דו"ח פרויקט

(מיני פרויקט 2)

- כל הפונקציות מחזירות פרמטרים חדשים תמיד – כולל הפונקציות של חיבור/חיסור, או הכפלה
- וקטורית (למעט נרמול שמתבצע על האיבר עצמו), בהתאם להנחיות רכז הקורס.
- השתמשנו כמעט תמיד בשמות משתנים אינדיקטיביים למעט במקומות בהם העתקנו נוסחאות מתמטיות, במקרה כזה, העתקנו את שמות המשתנים מהנוסחה עצמה.
- הסברים לפני כל פונקציה, ובמקרה הצורך גם in-code כדי לאפשר קריאה נוחה וקלה של הקוד גם למי שקורא אותו לראשונה.
- Javadoc - הסבר פורמלי עבור כל מחלקה.

השיפורים:

הוספנו שני שיפורים למנוע שלנו:

1. Depth of Field ליצירת אפקט עומק (Bokeh) בתמונות, כפי שניתן לראות בתמונה הזאת:

לפני השיפור, כל התמונה בפוקוס, מראה לא טבעי

לאחר השיפור, האובייקטים הקרובים למצלמה והאובייקטים הרחוקים ממנה נראים מטושטשים, כפי שהם נראים בצילום במצלמה אמיתית

על מנת להפעיל את השיפור, יש להפעיל במצלמה המוגדרת בטסט את השורות הבאות:

- קביעת הפעלה של האפקט
- קביעת מרחק מישור הפוקוס
- קביעת גודל הצמצם
- קביעת מספר הקרניים בתוך הצמצם

הדגמה של הפעלת האפקט בעמוד הבא

```
Camera camera = new Camera(
        new Point3D( x: 30,  y: 150,  z: 20),
        new Vector( x: -5,  y: -27,  z: -3),
        new Vector( x: -1.2,  y: -2,  z: 20))
        .set_DOF(true)
        .setFocalDistance(140)
        .setApertureSize(100)
        .setNumberOfRaysInAperture(81)
```

בתמונה המשופרת למעלה קבענו:

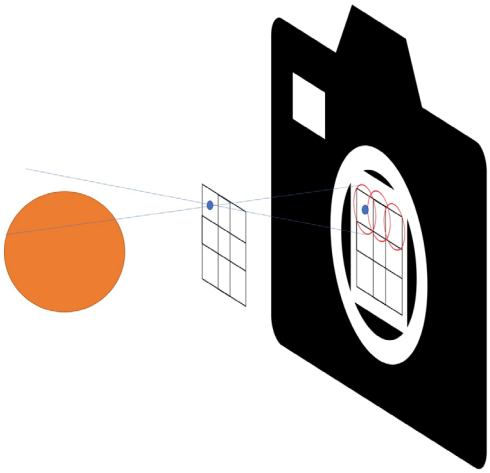
את אפקט העומק להיות true.

את מרחק משטח הפוקוס (ה־focal plane) ל־140.

את גודל הצמצם ל־100.

את מספר הקרניים לכל פיקסל ל־81.

המחשה ויזואלית של ביצוע השיפור:



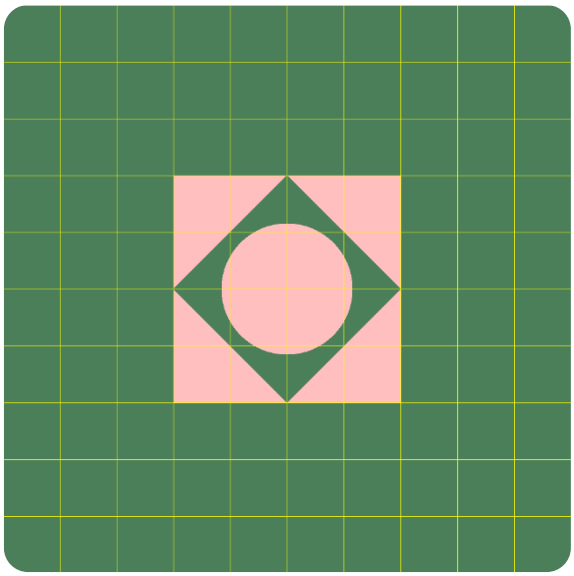
עומק השדה שנמצא בפוקוס מושפע מגודל הצמצם (צמצם גדול יותר יוביל לאפקט מוגבר, כלומר עומק שדה בפוקוס קטן יותר, כלומר יותר טשטוש) וממרחק ה־focal plane מה־view plane (מרחק גדול יותר יוביל ליותר איזורים שנמצאים בפוקוס, מרחק קרוב מאוד יוביל לכך שהסצנה כולה תהיה לא בפוקוס).

