

## דוח פרויקט

- אלעד פישר (213924624)
- נריה מחפוד (213919616)

## תוכן עניינים

[תוכן עניינים](#)

[מידע כללי על הפרויקט](#)

[הסבר על הבעיות והשיפורים של מיני פרויקט 1](#)

[בעיה 1](#)

[התיקון](#)

[בעיה 2](#)

[התיקון](#)

[השינויים בקוד](#)

[תמונות סופיות](#)

[תמונה סופית עם anti aliasing](#)

[תמונה סופית עם עומק שדה](#)

[תמונה סופית בלי שום שיפור](#)

[הסבר על הבעיות והשיפורים של מיני פרויקט 2](#)

[הבעיה](#)

[תיקונים](#)

[תיקון מספר 1: Multy Threading](#)

[תיקון מספר 2: Bounding Volume](#)

[תיקון מספר 3: בניית עץ היררכיות של Bounding Volumen](#)

[הבעיה](#)

[הפתרון](#)

## מידע כללי על הפרויקט

- ❑ יש גטרים וסטרים לכל השדות שיש כולל תיאור של הפרמטר, הסטרים הם בשיטה של chaining method מתי שביקשו(בעיקר במצלמה ורנדרר).
- ❑ השתמשנו בעיקרון של ניצול של קוד ושיטות בזה שלקחנו את הקוד שעובד מהמצגות של המרצים(בעיקר מצגת 7) וגם לקחנו את הפונקציה שמוצאת את הנקודות חיתוך של גליל סופי

ואינסופי מזוג אחר בקבוצה(דרזנר ורפאל גולדטיין) על מנת להשתמש בהם בתמונה הסופית, וכדי לממש את העיקרון שבקוד שניסו בד"כ יש פחות שגיאות.

❑ השתדלנו בנוסחאות להשתמש בשמות שיש במצגות על מנת שמי שייראה את הנוסחה יוכל בקלות לראות מה כל פרמטר, בשאר הפרוייקט ואיפה שראינו שהשמות לא טובים, בחרנו בשמות משמעותיים.

❑ הוספנו הערות לפי פורמט java Doc איפה שצריך(לפני כל מחלקה, פונקציה, שדה), וכמו כן הוספנו הערות בכל מקום שראינו שהקוד לא ברור מעצמו.

❑ הוספנו מחלקת Box שהיא מקבלת נקודת התחלה ושלוש ווקטורים, אחד לכל כיוון, כדי ליצור תיבה תלת ממדית, או 3 ערכים בשביל ליצור תיבה שהצלעות מקבילות לצירים.

❑ בונוסים כלליים שעשינו לאורך הפרויקט:

❑ תרגיל 2: מימוש נורמל לגליל סופי

❑ תרגיל 3: חיתוך עם מצולע

❑ תרגיל 4: טרנספורמצית מצלמה(הוספנו את זה רק בתרגיל 7, אך הבונוס לא מוגבל בזמן)

❑ תרגיל 6: סיום תוך שבוע

❑ תרגיל 7: כל הבונוסים: תמונה עם מעל 10 אובייקטים, סיבוב התמונה וצילום מכמה זוויות, מימוש בעיית ההצללה בדרך השניה.

❑ מיני פרוייקט 1:מימשנו שני שיפורי תמונה:עומק שדה, אנטי אלייסינג(החלקת עקומות).

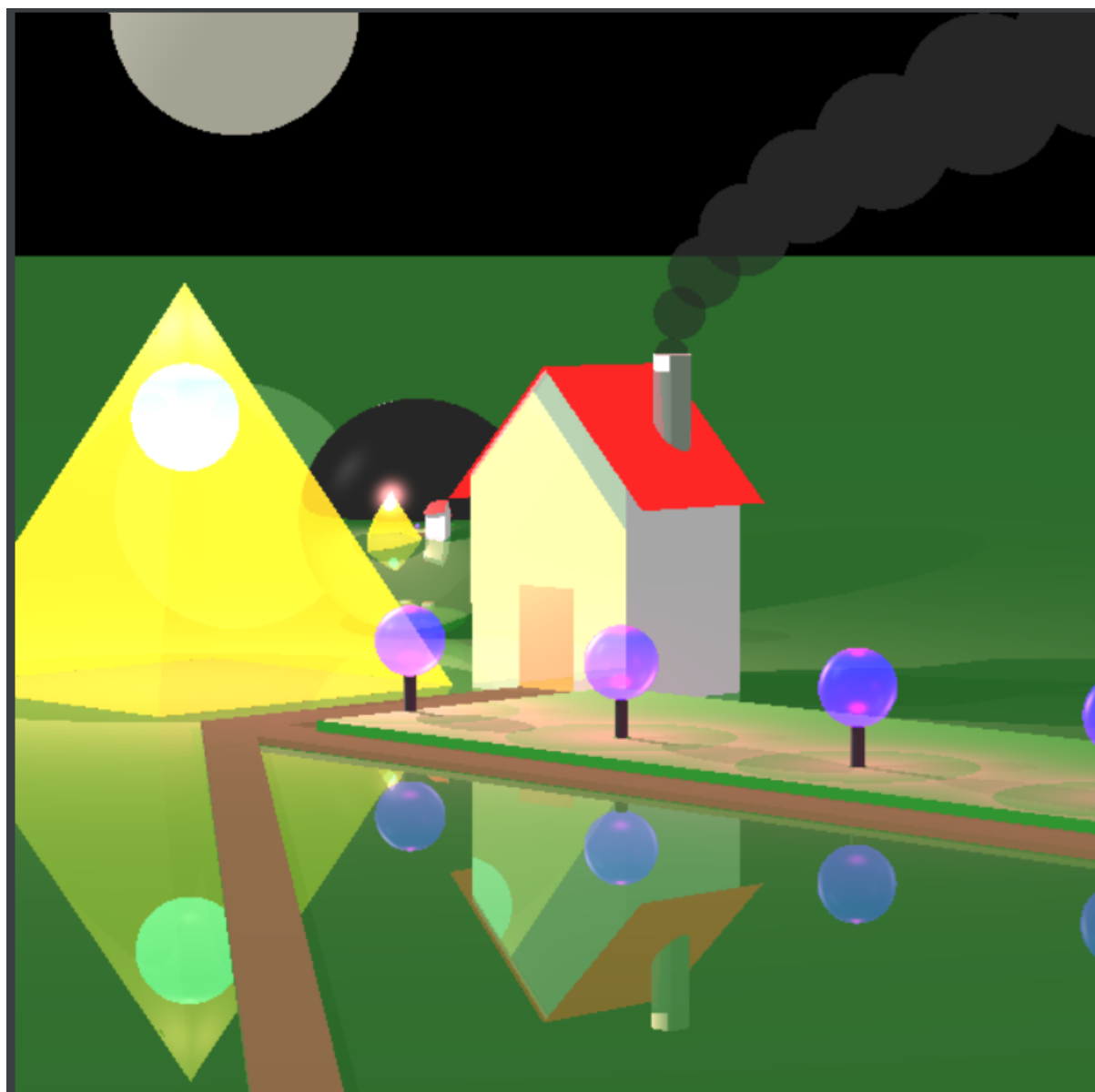
# הסבר על הבעיות והשיפורים של מיני פרויקט 1

