ray.getDir().dotProduct(n))).normalize();

    for (int i = 0; i < _nRays && !isZero(widthFactor); i++, widthFactor *= 0.9d) {
        ortho = ortho.scale(widthFactor);
        Vector dir = ray.getDir().add(ortho).normalize();
        for (int j = 0; j < _nRays;
             j++, dir = dir.rotate(ray.getDir(), 360d / _nRays)) {
            // if the rays are on the same side of surface
            if (dir.dotProduct(n) * ray.getDir().dotProduct(n) > 0) {
                beam.add(new Ray(ray.getP0(), dir));
            }
        }
    }
    return beam;
}
```

בס״ד

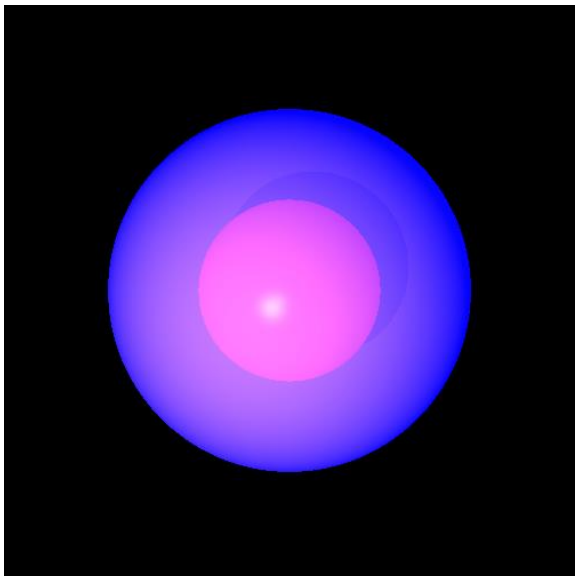
תוצאות:

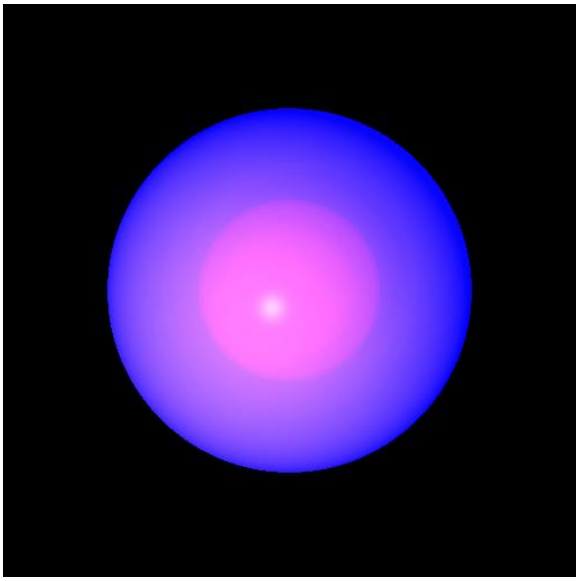
glossy surface





diffuse glass





שיפורי זמן ריצה

adaptive super sampling

תיאור:

הבעיה – באזור שבו צבע הפיקסלים אחיד, נוצר מצב שנשלחות קרניים רבות ללא צורך, מה שגורם לזמן ריצה גבוה מאוד.

השיפור – נדגום 4 קרניים מחלוקה של הפיקסל ל-4 חלקים קטנים יותר. במידה וצבע הקרן המתקבלת שונה מהצבע המתקבל מהקרן המרכזית, נמשיך ברקורסיה לחלק את המיני-פיקסל וכן הלאה, ולבסוף נעשה ממוצע של 4 הצבעים + הצבע מהקרן המרכזית.

מימוש:

נפעל בדומה ל-super sampling. נוסיף במחלקת Camera פונקציה renderImageAdaptiveSuperSampling:

```
/**
 * render the image using the image writer, using adaptive supersampling
 */
public Camera renderImageAdaptiveSuperSampling() {
    checkExceptions();
    // for each pixel
    for (int i = 0; i < _imageWriter.getNx(); i++) {
        for (int j = 0; j < _imageWriter.getNy(); j++) {
            _imageWriter.writePixel(j, i, castBeamAdaptiveSuperSampling(j, i));
        }
    }
    return this;
}
```

שתקרא ל-castBeamAdaptiveSuperSampling:

