רקע לפרויקט

הפרויקט עוסק במימוש מנוע רינדור המבוסס על טכנולוגיית ריי-טרייסינג (Ray Tracing), שמאפשר יצירת סצנות תלת-ממדיות מציאותיות על ידי הדמיית האופן שבו קרני אור מתפשטות ומתנגשות באובייקטים במרחב. השיטה מחייבת חישובים מורכבים של האינטראקציות בין קרני אור לאובייקטים שונים, כמו השתקפות, שבירה, והצללה.

המערכת בנויה ממספר חבילות (packages) שמחלקות את האחריות והפונקציונליות בין תחומים שונים כמו ניהול סצנה, חישובים גיאומטריים, מעקב אחר קרניים, וניהול תאורה. המבנה המודולרי הזה מאפשר לפתח, לבדוק, ולתחזק את המערכת בקלות.

חבילת `geometries`

חבילה זו כוללת את כל המחלקות הקשורות לגופים הגיאומטריים המופיעים בסצנה. גופים אלו הם האלמנטים הפיזיים שעליהם הקרניים מתנגשות ומשפיעים על הצבעים והאורות המופיעים בתמונה הסופית.

מחלקות בחבילה:

- **`Geometry`**: מחלקת הבסיס עבור כל הגופים הגיאומטריים בפרויקט. מכילה פונקציות כלליות לחישוב נקודות חיתוך בין קרן לגוף גיאומטרי, ומגדירה תכונות גנריות כמו חומר וצבע.
- **`RadialGeometry`**: מחלקת בסיס לגופים גיאומטריים בעלי רדיוס (כגון כדור, גליל, צינור). יורשת מ-`Geometry` ומוסיפה טיפול בתכונות הקשורות לרדיוס.
- **`Sphere`**: מייצגת כדור במרחב התלת-ממדי. כוללת תכונות כמו רדיוס ומרכז, ופונקציות לחישוב חיתוכים בין קרן לכדור והחזרת נורמל.
- **`Plane`**: מייצגת מישור אינסופי. כוללת תכונות כמו נקודה על המישור ונורמל, ופונקציות לחישוב חיתוכים בין קרן למישור.
- **`Triangle`**: מייצגת משולש במרחב. כוללת שלוש נקודות שמגדירות את המשולש, ופונקציות לחישוב חיתוכים והחזרת נורמל.
- **`Polygon`**: מייצגת מצולע במרחב, המורכב ממספר נקודות כלשהן. המימוש תומך בניתוח חיתוכים ונורמל למצולע.
- **`Cylinder`**: מייצגת גליל תלת-ממדי בעל רדיוס וגובה. מחלקה זו יורשת מ-`RadialGeometry`.
- **`Tube`**: מייצגת צינור אינסופי במרחב התלת-ממדי. מחלקה זו יורשת מ-`RadialGeometry`.
- **`Geometries`**: מייצגת אוסף של גופים גיאומטריים. מאפשרת לטפל במספר גופים כאובייקט אחד ומקלה על חישוב חיתוכים בסצנה מרובת גופים.
- **`Intersectable`**: ממשק המתאר את היכולת לחשב נקודות חיתוך עם קרניים עבור גופים גיאומטריים.

חבילת `lighting`

חבילה זו כוללת את כל המחלקות הקשורות למקורות האור בסצנה. אור הוא מרכיב קריטי לרינדור תלת-ממדי, והוא משפיע על האופן שבו הקרניים יוצרות את התמונה הסופית.

מחלקות בחבילה:

- **LightSource**: ממשק המגדיר את הפונקציות הכלליות שמשותפות לכל סוגי מקורות האור, כמו חישוב עוצמת האור בנקודה מסוימת.
- **Light**: מחלקה אבסטרקטית המייצגת מקור אור כללי בסצנה. מכילה את עוצמת האור.
- **AmbientLight**: מייצגת אור מפוזר שמאיר את כל הסצנה באור אחיד, ללא כיוון מוגדר. משמש ליצירת רקע מואר ורך ללא מקור אור מובהק.
- **DirectionalLight**: מייצגת מקור אור כיווני שמפיץ אור בכיוון אחד ואינו תלוי במיקום. אידיאלי להדמיית אור שמש או כל מקור אור מרוחק.
- **PointLight**: מייצגת מקור אור נקודתי שמפיץ אור לכל הכיוונים מנקודה אחת במרחב. כוללת פונקציות לחישוב עוצמת האור וכיוון האור בנקודות שונות בסצנה.
- **SpotLight**: מייצגת מקור אור מרוכז שמפיץ אור בזווית צרה מנקודה אחת במרחב. מתאים להדמיית פנסים או כל מקור אור ממוקד.