## בעיה 1

הקצוות של התמונה לפעמים חדות מאוד ואז התמונה נראית לא טבעית כמו שיש כאן דוגמא לתמונה לא טבעית:



חלק מהתמונה הסופית של "ירח" שנראה קצת מפוקסל בקצוות.



התמונה המלאה, פה ניתן לראות שיש קצוות מאוד חדים בעיקר בשביל לכיוון המצלמה, הפירמידה נראית עם "מדרגות", ובכללי יש עוד מקומות כמו ההשתקפות דרך הכדור השחור, הגג של הבית וההצללה שיש פשוט קווים מפוקסלים ברמות.

## התיקון

אנחנו נשלח מספר מסויים של קרניים לכל פיקסל ולכן מה שיהיה בקצוות ייראה למעשה בצורה יותר עדינה, של שילוב הצבעים לפי כמה הפיקסל קרוב לקצה, וככה זה ייראה בצורה חלקה יותר עם הצבעים.

כדי לשלוח מספר קרניים לכל פיקסל יש כמה שיטות:

1. להשתמש בשיטת מונטה קרלו לחלק פיקסל לחלקים ולקבל נקודה אקראית בכל "תת פיקסל"
  2. אקראי לחלוטין.
  3. לעשות גריד של נקודות.
- אנחנו בחרנו לעשות גריד של נקודות כי זה יחסית מדויק ונותן חלוקה טובה יחסית בלי "שטחים מתים" שאין בהם קרן (הכללנו את הקצוות של הפיקסל כדי לקבל חלוקה מאוד מדויקת).

בכללי אנחנו חושבים שעל פי העיקרון RDD האחריות של היצירה של הקרניים היא של המצלמה שמעלה ככה את האיכות, והרנדרר אחראי רק לרנדר את התמונה בפועל בצורה היבשה. לכן שינינו את והוספנו את היצירה של הקרניים אצל המצלמה, וגם מימשנו את היצירה של הגריד אצל המצלמה כי זה משמש רק אותו וזה אחריות שלו.

השינויים בקוד ובתצורה של הפרוייקט:

שינינו את הפונקציה `constructRaysThroughPixel` שתחזיר רשימה של קרניים עבור אותו פיקסל, ואת הרנדרר שינינו שידע לרנדרר רשימה של קרניים לכל פיקסל ולעשות מהם את הממוצע: השינוי של הרנדרר:

```
for (int i = 0; i < nX; ++i)
  for (int j = 0; j < nY; ++j)
  {
    Color color = getAverageColor(_camera.constructRaysThroughPixel(nX, nY, i, j));
    _imageWriter.writePixel(i, j, color);
  }
```

בפעולת הרנדרר זה יכתוב את הצבע הממוצע שמתקבל ע"י רשימת הקרניים.

```

/** helper function that calculate the average color of
private Color getAverageColor(List<Ray> rays) {
    Color c = Color.BLACK;

    // for each ray in rays we check the color and add
    for (Ray r : rays) {
        c = c.add(_rayTracerBase.traceRay(r));
    }

    return c.reduce(rays.size());
}

```

חישוב הממוצע של הצבע של הקרניים.  
השינוי שעשינו לתהליך יצירת הקרניים במצלמה:

```

List<Ray> rays = null;

if (raysSampling != 1)
    rays = createRaySampling(Pij, ratioY, ratioX);
else
    rays = List.of(new Ray(p0, Pij.subtract(p0)));

```

באן אם רוצים לירות יותר מקרן אחת לפיקסל אנחנו למעשה קוראים לפונקציה שתיצור את הרשימה עבור הפיקסל המסויים ושולחים גם את הגובה והרוחב של הפיקסל.

תהליך יצירת הקרניים לכל פיקסל:

```

//find the 4 corners of the pixel
Point3D ru = pixel.add(vUp.scale(pixelHeight / 2)).add(vRight.scale(pixelWidth / 2));
Point3D rd = pixel.add(vUp.scale(-pixelHeight / 2)).add(vRight.scale(pixelWidth / 2));
Point3D lu = pixel.add(vUp.scale(pixelHeight / 2)).add(vRight.scale(-pixelWidth / 2));
Point3D ld = pixel.add(vUp.scale(-pixelHeight / 2)).add(vRight.scale(-pixelWidth / 2));

List<Ray> res = new LinkedList<>();

List<Point3D> pointsInPixel = generatePointsInPixel(raysSampling, ru, rd, lu, ld);

//create all the rays that intersect the VB through the points
for (Point3D p : pointsInPixel)
{
    res.add(new Ray(point3D, p.subtract(point3D)));
}

return res;
}

```

## בעיה 2

התמונה כולה בפוקוס, ולכן זה לא נראה מציאותי, כמו כן לפקס רק את מה שאני רוצה שיהיה בולט כמו העצם המרכזי בתמונה זה מאוד חשוב לתמונות עתירות פרטים וצורות.

## התיקון

נעשה עוד מישור שכל מה שעליו יהיה בפוקוס, אנחנו נגידר את המישור בתור המרחק בינו לבין הview plane, כך שמה שעל המישור החדש יהיה בפוקוס. אנחנו נגדיר גם "צמצם" של המצלמה שלנו כך שכמה שהוא יהיה יותר גדול, מה שלא יהיה בפוקוס יהיה יותר מטושטש.