```
/**
 * casts beam of rays in pixel according to adaptive supersampling
 *
 * @param j col index
 * @param i row index
 * @return Color for a certain pixel
 */
private Color castBeamAdaptiveSuperSampling(int j, int i) {
    Ray center = constructRay(_imageWriter.getNx(), _imageWriter.getNy(), j, i);
    Color centerColor = _rayTracerBase.traceRay(center);
    return calcAdaptiveSuperSampling(_imageWriter.getNx(), _imageWriter.getNy(), j,
i, _maxLevelAdaptiveSS, centerColor);
}
```

שתקרא לפונקציה הרקורסיבית:

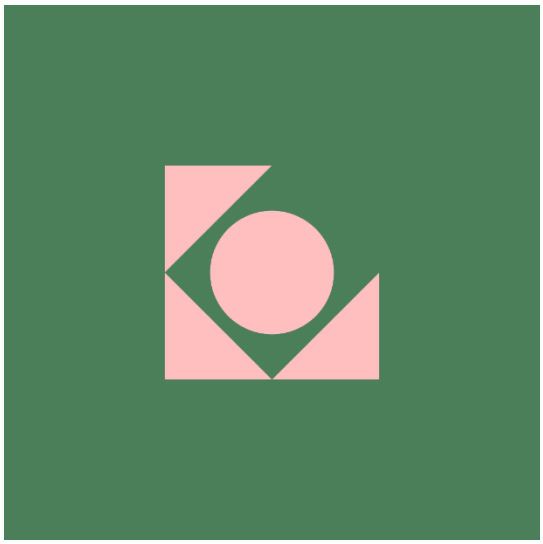
```
/**
 * calculates actual color using adaptive supersampling
 *
 * @param nX    num of rows
 * @param nY    num of cols
 * @param j     col index of pixel
 * @param i     row index of pixel
 * @param level level of recursion
 * @return color of pixel
 */
private Color calcAdaptiveSuperSampling(int nX, int nY, int j, int i, int level,
Color centerColor) {
    // recursion reached maximum level
    if (level == 0) {
        return centerColor;
    }
    Color color = centerColor;
    // divide pixel into 4 mini-pixels
    Ray[] beam = new Ray[]{constructRay(2 * nX, 2 * nY, 2 * j, 2 * i),
            constructRay(2 * nX, 2 * nY, 2 * j, 2 * i + 1),
            constructRay(2 * nX, 2 * nY, 2 * j + 1, 2 * i),
            constructRay(2 * nX, 2 * nY, 2 * j + 1, 2 * i + 1)};
    // for each mini-pixel
    for (int ray = 0; ray < 4; ray++) {
        Color currentColor = _rayTracerBase.traceRay(beam[ray]);
        if (!currentColor.equals(centerColor))
            currentColor = calcAdaptiveSuperSampling(2 * nX, 2 * nY,
                    2 * j + ray / 2, 2 * i + ray % 2, level - 1, currentColor);
        color = color.add(currentColor);
    }
    return color.reduce(5);
}
```

תוצאות:

שימוש ב random: 21.6 שניות

שימוש ב adaptive: 2.3 שניות

שיפור של בערך פי תשע



boundary volume hierarchy

תיאור:

הבעיה – מכיוון שהסצנה מורכבת מגופים רבים, חלקם מאוד קטנים או מורכבים, נקבל שכל קרן בודקת חיתוכים עם כל הגופים למרות שעבור רוב המקרים נקבל שאין חיתוך.

השיפור – ניצור היררכיית גופים תוחמים, המוגדרים כקופסאות סביב האובייקטים, ובכל פעם נבדוק האם הקרן נחתכת איתם (חישוב פשוט ומהיר לעומת החישוב המלא). אם לא, נעצור את הבדיקה, ואם כן, נמשיך מטה בהיררכיה לבדוק האם הקרן נחתכת עם הגופים עצמם.

מימוש:

נוסיף שני שדות למחלקה האבסטרקטית Intersectable שני שדות: bvhIsOn – מקבל ערך true כאשר השיפור דולק עבור הגאומטריה הזו, ושדה box עבור הקופסה התוחמת, מסוג BoundingBox, מחלקה פנימית חדשה בתוך Intersectable:

```
/**
 * class representing boundary box
 */
public class BoundingBox {
    public Point _minimums;
    public Point _maximums;

    public BoundingBox(Point minimums, Point maximums) {
        _minimums = minimums;
        _maximums = maximums;
    }
}
```

בנוסף, נוסיף פונקציה עבור חישוב חיתוכים עם קופסה תוחמת:

```
/**
 * return true if ray intersects object
 *
 * @param ray ray to check
 * @return whether ray intersects box
 * code taken from scratchpixel.com
 */
public boolean isIntersectingBoundingBox(Ray ray) {
    if (!_bvhIsOn || _box == null)
        return true;
    Vector dir = ray.getDir();
    Point p0 = ray.getP0();
    double tmin = (_box._minimums.getX() - p0.getX()) / dir.getX();
    double tmax = (_box._maximums.getX() - p0.getX()) / dir.getX();

    if (tmin > tmax) {
        double temp = tmin;
```

```
        tmin = tmax;
        tmax = temp;
    }


    double tymin = (_box._minimums.getY() - p0.getY()) / dir.getY();
    double tymax = (_box._maximums.getY() - p0.getY()) / dir.getY();

    if (tymin > tymax) {
        double 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._minimums.getZ() - p0.getZ()) / dir.getZ();
    double tzmax = (_box._maximums.getZ() - p0.getZ()) / dir.getZ();

    if (tzmin > tzmax) {
        double temp = tzmin;
        tzmin = tzmax;
        tzmax = temp;
    }

    if ((tmin > tzmax) || (tzmin > tmax))
        return false;

    if (tzmin > tmin)
        tmin = tzmin;

    if (tzmax < tmax)
        tmax = tzmax;

    return true;
}
```

בנוסף, נוסיף פונקציה אבסטרקטית שכל מחלקה שתירש מ-Intersectable תצטרך לממש:

```
/**
 * create boundary box for object
 */
public abstract void createBoundingBox();
```

כעת נוסיף בכל בנאי של גאומטריה את השורות הבאות:

```
if (_bvhIsOn)
    createBoundingBox();
```

ונשנה את המימוש של findGeoIntersections בתוך Intersectable:

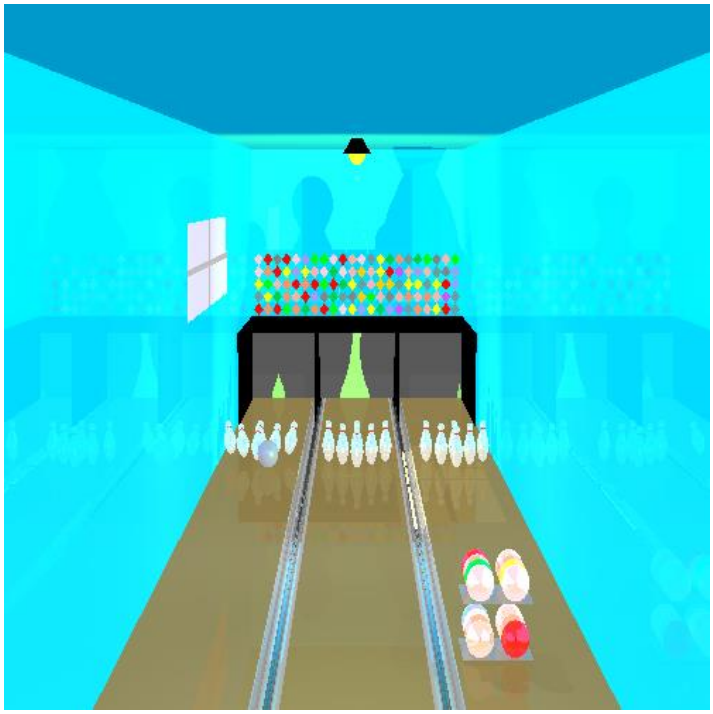
```
/**
 * finds geo intersections between ray and object
 *
 * @param ray ray that intersects
 * @return list of intersection geopoints
 */
public List<GeoPoint> findGeoIntersections(Ray ray) {
    if (_bvhIsOn && !isIntersectingBoundingBox(ray))
        return null;
    return findGeoIntersectionsHelper(ray, Double.POSITIVE_INFINITY);
}
```

כלומר, כעת נחשב את החיתוך של הקרן עם הגאומטריה רק אם היא חותכת את הקופסה התוחמת. אחרת, נדע ישר להחזיר NULL.

תוצאות:

הרצה עם BVH – 13.5 שניות

הרצה ללא BVH- 80 שניות



ביבליוגרפיה

- חומרי עזר מהמודל:

מצגות הקורס התיאורטי, מצגות הקורס המעשי וקובצי קוד.

- Builder pattern:

https://www.geeksforgeeks.org/builder-design-pattern/

- BVH:

https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part2

- תודה רבה לשרה מלכה חמו ותהילה שפייר שעזרו ברעיונות למימושים.
- תודה רבה למרצה אליעזר גנסבורגר.