חבילת primitives

חבילה זו כוללת את המחלקות שמייצגות מבנים מתמטיים בסיסיים כמו נקודות, וקטורים, קרניים, וצבעים. מבנים אלו הם הבסיס לכל החישובים הגיאומטריים והאופטיים בפרויקט.

מחלקות בחבילה:

- **Point**: מייצגת נקודה במרחב תלת-ממדי. כוללת פונקציות לחיבור, חיסור ומדידת מרחק בין נקודות.
- **Vector**: מייצגת וקטור במרחב תלת-ממדי. כוללת פונקציות לחיבור, כפל בסקלר, וכיוון הווקטור.
- **Ray**: מייצגת קרן, שהיא וקטור שמתחיל בנקודה מסוימת. הקרן משמשת למעקב אחר המסלול של אור מהמצלמה דרך הסצנה.
- **Color**: מייצגת צבע באמצעות שלושה מרכיבים (אדום, ירוק, כחול - RGB). כוללת פונקציות לחיבור צבעים, הכפלתם, ושינוי עוצמתם.
- **Material**: מייצגת את החומר שממנו עשויים הגופים הגיאומטריים. כוללת תכונות כמו שקיפות, החזרת אור, ושבירת אור.
- **Double3**: מייצגת וקטור בעל שלושה ערכים (משמשת לעוצמת האור או תכונות חומר).
- **Util**: מחלקת עזר הכוללת פונקציות סטטיות לתמיכה בחישובים גיאומטריים.

חבילת renderer

חבילה זו כוללת את המחלקות האחראיות על רינדור הסצנה והפקת התמונה הסופית. זהו החלק המרכזי במערכת שמטפל במעקב אחר קרניים, חישוב הצבעים, ויצירת התמונה.

מחלקות בחבילה:

- **Camera**: אחראית על יצירת הקרניים שיוצאים מנקודת המבט של המצלמה ועוברים דרך הפיקסלים במישור התצוגה. כוללת פונקציות לחישוב הקרניים ולביצוע רינדור של התמונה הסופית.
- **ImageWriter**: מתפקידה לשמור את התמונה הסופית כקובץ, ומכילה פונקציות לרישום הצבעים של הפיקסלים ושמירתם בפורמט המתאים.
- **RayTracerBase**: מחלקה אבסטרקטית המייצגת את מנגנון מעקב הקרניים לצורך רינדור הסצנה.
- **SimpleRayTracer**: מימוש בסיסי של מעקב קרניים לצורך חישוב צבעי הפיקסלים בסצנה.

- **`PixelManager`**: מחלקת עזר לניהול תהליכי מעקב הקרניים, בעיקר עבור פעולות של רינדור מקבילי.

חבילת `scene`

חבילה זו מכילה את המחלקות המרכזיות לניהול הסצנה התלת-ממדית. חבילה זו עוסקת בארגון האובייקטים השונים בסצנה ובקישור ביניהם לבין מנוע הרינדור.

מחלקות בחבילה:

- **`Scene`**: מנהלת את הסצנה כולה, כולל הגופים הגיאומטריים, מקורות האור, והרקע. כוללת פונקציות להוספה והסרה של אובייקטים ומקורות אור, וכן הגדרת המצלמה והרקע של הסצנה.

חבילת `xmlParser`

חבילה זו מכילה מחלקות לטיפול וקריאת קבצי XML לתיאור סצנות ואלמנטים.

מחלקות בחבילה:

- **`SceneXMLParser`**: מחלקה שאחראית על קריאת קבצי XML ליצירת סצנה מלאה, כולל הגדרת גופים גיאומטריים, מקורות אור, ומאפייני הסצנה.

חבילת `tests`

חבילת `tests` כוללת את כל הבדיקות שנערכו למערכת. כל מחלקה בפרויקט נבדקת כאן כדי לוודא שהפונקציות שלה מתפקדות כראוי ושהן תואמות את הדרישות.

מחלקות בחבילה- מחלקות בדיקה עבור כל מחלקה בפרויקט:

- כל מחלקה בפרויקט כוללת מחלקת בדיקה נפרדת שמוודאת את תקינות הפונקציות שלה.
- מחלקות אלו משתמשות ב-JUnit או כל כלי בדיקה אחר כדי להבטיח שהתנהגות המערכת עקבית, מדויקת, ונטולת באגים.