## Camera

Size depends on camera aperture

If in focus, all rays return the same color. If not, colors are blended

Focal plane

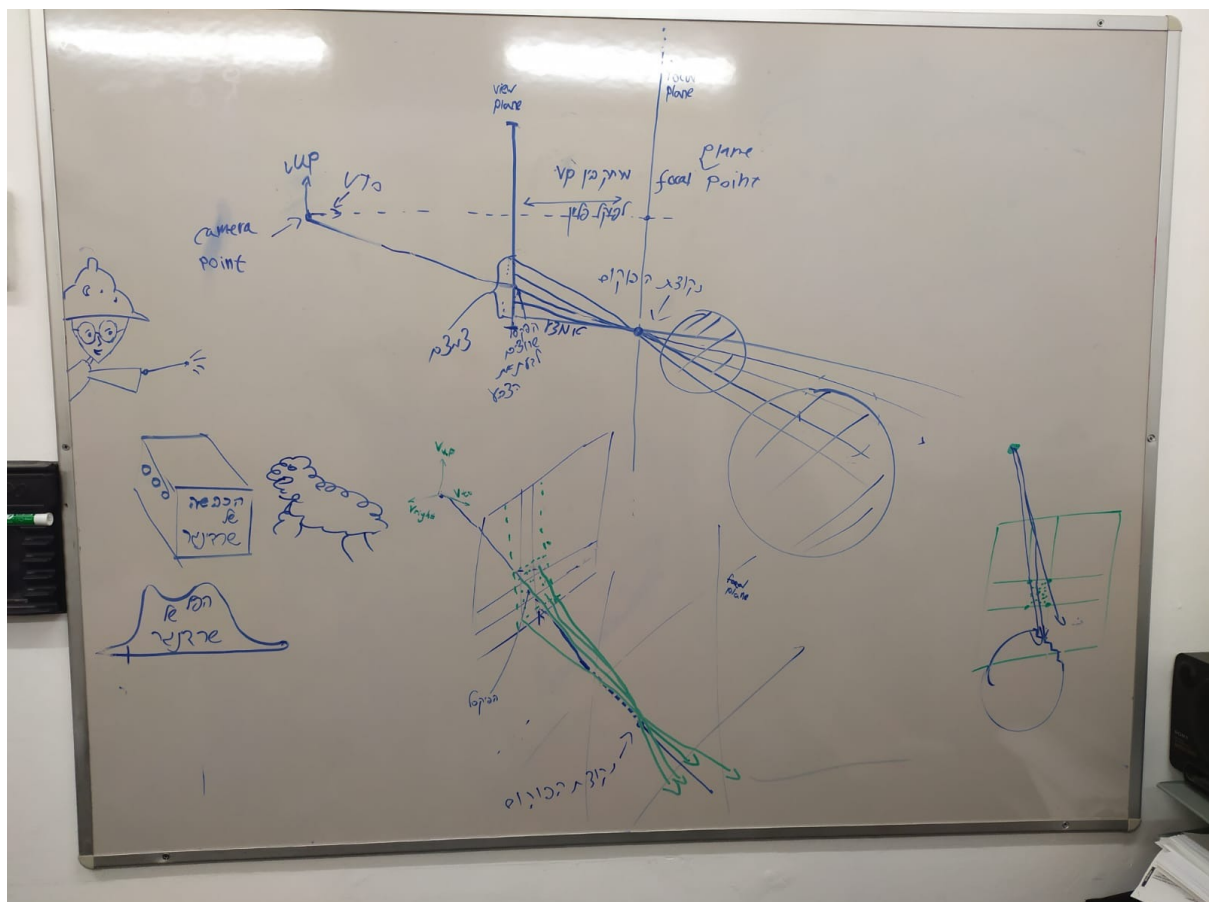
Focal point

View plane

### 3D model

המחשה של הרעיון עם הצמצם והמרחק.

במהלך ההגנה ציירנו על הלוח ציור הממחיש את הלז:



## השינויים בקוד

הרנדור יעבוד באותה דרך, כי אני אחזיר רשימה של קרניים. למצלמה יתווספו השדות הרלוונטיים של הנתונים שהגדרתי לעיל.

הקוד של החישוב קרניים דרך פיקסל ישתנו ככה:

```
private List<Ray> createDepthOfFieldRays(Point3D pij, Ray ray) {
    //calc the focal point
    Point3D focalPoint = focalPlane.findIntersections(ray).get(0);

    //calc the edges of the aperture
    Point3D ru = pij.add(vUp.scale(apertureSize / 2)).add(vRight.scale(apertureSize / 2));
    Point3D rd = pij.add(vUp.scale(-apertureSize / 2)).add(vRight.scale(apertureSize / 2));
    Point3D lu = pij.add(vUp.scale(apertureSize / 2)).add(vRight.scale(-apertureSize / 2));
    Point3D ld = pij.add(vUp.scale(-apertureSize / 2)).add(vRight.scale(-apertureSize / 2));

    //get points on the aperture
    List<Point3D> gridPoints = generatePointsInPixel(samplingDepth, ru, rd, lu, ld);

    /** a beam of rays that go through the pixel and intersect the focal point. */
    List<Ray> r = new LinkedList<Ray>();
    r.add(ray);

    //create all the focused rays
    for (Point3D p : gridPoints)
    {
        r.add(new Ray(p, focalPoint.subtract(p)));
    }

    return r;
}
```

החישוב של הקרני עומק שדה יתבצע ע"י קבלה של הקרן(זה חשוב על מנת לתמוך גם בעומק שדה וגם בהחלקת קצוות) והנקודה שהיא חותכת את view plane (על מנת לחשוך בביצועים).