הערה:

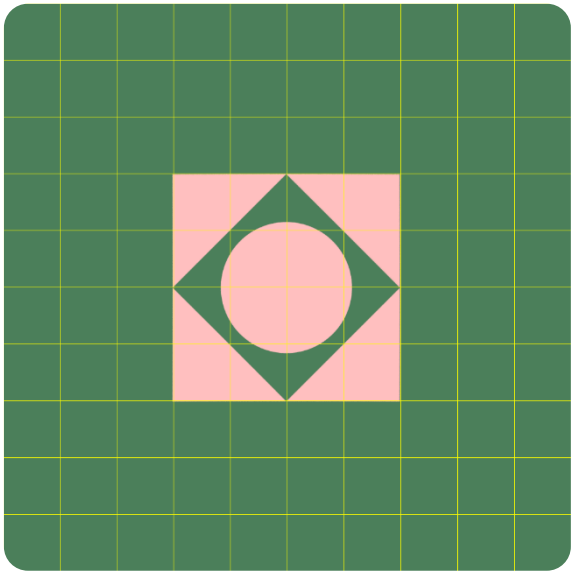
*פיזור הכוכבים מתבצע רנדומלית בכל הרצה, ולכן משתנה מתמונה לתמונה

2. שיפור Anti-Aliasing ליצירת מראה טבעי ומוחלק יותר בקצוות של אובייקטים המופיעים בתמונות, כפי שניתן לראות בהדגמה הפשוטה הזאת:

לפני השיפור, הקצוות של הכדור נראים משוננים בצורה לא טבעית



לאחר השיפור, קצוות הכדור מוחלקים ונראים טבעיים יותר



על מנת להפעיל את השיפור, יש להפעיל במצלמה המוגדרת בטסט את השורות הבאות:

- קביעת הפעלה של האפקט
- קביעת מספר הקרניים לכל פיקסל

הדגמה של הפעלת האפקט בעמוד הבא

```
camera
        .setAA(true)
        .setNumberOfRaysInPixel(36);
```

בתמונה המשופרת למעלה קבענו:

את אפקט ה־AA להיות true.

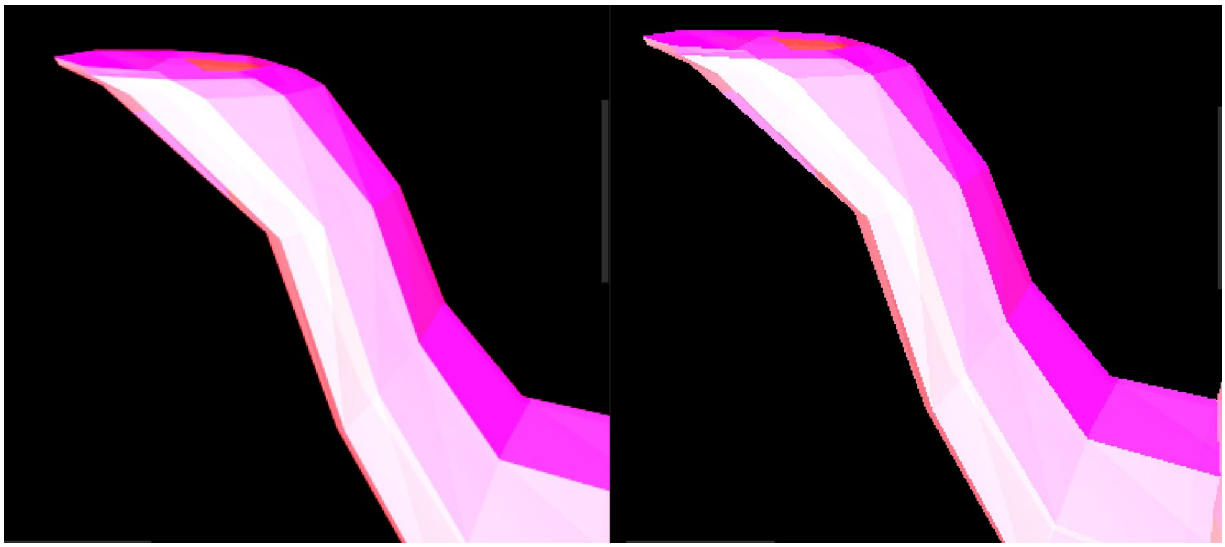
את מספר הקרניים לכל פיקסל ל־36.