בונוסים

- שלב 2- נורמל לגליל סופי. 1
- שלב 3- חיתוך עם מצולע. 1
  חיתוך עם גליל אין סופי. 2
  חיתוך עם גליל סופי. 1
- שלב 4- טרנספורמציות מיקום וסיבוב של מצלמה. 1

- שלב 5- שימוש בקבצי xml להגדרת סצינה. 2
- שלב 7- תמונה עם 10+ עצמים. 1

צילום בונוס ראשון מזויות שונות. 2

מיניפ 1- שיפור בתמונה

תיאור החבילה:

הבעיה- בשיטה הנוכחית, כל פיקסל מקבל את הצבע שיש במרכז שלו, ללא התחשבות במה שקורה בשאר הפיקסל. לכן בתמונה שנוצרת יש מעברים חדים בין צבע של פיקסל אחד לצבע של הפיקסל לידו, מה שיוצר מראה מפוקסל.

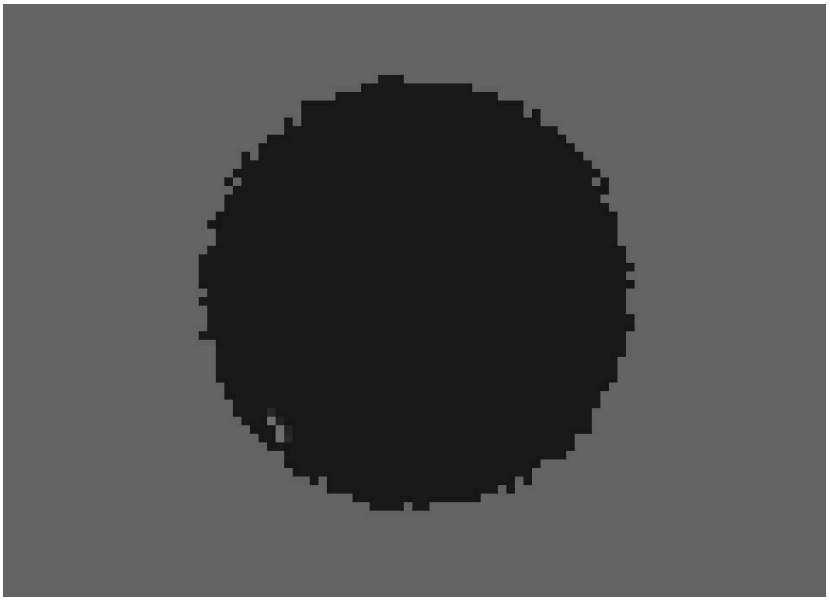
הפתרון- יצירת קרניים לנקודות נוספות בפיקסל מלבד המרכז, וחישוב ממוצע הצבעים בכל נקודה. ככה במעבר בין צבע לצבע, יהיו פיקסלים בצבע הבינוני, ונקבל מראה פחות מפוקסל.