השינוי בקבלת הקרניים לכל פיקסל יתבטא בזה שניצור רשימה של קרני עומק שדה עבור כל קרן שיש בבלי.



```

//if there is focus generate and return the focused rays
if (0 != focalPlaneDist && samplingDepth != 1)
{
    List<Ray> DOFrays = new LinkedList<>();
    focalPlane = new Plane(vTo, getCenterPoint().add(vTo.scale(focalPlaneDist)));
    |
    //calc and add all the DOF rays
    for (Ray r: rays)
    {
        DOFrays.addAll(createDepthOfFieldRays(Pij,r));
    }

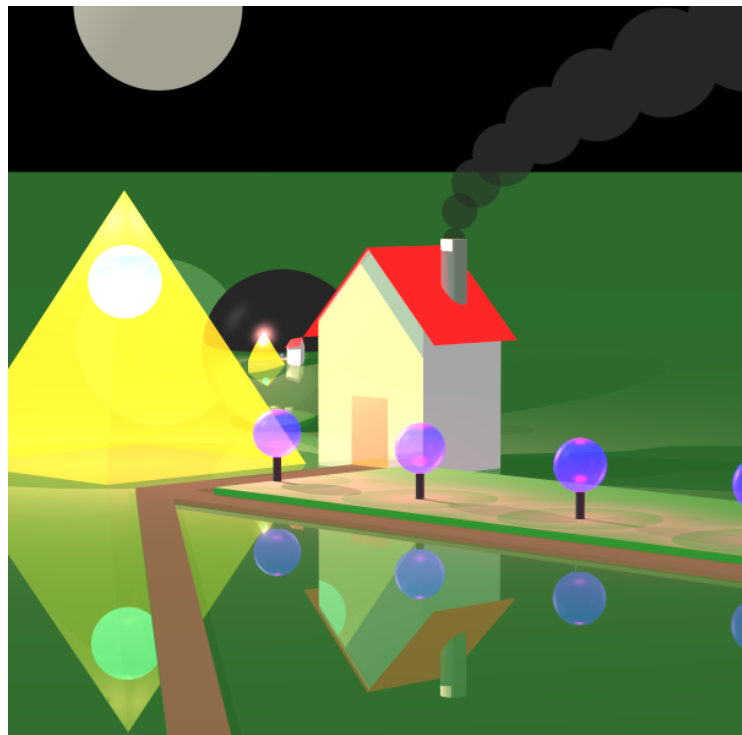
    //return all the DOF rays
    return DOFrays;
}

//if there isn't focus return the rays that calculated before no matter if the rays
else
    return rays;
}

```

## תמונות סופיות

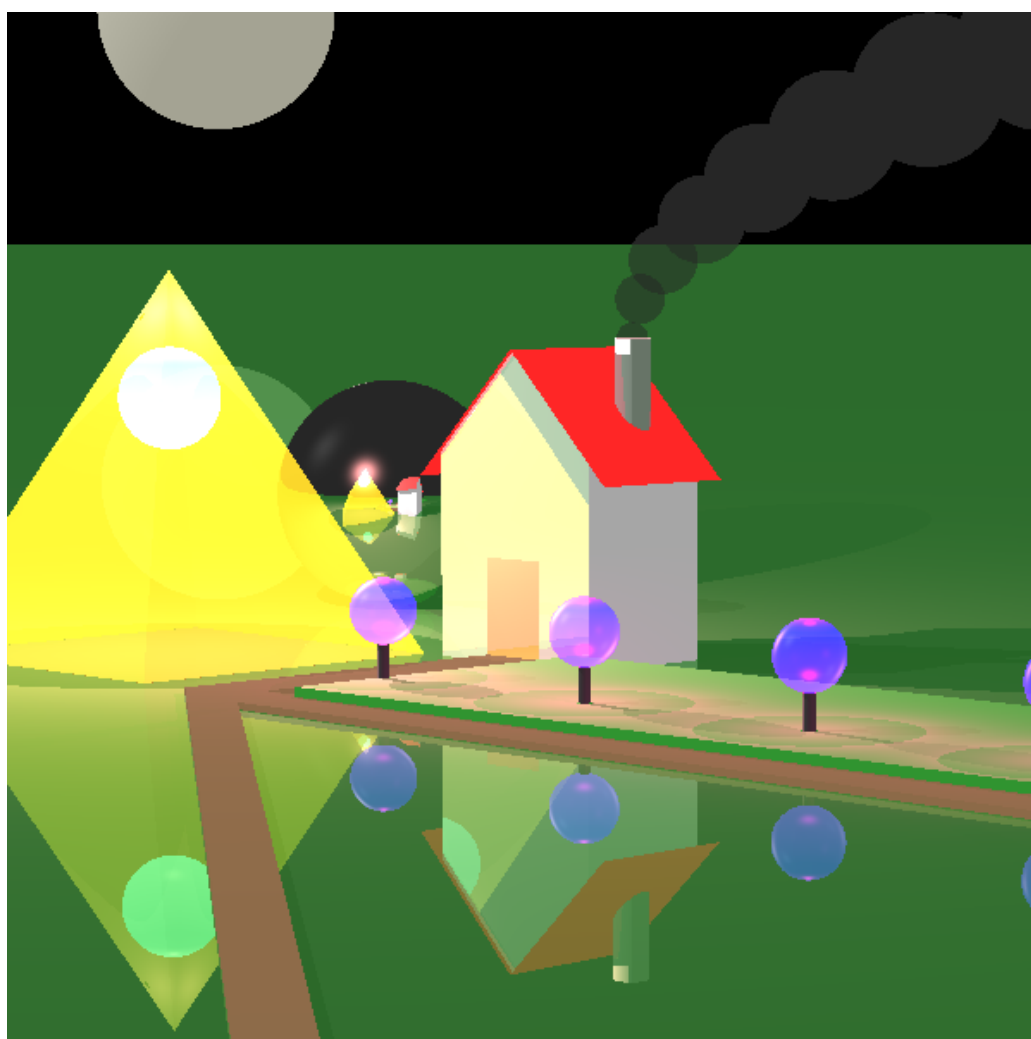
תמונה סופית עם anti aliasing



תמונה סופית עם עומק שדה



תמונה סופית בלי שום שיפור



## הסבר על הבעיות והשיפורים של מיני פרויקט 2

### הבעיה

בחישוב תמונה, צריך לשלוח קרן עבור כל פיקסל, ולבדוק עבור כל גיאומטריה בסצנה האם הקרן חותכת אותה, לחשב את נקודת החיתוך הקרובה ביותר ואז לשלוח קרן עבור כל מקור אור, ואם המישור משקף או שקוף או גם וגם צריך לשלוח עוד קרניים ועבור כל קרן כזו גם לחשב את כל נקודות החיתוך... וכן הלאה. במיני פרויקט אף הגדלנו את מספר הקרניים ל $19 \times 19$ , וכדי שתצא תמונה באמת טובה אנחנו צריכים לשגר כמעט פי 400 קרניים, ובקיצור, זה לוקח הרבה מאוד מן ריצה.

