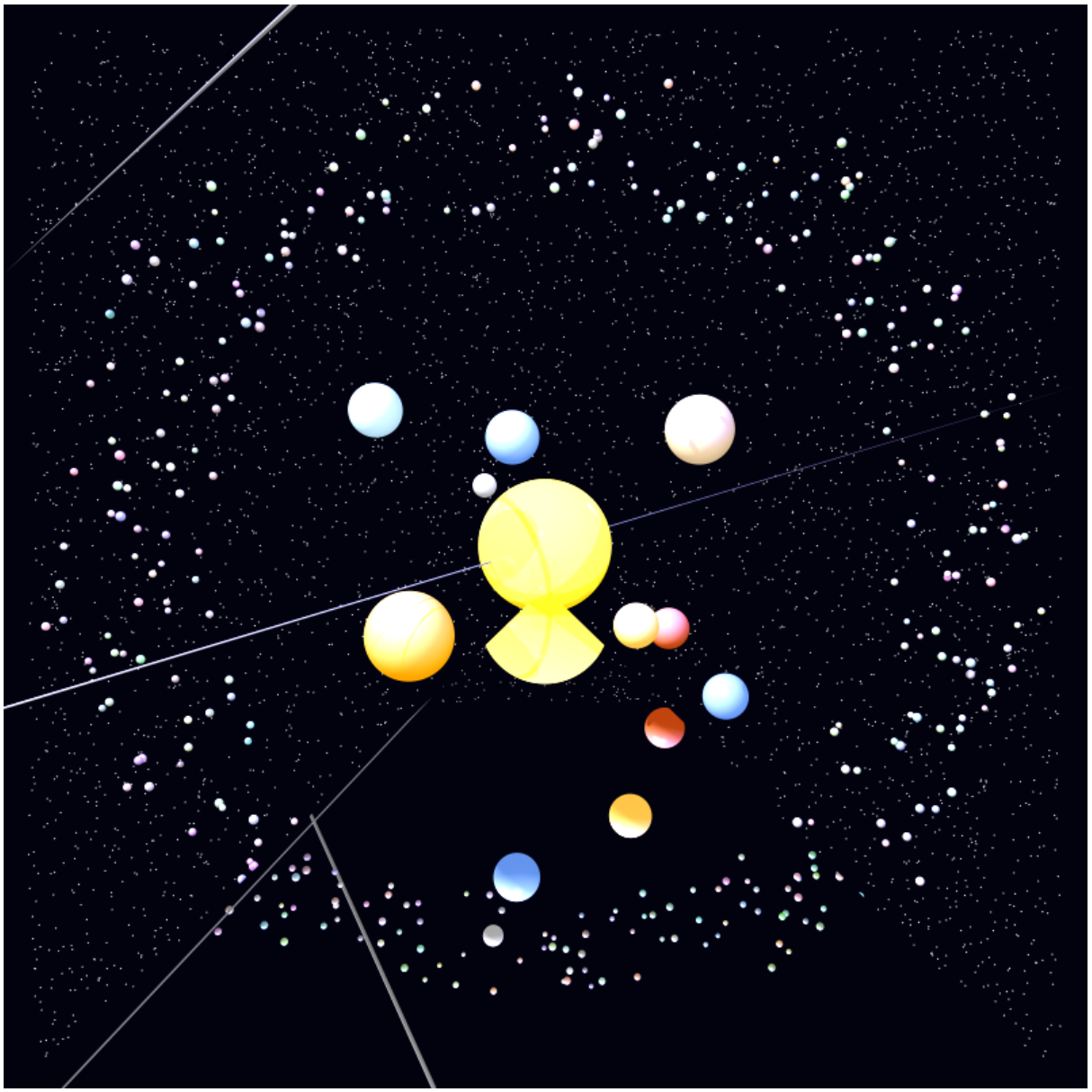
דוח שיפורים

מיני פרויקט במבוא להנדסת תכנה



לאה ברדוגו 341032068

חנה אזולאי 209861400

תוכן עניינים:

שיפורי תמונה

Depth of Field

תיאור:

הבעיה- כשלא משתמשים בDoF, כל האובייקטים בתמונה הם ברורים באותה מידה, גם אם הם נמצאים במרחקים שונים מהמיקוד של המצלמה. זה לא דומה למה שמצלמה אמיתית עושה, ולכן התמונה יכולה להראות פחות טבעית.