הערה:

*מומלץ להגדיל את התמונות כדי לראות ביתר בירור את השפעת האפקט.

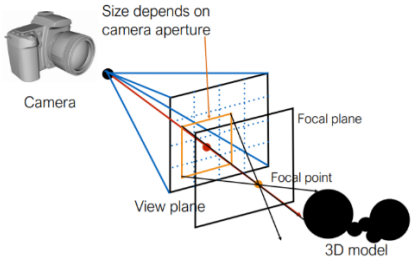
דוגמה נוספת עם התמונה של הקומקום המפורסם (באדיבות מר דן זילברשטיין, בהתאמות קלות):

כמובן מצד ימין ללא AA ומצד ימין עם AA.



הסבר קצר על מימוש השיפורים – Depth of Field:

מתבצע ע"י יצירת צמצם (במקרה שלנו בחרנו לעשות אותו עגול כדי להשיג אפקט טוב יותר) והעברתו על פני כל פיקסל בנפרד, כפי שנראה במצגת ההסבר על השיפורים.



במקום להשתמש בפונקציה הזו שמטילה **קרן בודדת**

```
private void castRay(int nX, int nY, int col, int row) {
```

שקוראת לפונקציה

```
public Ray constructRayThroughPixel(int nX, int nY, int j, int i) {
```

השתמשנו בפונקציה שמטילה **אלומה**

```
private void castBeam(int nX, int nY, int col, int row) {
```

שקוראת לפונקציה שמעבירה **יותר מקרן אחת** לכל פיקסל

```
public List<Ray> constructRaysThroughPixel(int nX, int nY, int j, int i) {
```

בתוך הפונקציה הזו, אם המשתנה שמגדיר כמות קרניים לצמצם אינו 1

```
if (_numberOfRaysInAperture != 1) {
```

ניצור רשימת קרניים מפוזרות בצורה אחידה בשטח הצמצם בעזרת הפונקציה הזו

```
public List<Ray> randomRaysInCircle(
```

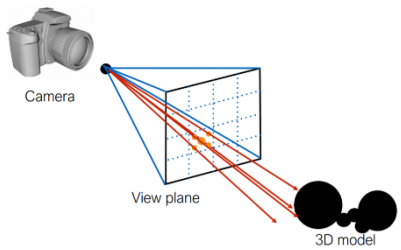
ונחזיר את רשימת הקרניים, נעקוב אחרי כל קרן, נחשב את ממוצע הצבעים עבור הפיקסל, ונכתוב אותו בתמונה, באופן הבא

```
List<Ray> rays = camera.constructRaysThroughPixel(nX, nY, col, row);

List<Color> colors = new LinkedList<>();
for (Ray ray : rays) {
    colors.add(tracer.traceRay(ray));
}
imageWriter.writePixel(col, row, Color.avgColor(colors));
```

הסבר קצר על מימוש השיפורים – Anti-Aliasing:

מתבצע ע"י שליחת קרניים מרובות המפוזרות ברחבי הפיקסל, וחישוב הממוצע עבור כל הקרניים.