המטרה של מיני פרויקט 2 היא להקטין את זמן הריצה.

### תיקונים

#### תיקון מספר 1: Multy Threading

התיקון הראשון הוא לחלק את העבודה שהמחשב עושה למספר תהליכונים שעובדים במקביל. פתרון זה הוא חסכוני מאוד, מפני שעד עכשיו עבדנו עם תהליך אחד שרץ על מעבד אחד, והוספת תהליכונים מאפשר למחשב לפצל את העבודה לכמה מעבדים, וכך לנצל את המעבדים יותר טוב, ולחסוך בזמן ריצה.

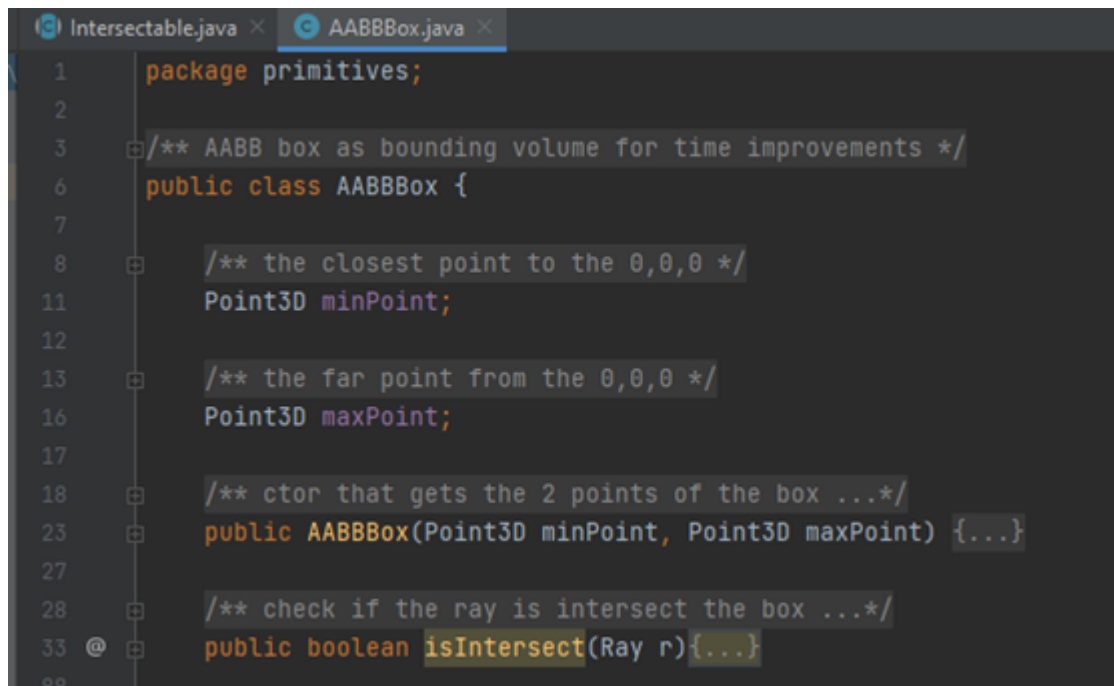
השתמשנו בקובץ התהליכונים ששלח **דן זילברשטיין** עם השינויים כדי שריבוי תהליכונים יתמוך באפקטים שלנו של השלב הראשון בשביל הקורס, וזה באמת הקטין את זמן העבודה פי 2 (!).

#### תיקון מספר 2: Bounding Volume

הנתח הכי גדול ומשמעותי בתהליך הזה הוא חישוב נקודות החיתוך של הקרן עם כל האובייקטים בסצנה, כי עבור כל קרן צריך לעבור על כל האובייקטים ועבור כל אחד מהם צריך לחשב את כל נקודות החיתוך. זה אומר שאם יש לנו עשרות גופים ועשרות אלפי קרניים זה אומר לחשב נקודות חיתוך מאות אלפי פעמים, כך שכל שיפור של זמן ריצה במציאת נקודות חיתוך יכול להיות **מאוד** יעיל.

הפתרון בו השתמשנו נקרא Bounding Box. הפיתרון הזה אומר שבמקום לחשב את נקודת החיתוך עם הגוף עצמו, מה שיכול להיות מסובך וארוך, נחשב נקודת חיתוך עם תיבה המקבילה לצירים שמכילה את האובייקט, ורק אם הקרן באמת חותכת את התיבה נמשיך לחישוב המדויק. זה חוסך המון זמן ריצה כי הבדיקה האם הקרן עוברת בתיבה היא מאוד פשוטה, ורוב הקרניים באמת לא עוברות בקופסה, מה שאומר שרוב הקרניים רק תבדוקנה האם הן עוברות בקופסה (שזה חישוב פשוט ומהיר) ולא תאלצנה לחשב חישוב ארוך כדי לגלות שהן לא חותכות את האובייקט.

ולעומק:



```

1  package primitives;
2
3  /** AABBB box as bounding volume for time improvements */
6  public class AABBBBox {
7
8      /** the closest point to the 0,0,0 */
11     Point3D minPoint;
12
13     /** the far point from the 0,0,0 */
16     Point3D maxPoint;
17
18     /** ctor that gets the 2 points of the box ...*/
23     public AABBBBox(Point3D minPoint, Point3D maxPoint) {...}
27
28     /** check if the ray is intersect the box ...*/
33     public boolean isIntersect(Ray r){...}