הפיתרון- במקום לשלוח קרן אחד לכל פיקסל בview plane, נשלח הרבה קרניים בכל פיקסל שיטשטשו חלק מהתמונה.

מימוש:

הוספנו למצלמה את הפרמטרים הבאים:

```
//aperture properties
private int APERTURE_NUMBER_OF_POINTS=9;
private double depthOfField;  4 usages
private double aperture;  6 usages
private boolean DofON=false;  3 usages
```

1.פרמטר בשם APERTURE_NUMBER_OF_POINTS שמגדיר כמה קרניים נשלח בכל פיקסל.

2.פרמטר בשם depthOfField שמגדיר את המרחק בין ה view plane המקורי ל view plane שניצור בשביל השיפור הזה.

3.פרמטר aperture שמגדיר את הקוטר של העדשה של המצלמה (ככל שהוא יותר גדול, ככה רק חלק קטן מהסצנה יהיה בפוקוס.

4.פרמטר בוליאני DofON- כמו כפתור כדי להפעיל או לא את השיפור הזה.

כתבנו סטרים וגטרים לכל שדה.

פונקציה זו-constructRayGridDOF יוצרת אוסף של קרניים לשיפור זה.

היא מחשבת את הנקודת פוקוס, מחשבת גודל פיקסל.

ואז נלך לפונקציה שתחזיר את הקרניים (generateRayGrid).

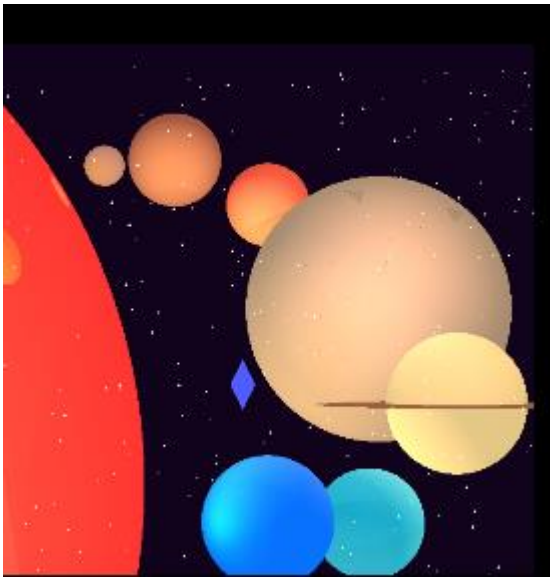
השתמשנו באלגוריתם super sampling jitter- נחלק את הפיקסל לתתי פיקסל ובכל תת פיקסל נשלח קרן רנדומלית.

```java
/**
 * Constructs a grid of rays for depth of field (DOF) effects based on a given primary ray.
 * This method computes a grid of rays by applying DOF to simulate the effect of a camera lens.
 *
 * @param ray          The primary {@link Ray} used to determine the focus point for depth of field effects.
 * @return             A list of {@link Ray} objects representing the rays generated for depth of field simulation.
 */
private List<Ray> constructRayGridDOF(Ray ray)  1 usage  Hannah Michal Azoulay +1
{
    // Compute the focus point for depth of field
    double t0=depthOfField+distance;
    double t=t0/(vTo.dotProduct(ray.getDirection()));
    Point FocusPoint=ray.GetPoint(t);

    // Calculate the size of each pixel for the grid
    double PixelSize =alignZero( number: (aperture*2)/APERTURE_NUMBER_OF_POINTS);

    // Generate a grid of rays for DOF
    return generateRayGrid(location, APERTURE_NUMBER_OF_POINTS, PixelSize, isDOF: true, FocusPoint);
}
```