השינויים שביצענו-

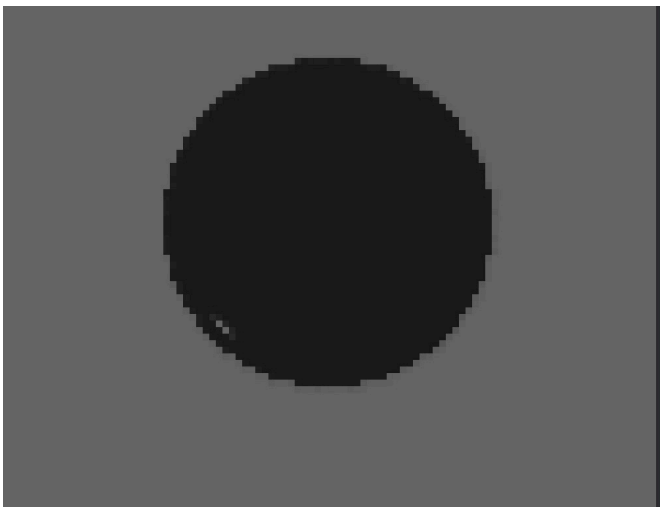
```
public Camera renderImage() {  31 usages  shira kernberg +1
    int nx = this.imageWriter.getNx();
    int ny = this.imageWriter.getNy();
    PixelManager pixel = new PixelManager(ny, nx, print);
    IntStream.range(0, ny).parallel().forEach(i -> IntStream.range(0, nx).parallel().forEach(j -> {
        this.imageWriter.writePixel(i,j,castRay(nx, ny,i, j));

        pixel.pixelDone();
    }));
    return this;
}
```

```java
private Color castRay(int nX, int nY, int j, int i) {  no usages  shira kernberg +1
    // Initialize the color sum to black.
    Color sum = new Color( r: 0,  g: 0,  b: 0);

    // Loop through each sample within the pixel.
    for (int k = 0; k < numOfRays * numOfRays; ++k) {
        // Calculate the X and Y offsets for the current sample.
        double x = j + (k % numOfRays + (Math.random() - 0.5)) / (double) numOfRays;
        double y = i + (k / numOfRays + (Math.random() - 0.5)) / (double) numOfRays;

        // Construct the ray for the current sample.
        Ray ray = constructRay(nX, nY,(int)x, (int)y);

        // Trace the ray and get its color.
        Color color = rayTracer.traceRay(ray);

        // Add the color to the sum.
        sum = sum.add(color);
    }

    // Calculate the average color for the pixel by scaling the sum.
    Color color = sum.scale( k: 1d / (numOfRays * numOfRays));

    // Write the average color to the pixel in the image.
    imageWriter.writePixel(j, i, color);

    return color;
}
```

לפני



אחרי



מיניפ 2- שיפור ביצועים

תיאור החבילה:

הבעיה- זמני הריצה היו ארוכים מאוד, במיוחד כשנדרשו חישובים רבים לפיקסלים רבים בתמונות ברזולוציה גבוהה. החישוב התבצע באופן סדרתי, כלומר, כל פיקסל חושב אחד אחרי השני, מה שגרם לכך שכל החישובים עבור כל הפיקסלים בוצעו ברצף ולא במקביל.

הפתרון- מימוש multithreading- מאפשר לפצל את חישוב הצבעים של הפיקסלים למספר תהליכים (threads) שעובדים במקביל. הדבר נעשה בעזרת `IntStream.range().parallel()`, שיוצר זרם של תהליכים שמחשבים את צבעי הפיקסלים בצורה מקבילית, במקום סדרתית.

השינויים שביצענו- במחלקת Camera - בעזרת שימוש ב-IntStream עם `parallel()` הצלחנו לחלק את החישובים של הפיקסלים על פני מספר תהליכים במקביל.

- **()**: זו פונקציה שמייצרת רצף של מספרים (למשל, כל מספרי השורות בתמונה). כך שרצף המספרים הזה מעובד ברצף (אחד אחרי השני).
- **parallel()**: כאשר מוסיפים את הפונקציה הזו ל-IntStream, כל מספר ברצף מטופל במקביל על ידי תהליך נפרד (Thread). כלומר, במקום ששורות התמונה יעובדו אחת אחרי השנייה, כמה שורות מעובדות בו זמנית.
- **שימוש בשני IntStream.parallel()**: ברגע שהשתמשנו ב-IntStream.parallel() פעמיים, גם עבור השורות וגם עבור העמודות של הפיקסלים, כל פיקסל יכול להיות מחושב במקביל לפיקסלים אחרים, מה שמנצל בצורה מיטבית את כל ליבות המעבד.

הדבר מאפשר לשפר את ביצועי המערכת פי כמה מבלי לשנות את הלוגיקה המרכזית של חישוב צבע הפיקסלים, אלא רק את הדרך בה החישובים מבוצעים.

ככה זמן הריצה התקצר בצורה משמעותית, והמערכת יכולה לנצל טוב יותר את הקוד הקיים.

הפתרון הזה משפר את היעילות ואת מהירות הביצועים, תוך שמירה על איכות התוצאה הסופית.

```java
public Camera renderImage() {  34 usages  ReutGoldshmid +1 *
    int nx = this.imageWriter.getNx();
    int ny = this.imageWriter.getNy();
    PixelManager pixel = new PixelManager(ny, nx, print);

    // ()IntStream.parallel באמצעות multithreading-שימוש ב
    IntStream.range(0, ny).parallel().forEach(row ->
            IntStream.range(0, nx).parallel().forEach(col -> {
                this.imageWriter.writePixel(col, row, castRay(nx, ny, col, row));
                pixel.pixelDone();
            })
    );

    return this;
}
```

זמן לפני שיפור:7.5 דק

זמן אחרי:5 דק