```

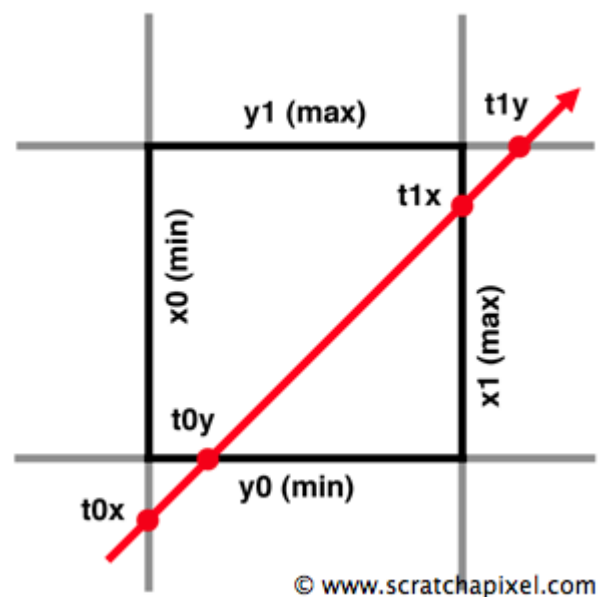
בתמונה רואים את המחלקה החדשה שיצרנו, AABBBBox, שמייצגת את התיבה המכילה את האובייקט. התיבה מורכבת מהנקודה הקרובה לראשית צירים, והנקודה הרחוקה. צריך לשים לב שהפונקציה isIntersect לא מחזירה נקודת חיתוך, אלא האם הקרן עוברת בקופסה, שזה חישוב פשוט בהרבה:

```

Intersectable.java x AABBox.java x
29  /** check if the ray is intersect the box ...*/
35  @ public boolean isIntersect(Ray r) {
36      // Smits' method
37      double tmin, tmax, tymin, tymax, tzmin, tzmax;
38
39      if (r.direction.head.x.coord >= 0) {
40          tmin = (minPoint.getX() - r.head.getX()) / r.direction.head.getX();
41          tmax = (maxPoint.getX() - r.head.getX()) / r.direction.head.getX();
42      } else {
43          tmin = (maxPoint.getX() - r.head.getX()) / r.direction.head.getX();
44          tmax = (minPoint.getX() - r.head.getX()) / r.direction.head.getX();
45      }
46
47      if (r.direction.head.y.coord >= 0) {
48          tymin = (minPoint.getY() - r.head.getY()) / r.direction.head.getY();
49          tymax = (maxPoint.getY() - r.head.getY()) / r.direction.head.getY();
50      } else {
51          tymin = (maxPoint.getY() - r.head.getY()) / r.direction.head.getY();
52          tymax = (minPoint.getY() - r.head.getY()) / r.direction.head.getY();
53      }
54
55      if ((tmin > tymax) || (tymin > tmax))
56          return false;
57
58      if (tymin > tmin)
59          tmin = tymin;
60
61      if (tymax < tmax)
62          tmax = tymax;

```

יותר פשוט להסביר בדו ממד על לוח, אבל בקצרה:



בשביל הפשטות, נניח שהקרן יוצאת מראשית הצירים, היינו הקרן היא וקטור מנורמל בשם  $v$ , והוא חותך את התיבה. אזי, כדי להיכנס לתיבה צריך להכפיל את הוקטור ב  $t$  כלשהו. היינו, אם ראש הוקטור הוא  $(x,y)$ , אז  $(tx,ty)$  היא נקודה בתוך התיבה. זאת אומרת שא  $t$  צריך להיות בין הצד הימני של התיבה לצד השמאלי של התיבה, ו  $ty$  בין התחתית לתקרה של התיבה. היינו:

$$\min X < t \cdot x < \max X$$

$$\min Y < t \cdot y < \max Y$$

במילים אחרות, צריך למצוא ערך  $t$  שמקיים את שני התנאים, ואם לא קיים  $t$  כזה אז הקרן לא חותכת את התיבה. לבדוק אם קיים  $t$  כזה זה חישוב פשוט:

$$\min X < t_0 \cdot x < \max X \Rightarrow \frac{\min X}{x} < t_0 < \frac{\max X}{x}$$

$$\min Y < t_1 \cdot y < \max Y \Rightarrow \frac{\min Y}{y} < t_1 < \frac{\max Y}{y}$$

ה  $X$  וה  $Y$  המקסימליים והמינימליים כבר ידועים לנו (מהנקודה המקסימלית והנקודה המינימלית), וכן ה  $X$  וה  $Y$ , שהם הערכים של ראש הוקטור. עכשיו רק צריך לבדוק אם קיים מקרה בו  $T_0$  ו  $T_1$  יכולים להיות אותו אחד, היינו אם תחומי ההגדרה שלהם חותכים אחד את השני.

כדי להרחיב את זה לתלת ממד פשוט צריך להוסיף עוד תנאי למינימום והמקסימום של  $Z$ .

אם הקרן לא מתחילה ב  $0,0,0$  אז צריך לבדוק כמה הקרן צריכה לעבור על אותו ציר כדי להגיע לתיבה, ולכן במקום לעשות את  $\min X$  במונה נעשה  $\min X$  פחות קואורדינטת ה  $X$  של ראש הקרן, ככה שאם למשל ראש הקרן עם  $X=1$  והתיבה מתחילה ב  $X=4$  אז אם למשל קואורדינטת ה  $X$  של הכיוון זה  $2$  אז  $t$  המינימלי הוא לפחות  $1.5$  כדי שהקרן תגיע ל-  $4$ .

החישובים האלה הם חישובים מאוד פשוטים, רק קצת חילוק ובדיקת תנאים, ולכן במקום לבצע את החישובים הארוכים לגופים מסובכים, עדיף לחשב רק את החישובים הפשוטים אלה ורק במקרים בהם באמת הקרן עוברת בקופסה להמשיך לחישוב הארוך.

אחרי שהסברנו את זה, צריך להבהיר את התיקונים שעשינו לקוד כדי להעיל את התכונה הזאת. התיבה אמורה להקיף כל דבר הניתן לחיתוך, היינו צריך לשים  $AABBBox$  בכל  $Intersectable$ , שזה בעייתי, כי  $Intersectable$  זה ממשק. לכן הפכנו את הממשק למחלקה אבסטרקטית ושינינו את  $Geometry$  שתירש מאינטרסקטאבל ולא שתממש אותו:

```

1 package geometries;
2
3 import ...
4
5
6
7
8
9
10 /**
11  * a class for a 3D graphic model that finds the intersection points between a shape to a ray
12  */
13 public abstract class Intersectable {
14
15     /** the bounding box of the geometry */
16     protected AABBBox boundaryBox;
17
18     geoPoint & intersection
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90 /** getter for the min point of the boundary box ...*/
91 public abstract Point3D getMinPoint();
92
93
94
95
96 /** getter for the max point of the boundary box ...*/
97 public abstract Point3D getMaxPoint();
98
99
100
101
102 /** setter that build a boundary box around the geometry */
103 public void setBoundingBox() { boundaryBox = new AABBBox(getMinPoint(),getMaxPoint()); }
104
105
106
107
108
109 /** getter for the bounding box ...*/
110 public AABBBox getAABBBox() { return boundaryBox; }
111
112
113
114
115
116 }

```

כל אובייקט שניתן לחיתוך יכול גם תיבה, שכל פעם שמבקשים ממנו להחזיר את נקודות החיתוך, לפני שהאובייקט יחשב באמת את נקודות החיתוך, הוא קודם יבדוק אם הקרן בכלל עוברת בתיבה שלו. בשביל לדעת מהי התיבה, צריך לדעת מה הנקודה המקסימלית ומה המינימלית, ועבור כל אובייקט זה שונה.