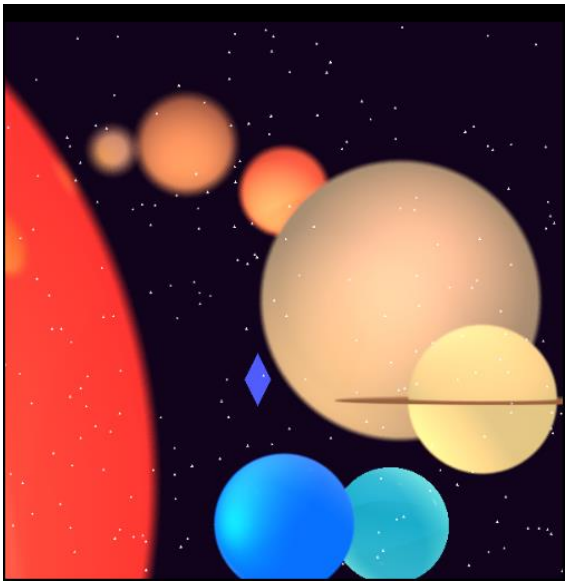
```java
private List<Ray> generateRayGrid(Point center, int gridSize, double pixelSize, boolean isDOF, Point focusPoint) {  2 usa
    List<Ray> rays = new LinkedList<>();
    Random r = new Random();
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            // Compute the offset for the current pixel in the grid
            double xJ = ((j + r.nextDouble() / (r.nextBoolean() ? 2 : -2)) - ((gridSize - 1) / 2d)) * pixelSize;
            double yI = -((i + r.nextDouble() / (r.nextBoolean() ? 2 : -2)) - ((gridSize - 1) / 2d)) * pixelSize;

            // Calculate the intersection point of the ray with the view plane
            Point pIJ = center;
            if (!isZero(xJ)) {
                pIJ = pIJ.add(vRight.scale(xJ));
            }
            if (!isZero(yI)) {
                pIJ = pIJ.add(vUp.scale(yI));
            }
            // Determine the direction of the ray based on DOF or anti-aliasing
            Vector vIJ = isDOF ? focusPoint.subtract(pIJ) : pIJ.subtract(location);
            Ray ray = new Ray(pIJ, vIJ);

            // Add the ray to the list if it's within the DOF aperture or if DOF is not applied
            if (!isDOF || pIJ.equals(location) ||
                    pIJ.subtract(location).dotProduct(pIJ.subtract(location)) <= aperture * aperture) {
                rays.add(ray);
            }
        }
    }
    return rays;
```

תוצאה:

בלי Depth of Field:



עם Depth of Field:



:AntiAliasing

תיאור:

הבעיה-הגופים והצורות נראים לא חלקים ומקוטעים בקצוות.

הפיתרון- נשלח הרבה קרניים לכל פיקסל, ונחשב ממוצע הצבע.

מימוש:

הוספנו את השדות הבאים:

```
//anti-aliasing properties
private int ANTI_ALIASING_NUMBER_OF_RAYS = 1;  3 usages
private boolean antiAliasing = false;  2 usages
```

1.פרמטר בשם ANTI_ALIASING_NUMBER_OF_RAYS שמסמל כמה קרניים נשלח לכל פיקסל.(מספר השורות ומספר העמודות ב gridבכל פיקסל)

2.פרמטר בשם antiAliasing שהוא כמו כפתור-כדי להפעיל או לא את השיפור.

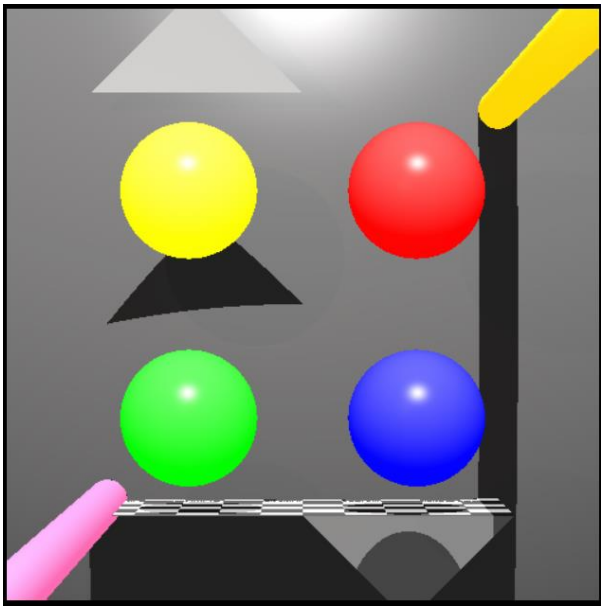
כתבנו סטרים וגטרים לפרמטרים האלה.