במקום להשתמש בפונקציה הזו שמטילה **קרן בודדת**

```
private void castRay(int nX, int nY, int col, int row) {
```

שקוראת לפונקציה

```
public Ray constructRayThroughPixel(int nX, int nY, int j, int i) {
```

השתמשנו בפונקציה שמטילה **אלומה**

```
private void castBeam(int nX, int nY, int col, int row) {
```

שקוראת לפונקציה שמעבירה **יותר מקרן אחת** לכל פיקסל

```
public List<Ray> constructRaysThroughPixel(int nX, int nY, int j, int i) {
```

בתוך הפונקציה הזו, אם המשתנה שמגדיר כמות קרניים לפיקסל אינו 1

```
if (_numberOfRaysInPixel != 1)
```

ניצור רשימת קרניים מפוזרות בצורה אחידה בשטח הצמצם בעזרת הפונקציה הזו

```
public List<Ray> randomRaysInGrid
```

ונחזיר את רשימת הקרניים, נעקוב אחרי כל קרן, נחשב את ממוצע הצבעים עבור הפיקסל, ונכתוב אותו בתמונה, באופן הבא

```
List<Ray> rays = camera.constructRaysThroughPixel(nX, nY, col, row);

List<Color> colors = new LinkedList<>();
for (Ray ray : rays) {
    colors.add(tracer.traceRay(ray));
}
imageWriter.writePixel(col, row, Color.avgColor(colors));
```

השיפורים:

הוספנו שני שיפורים למנוע שלנו:

1. Multi-Threading על מנת לאפשר למנוע להשתמש ביותר מתהליכון בודד, מה שמשיג שיפור ביצועים של פי 2.5 (כמו שכתב רכז הקורס: "כשמשתמשים ב-3 תהליכונים נצפה שיפור של פי 2.5 בערך בזמן ריצה עבור תמונות כבדות יחסית בהשוואה לרינדור ללא תהליכונים"), הדבקנו בקוד שלנו את הקוד שרכז הקורס העלה למודל, וביצענו בו תיקונים על מנת לאפשר הדפסה של אחוזי ההתקדמות של הרינדור למרות שתחילה היה נראה שסביבת העבודה IntelliJ לא מבצעת את ההדפסה בשימוש בקוד כפי שניתן ע"י הרכז.

על מנת להפעיל את השיפור, יש להגדיר ב־RayTracer המוגדר בטסט:

- הפעלת Multi-Threading עם הגדרת מספר התהליכונים שנרצה להקצות
- הפעלת הדפסת האחוזים

הדגמה של הפעלת האפקט

```
.setRayTracer(new BasicRayTracer(scene))
.setMultithreading(3)
.setDebugPrint();
```

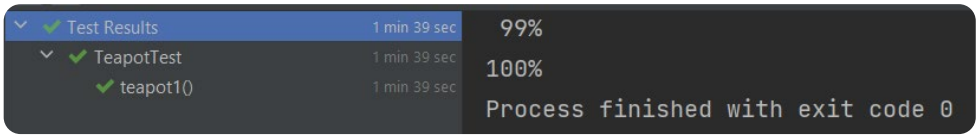
בתמונה המשופרת למעלה קבענו:

את מספר התהליכונים המוקצים לתיצוג (רינדור) ל־3.

הפעלת הדפסת האחוזים.

זמן ריצה לפני השיפור:

זמן ריצה לאחר השיפור (הוקצו 3 ליבות), ובנוסף הדפסת אחוזי התקדמות:



בסך הכל שיפור של פי 226/99 כלומר שיפור של קצת יותר מפי 2.25 בסך הכל.

2. Bounding Boxes – על מנת לחסוך את חישוב חיתוך הקרניים עם כל האובייקטים בסצנה (כלומר עבור כל קרן נצטרך לבדוק את כל הגיאומטריות), נבצע חסימה פשוטה של כל אובייקט בסצנה בקופסה חוסמת (bounding box) שתכיל בצורה מלאה את האובייקט (כמובן למעט אובייקטים אינסופיים, שלא ניתן להגדיר עבורם קופסה חוסמת) ולאחר מכן תתבצע בנייה (אוטומטית, לפי אלגוריתם שקיבלנו) של עץ היררכי בעזרת הפונקציה הבאה

```
public void BuildTree() {

    this.flatten();
    double distance;
    Container bestGeometry1 = null;
    Container bestGeometry2 = null;

    while (_containers.size() > 1) {
        double best = Double.MAX_VALUE;
        for (Container geometry1 : _containers) {
            for (Container geometry2 : _containers) {
                if (geometry1._boundingBox == null || geometry2._boundingBox == null) {
                    System.out.println("hello bug");
                }
                distance = geometry1._boundingBox.BoundingBoxDistance(geometry2._boundingBox);
                if (!geometry1.equals(geometry2) && distance < best) {
                    best = distance;
                    bestGeometry1 = geometry1;
                    bestGeometry2 = geometry2;
                }
            }
        }
        _containers.add(new Geometries(bestGeometry1, bestGeometry2));
        _containers.remove(bestGeometry1);
        _containers.remove(bestGeometry2);
    }
}
```

הסבר קצר על הפונקציה:

1. שיטוח של כל הגיאומטריות (למקרה שיש גיאומטריות מורכבות, כמו קובייה/פירמידה/וכד').
2. כל עוד אין שורש עץ מוגדר (כלומר אין קופסה מכילה של הכל)
3. מצא את שני המועמדים הטובים ביותר להכנסה לקופסה
4. הכנס אותם לקופסה
5. הכנס את הקופסה במקומם ל־containers.

בסוף התהליך נקבל עץ בינארי (כמובן שורש בודד) שמייצג היררכית את כל הגיאומטריות בסצנה.

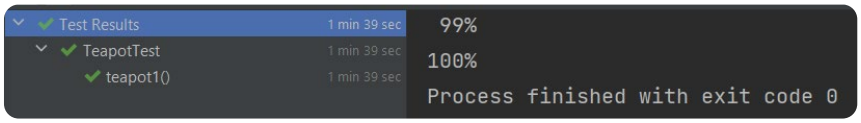
לאחר יצירת העץ ההיררכי, ה־RayTracer שלנו ימשיך לעבוד כרגיל, אבל הפעם, במקום לבדוק חיתוכים של הקרן עם כל גיאומטריה בעזרת הפונקציה findGeoIntersections, נשתמש בפונקציה findIntersectBoundingRegion, שהמימוש שלה כך

```
public List<GeoPoint> findIntersectBoundingRegion(Ray ray) {
    if (_boundingBox == null || _boundingBox.intersectBV(ray)) {
        return findGeoIntersections(ray, Double.POSITIVE_INFINITY, bb: true);
    }
    return null;
}
```

כלומר נבדוק אם לא קיימת קופסה חוסמת (כלומר אנחנו בודקים חיתוך עם גיאומטריה ממש, כלומר הגענו לעלה בעץ) או שהקרן חותכת את הקופסה החוסמת, נבצע בדיקת חיתוך, אחרת נוכל לומר מראש שאין טעם לבדוק חיתוכים ולכן נקבל null (ובסופו של דבר הקרן תחזיר את צבע הרקע).

זמן ריצה לפני השיפור:

תמונה 1 (הקומקום)

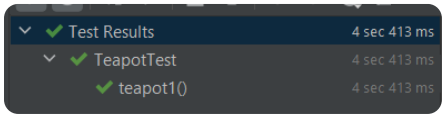


תמונה 2 (סצנת עצים)

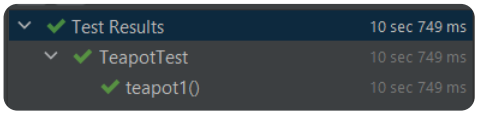
זמן ריצה לאחר השיפור:

תמונה 1 (הקומקום)

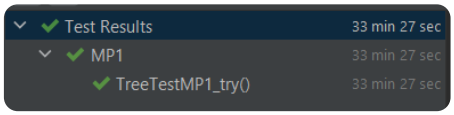
ללא בניית עץ (ללא תקורה)



עם בניית עץ (תקורה נמוכה)



תמונה 2 (סצנת עצים)



בתמונה 1 נצפה שיפור של 99/4.4 או 99/10.75 כלומר שיפור של פי 9.2 עד פי 22.5.

בתמונה 2 נצפה שיפור של 448/33.5 כלומר שיפור של בערך פי 13.5.

השיפור יבוא לידי ביטוי בצורה שונה בכל פעם, בהתאם למורכבות הסצנה, כלומר בהתאם לכמות הגיאומטריות המוכלות בה ופיזורן במרחב.

בכל מקרה, הצלחנו להשיג שיפור של לכל הפחות פי 20.7 (ולכל היותר בערך פי 50!).