הפעם לא צריך לשנות כלום ברנדור, מפני שהרנדור לא קשור. כאן כל העבודה היא רק של הגיאומטריות עצמן וזהו. ישנן שתי מחלקות שיורשות ישירות מ-Intersectable, שהן Geometry (שהיא מחלקה אבסטרקטית ולא צריך לשנותה בכלל) ו-Geometries. הדרך בה השתמשנו למצוא את נקודות הקיצון של אוסף הגיאומטריות היא לעבור על כל הגיאומטריות באוסף ולקחת את נקודת הקיצון הכי גדולה (או קטנה, תלוי מאיזה כיוון מסתכלים).



```

73      @Override
74      public Point3D getMinPoint() {
75          List<Point3D> minPs = new LinkedList<>();
76          for (Intersectable g :
77              geometries) {
78              minPs.add(g.getMinPoint());
79          }
80
81          double x = Double.POSITIVE_INFINITY
82              , y = Double.POSITIVE_INFINITY
83              , z = Double.POSITIVE_INFINITY;
84          for (Point3D p :
85              minPs) {
86              if(p.getX()<x)
87                  x=p.getX();
88              if(p.getY()<y)
89                  y=p.getY();
90              if(p.getZ()<z)
91                  z=p.getZ();
92          }
93
94          return new Point3D(x,y,z);
95      }
96
97      @Override
98      public Point3D getMaxPoint() {...}

```

לעיל הפונקציה של minPoint בתור דוגמא.

במובן, כדי להפעיל את התיקון צריך להוסיף את הבדיקה בכל פונקציה למציאת נקודות חיתוך. לדוגמא  
בבדור:

```

93      /** override for the findGeoIntersections function of the geometry interface ...*/
99      @Override
100     public List<GeoPoint> findGeoIntersections(Ray ray, double maxDistance) {
101
102         if(boundingBox!=null)
103             if(!boundingBox.isIntersect(ray))
104                 return null;
105
106         Point3D p0 = ray.getHead();
107         Vector v = ray.getDirection();

```

הוספנו בפונקציה בדיקה אם יש תיבת בונדרי, ואם יש אז האם הקרן חותכת אותה. אם לא אז אין סיכוי שהקרן חותכת את האובייקט וזה מחזיר null. אם הקרן כן חותכת את התיבה, אז ממשיכים לבדוק מהן נקודות החיתוך (אם יש).

אחרי התיקון הזה זמן הריצה ירד פי 1.5 בערך!

## תיקון מספר 3: בניית עץ היררכיות של Bounding Volumen

### הבעיה

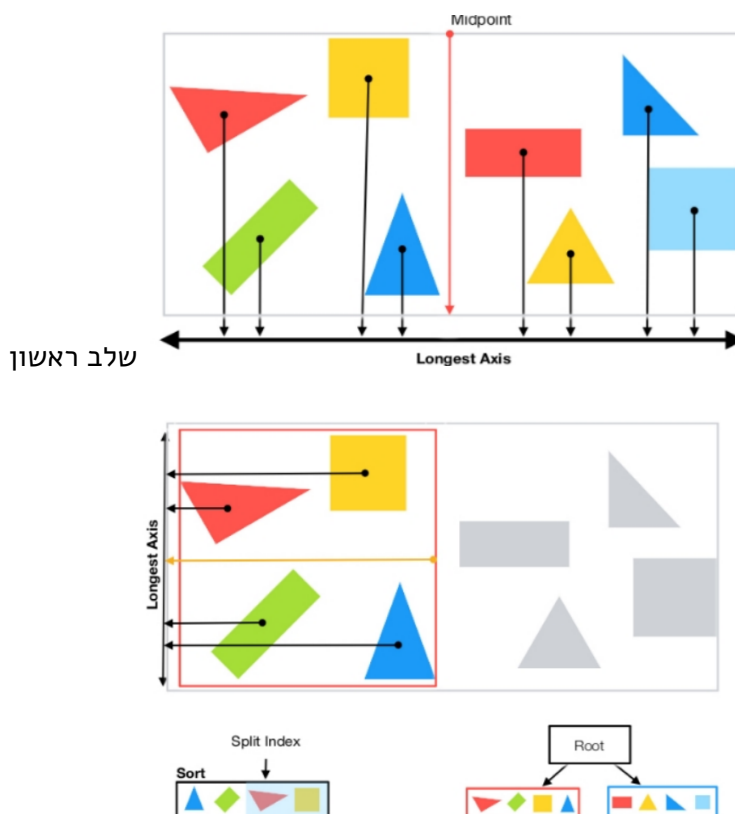
עדיין יש לנו בערך 50 גיאומטריות בתמונה הסופית שלנו ואנחנו בודקים חיתוך עם כל אחד מהבאונדינג ווליום. בדיקה זאת עדיין לוקחת לנו זמן רב.

### הפתרון

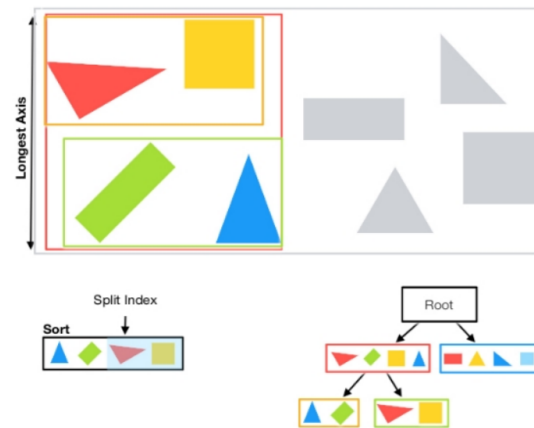
בשלב השלישי עשינו בנייה אוטומטית של עץ היררכיות. הבנייה למעשה תעבוד על פי האלגוריתם הבא:

1. נבנה באונדינג בוקס לכל גיאומטריה בסיסית שקיימת.
2. נבנה את הבאונדינג בוקס של האוסף של הגיאומטריות.
3. נחפש את הציר הגדול ביותר של הבאונדינג בוקס.
4. נמין את כל הצורות על פי נקודת האמצע שלהם והקואורדינטה על הצלע הגדולה ביותר.
5. נבנה לכל צד את האוסף גיאומטריות שלו.
6. נשנה את האוסף גאומטריות לשני האוספים.
7. לחזור על הצעדים 3-6 עד שכל הגופים בתוך באונדינג בוקס, ויש פחות ממספר קטן של גופים בכל עלה(על פי המצגת זה אמור להיות עד שני אובייקטים).

דוגמה ויזואלית של החלוקה :



מחלקים את החלק הימני ל-2.



מפסיקים כי יש 2.

תמונה של המימוש שלנו

```
//build bounding box for each geometry
for (Intersectable g : geometries)
{
    g.setBoundingBox();
}

BuildBoundingBoxHierarchy();
```

באן יש לנו את המימוש של השלבים 1-3 שזה בעצם בונה לכל צורה את הבאונדינג שלה ואז קורה לחלק הרקורסיבי.

החלק הרקורסיבי:

```
//sort the geometries in the geometries list
geometries.sort((Intersectable g1, Intersectable g2) -> { //sorts the points according to the biggest axis
    double sizea = g1.boundingBox.getMiddlePoint().subtract(Point3D.ZERO).dotProduct(axis); //Leave only the coordinate of the biggest axis
    double sizeb = g2.boundingBox.getMiddlePoint().subtract(Point3D.ZERO).dotProduct(axis);
    return Double.compare(sizea, sizeb);
});
```

באן רואים את המיון שנעשה ע"י "מיסוך", אנחנו למעשה מחפשים את נקודת האמצע וככה אנחנו מכפילים בווקטור הכיוון מה שמשאיר רק את הקורדינטה על הישר הגדול ביותר.

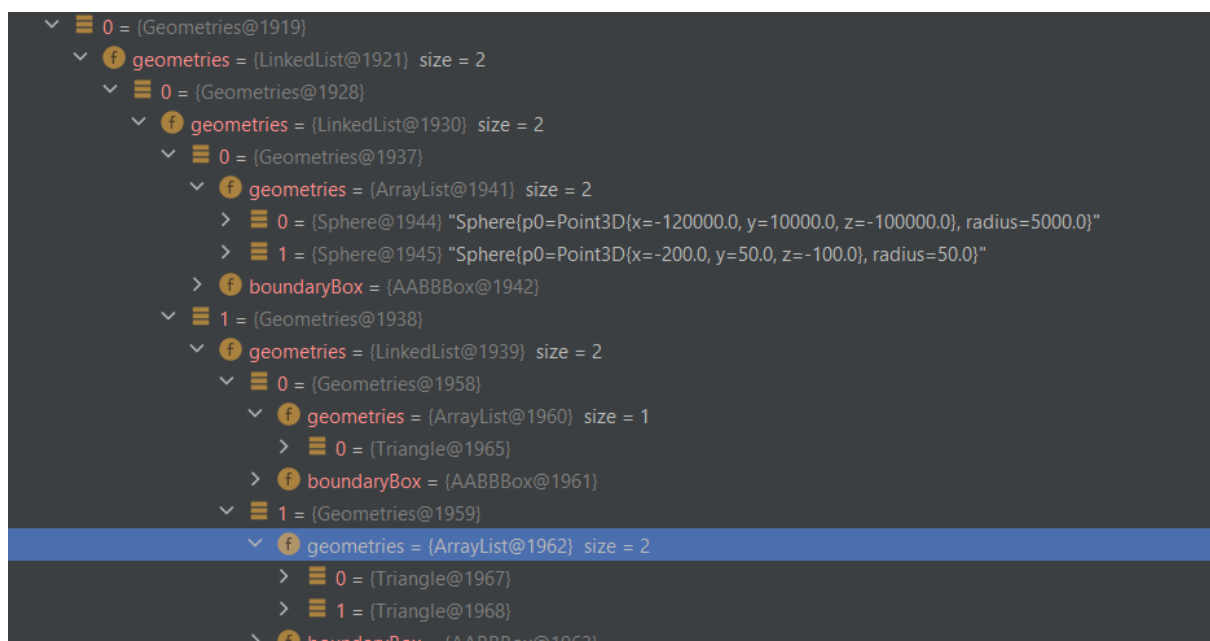
```
int half = geometries.size() / 2;
smaller.geometries.addAll(geometries.subList(0, half));
bigger.geometries.addAll(geometries.subList(half, geometries.size()));
```

באן יש את החלק שבו אנחנו מחלקים ל-2 את הגאומטריות המסודרות

```
//the recursive call for each part
bigger.BuildBoundingBoxHierarchy();
smaller.BuildBoundingBoxHierarchy();

this.geometries = new LinkedList<>();
this.add(smaller, bigger);
```

ובאן יש את החלק שקורא לעצמו רקורסיבית עם כל תת עץ, ומאחד את התשובות ברשימה של הגאומטריות שלו עצמו.



תמונה של חלק מהעץ גאומטריות שנוצר לנו במהלך ריצת האלגוריתם.

אחרי השיפור הזה הזמן ירד גם אבל לא הרבה יותר, כנראה בגלל שבכל מקרה בתמונה שבנינו עשינו גיאומטריות של גופים, היינו הבית היה גיאומטריה אחת וזה אומר שכבר היה סוג של היררכיה של תיבות באונדינג.

בהוספה של כל השיפורים זמן הריצה קטן **פי 4** והגיע ל- 45 דקות לעומת 3 שעות בלי שיפורי ריצה

✓ Test Results	45 m 10 s 979 ms
✓ MP1Tests	45 m 10 s 979 ms
✓ WOW_ImageTest()	45 m 10 s 979 ms

בתמונות בלי שיפורי מראה זמן הריצה קטן בכפי 3-, כנראה בגלל שזמן היצירה של כל האובייקטים יותר משמעותי