כתבנו את הפונקציה calcAveragePixelColor שמקבלת כמות הפיקסלים בשורות ועמודות ב grid שלנו, וכן שורה ועמודה של הפיקסל שאני מחשבת עבורו את הצבע, ושולחת הרבה קרניים (לפי super sampling jitter) בעזרת אותה הפונקציה שהשתמשנו בDoF ומחשבת את הממוצע של הצבע בפיקסל זה.

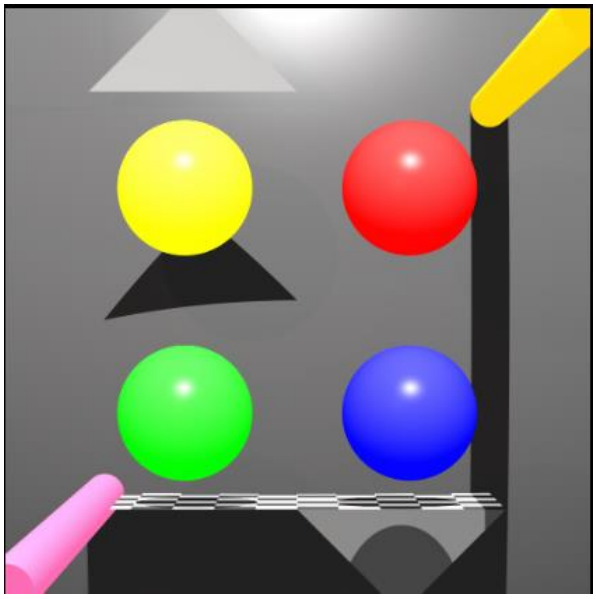
```
private Color calcAveragePixelColor(int nX, int nY, int j, int i) {  1 usage  Lea Berdugo
    // Calculate the width and height of each pixel
    double pixelWidth = width / (double) nX;
    double pixelHeight = height / (double) nY;
    double pixelSize = Math.max(pixelWidth, pixelHeight) / ANTI_ALIASING_NUMBER_OF_RAYS;
    // Calculate the offset of the pixel from the center of the view plane
    double xOffset = (j - (nX - 1) / 2d) * pixelWidth + pixelWidth / 2d;
    double yOffset = -(i - (nY - 1) / 2d) * pixelHeight - pixelHeight / 2d;

    // Calculate the intersection point of the ray with the view plane
    Point pC = location.add(vTo.scale(distance));
    if(!isZero(xOffset)) pC = pC.add(vRight.scale(xOffset));
    if(!isZero(yOffset)) pC = pC.add(vUp.scale(yOffset));


    // Generate a grid of rays for anti-aliasing
    List<Ray> rays = generateRayGrid(pC, ANTI_ALIASING_NUMBER_OF_RAYS, pixelSize, isDOF: false, focusPoint: null);

    // Trace each ray and compute the average color
    Color color = Color.BLACK;
    for (Ray ray : rays) {
        color = color.add(rayTracer.traceRay(ray));
    }
    return color.reduce(rays.size());
}
```

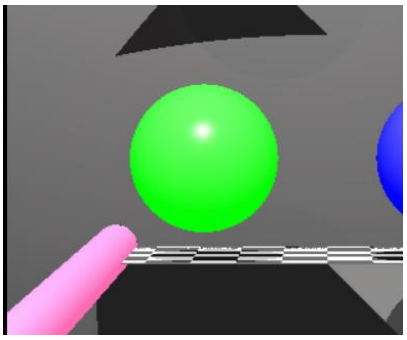
תוצאה:

בלי Anti Aliasing:



עם Anti Aliasing:





שיפורי זמני ריצה

Multi Threading:

תיאור:

בעיה- לוקח מדי הרבה זמן לחשב כל פיקסל בנפרד.

פיתרון- נחשב ביחד כמות של פיקסלים על ידי ריבוי תהליכונים- כל תהליכון יחשב במקביל צבע של פיקסל.

מימוש:

יוצרים תהליכונים חדשים ומוסיפים לרשימה.

כל תהליך שמוסיפים לרשימה מקבל בלוק קוד שמוגדר בגוף הפונקציה:

כל עוד יש פיקסל שיש צורך לעבד אותו, נעשה castRay לאותו פיקסל.

```
else
{
    var threads = new LinkedList<Thread>(); // list of threads
    while (threadsCount-- > 0) // add appropriate number of threads
        threads.add(new Thread(() -> { // add a thread with its code
            PixelManager.Pixel pixel; // current pixel(row,col)
            // allocate pixel(row,col) in loop until there are no more pixels
            while ((pixel = pixelManager.nextPixel()) != null){

                // Cast a ray through the pixel
                castRay(nX, nY, pixel.col(), pixel.row());
                // System.out.println(pixel.col());
            }
        }));
    for (var thread : threads) thread.start();
    try { for (var thread : threads) thread.join(); } catch (InterruptedException ignore) {}
```

תוצאה:

בלי MT:

```
✓ IntegrationTests (renderer)      6 sec 323 ms
  ✓ AntiAliasingTestScene1()       6 sec 323 ms

✓ Tests passed: 1 of 1 test – 6 sec 323 ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...

Process finished with exit code 0
```

עם MT:

```
✓ IntegrationTests (renderer)      2 sec 973 ms
  ✓ AntiAliasingTestScene1()       2 sec 973 ms

✓ Tests passed: 1 of 1 test – 2 sec 973 ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...

Process finished with exit code 0
```

שיפור כפול 2.1

:BVH

תאור:

בעיה- יש הרבה פיקסלים שלכאורה לא צריך לחשב אותם ולבזבז זמן ריצה עליהם- כי הצבע שלהם הם צבע הרקע.

פיתרון- כל גוף, נשים בקופסא, וכל פעם שנבדוק חיתוך, נבדוק- אם הקרן חותכת קופסא – אז נבדוק חיתוך רגיל, חיתוך עם הגוף... ואם הקרן לא חותכת את הקופסא, אז נחזיר את צבע הרקע.

מימוש:

הוספנו מחלקה בשם Bounding Box.

מחלקה זו מייצגת תיבה AAB (Axis Aligned Bounding Box)

המחלקה משמשת לבדיקה האם קרן פוגעת בתיבה או לא.

```
public class BoundingBox {  5 usages  Lea Berdugo


    private double xMin = Double.POSITIVE_INFINITY;  8 usages

    private double yMin = Double.POSITIVE_INFINITY;  8 usages

    private double zMin = Double.POSITIVE_INFINITY;  8 usages

    private double xMax = Double.NEGATIVE_INFINITY;  8 usages

    private double yMax = Double.NEGATIVE_INFINITY;  8 usages

    private double zMax = Double.NEGATIVE_INFINITY;  8 usages
```

xMin,yMin,zMin הם הערכים המינימליים של התיבה בכל ציר.

xMax,yMax,zMax הם הערכים המקסימליים של התיבה בכל ציר.

הגדרנו בנאי, וכן גטרים לכל שדה.

וכן פונקציה intersectBV שמקבלת קרן ובודקת אם הקרן פוגעת בקופסא.

הוספנו מחלקה Container שיורשת מintersectable.

במחלקה נגדיר קופסאות לכל geometries וgeometry.

```
/**
 * class to contain the proper methods for setting a bounding box for both geometry and geometries
 */
public abstract class Container extends Intersectable {  15 usages  9 inheritors  Lea Berdugo *

    /**
     * every Intersectable composite have his bounding volume, which is represented by a bounding box
     */
    public BoundingBox boundingBox; // = null as default  19 usages
```

הגדרנו פונקציה findIntersectBoundingRegion שמקבל Ray ובודקת – אם אין לצורה קופסא או שהקרן חותכת את הקופסא, אז זה בודק את החיתוך של הקרן עם הצורות\צורה בתוך הקופסא.

```
/**
 * Finds the intersection points of a given ray with a bounding region defined by this object's bounding box.
 * If the bounding box is not defined or the ray intersects the bounding box, it retrieves the intersections
 * of the ray with the geo objects within the bounding region.
 *
 * @param ray The ray to test for intersections.
 * @param max The maximum distance along the ray to check for intersections.
 * @return A list of {@link GeoPoint} objects representing the intersection points, or {@code null}
 *         if there are no intersections or the bounding box is not intersected.
 */
public List<GeoPoint> findIntersectBoundingRegion(Ray ray, double max) {  2 usages  Lea Berdugo
    if (boundingBox == null || boundingBox.intersectBV(ray)) {
        return findGeoIntersections(ray, Double.POSITIVE_INFINITY, bb: true);
    }
    return null;
}
```

הוספנו פונקציה בשם BuildBvhTree שמבצעת בנייה אוטומטית של עץ BVH.

קודם כל, הפונקציה משטחת את הרשימה containers, כך שכל צורה ברשימה מוצגת בנפרד. אם יש אובייקטים מורכבים שמכילים כמה צורות (כמו קופסאות שבתוכן יש כמה צורות), הפונקציה מפרקת אותם לרשימה של צורות בודדות.

אחרי זה, הפונקציה בונה את העץ. היא משווה את המרחקים בין קופסאות הגבול של הצורות השונות, ומחפשת את הצורות הכי קרובות אחת לשנייה. היא מאחדת אותן לקופסא חדשה, וממשיכה בתהליך הזה עד שנשארת רק קופסא אחת שמכילה את כל הצורות.

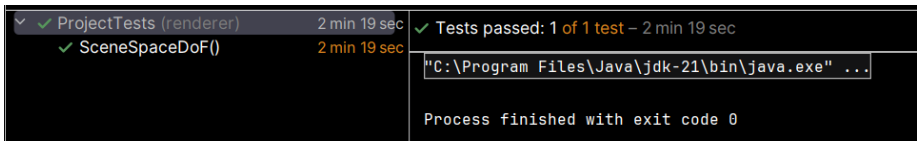
בסוף, הפונקציה מוסיפה את הקופסא החדשה לרשימת ה containers, ומסירה ממנה את הצורות שאוחדו לתוך הקופסא.

הפונקציה הזו עוזרת לסדר את הצורות בצורה חכמה יותר, מה שמקל על תהליך החישוב, כמו ב Ray tracing, ויכול לחסוך זמן כשבודקים אם קרן חותכת קופסא מסוימת.

```
public void BuildBvhTree() {  1 usage  Lea Berdugo *
    this.flatten();
    double distance;
    Container bestGeometry1 = null;
    Container bestGeometry2 = null;

    // while there are more than one container in the list
    while (containers.size() > 1) {
        // initialize the best distance to the maximum value
        double best = Double.MAX_VALUE;
        // for each geometry in the list
        for (Container geometry1 : containers) {
            // for each geometry in the list
            for (Container geometry2 : containers) {
                // if the geometries are not the same and the distance between them is less than the best distance
                distance = geometry1.boundingBox.BoundingBoxDistance(geometry2.boundingBox);
                if (!geometry1.equals(geometry2) && distance < best) {
                    best = distance;
                    bestGeometry1 = geometry1;
                    bestGeometry2 = geometry2;
                }
            }
        }

        if (bestGeometry1 != null && bestGeometry2 != null)
            containers.add(new Geometries(bestGeometry1, bestGeometry2));

        containers.remove(bestGeometry1);
        containers.remove(bestGeometry2);
    }
}
```

תוצאה:

בלי BVH:



עם BVH: