דוח מיני פרוייקט

מטרת הפרויקט: יצירת שיפורים בתמונות שנוצרות במנוע רינדור תלת-ממדי, באמצעות טכניקות של Soft Shadows, ובהמשך האצת הביצועים בעזרת תהליכונים מרובים (Multithreading) וטכניקת Boundary Volume Hierarchy.

חלק 1:

שיפור איכות בתמונה בעזרת Soft Shadows

הגדרת הבעיה:

במעקב קרניים סטנדרטי (Ray Tracing), חישוב הצללים מתבצע באמצעות קרן צל אחת לכל מקור אור. גישה זו יוצרת צללים חדים ובלתי־מציאותיים, שאינם משקפים את ההתנהגות הפיזיקלית של אור המוקרן ממשטח אור ולא מנקודה בודדת.

החידוש שלנו:

יישמנו הצללות רכות (Soft Shadows) באמצעות קרני צל מבוזרות. עבור כל מקור אור, במקום לבדוק כיוון אחד בלבד, אנו יוצרים מספר קרני צל מהנקודה הפוגעת לעבר מיקומים אקראיים על פני שטח האור. פעולה זו מדמה הסתרה חלקית של האור, מה שמוביל לשוליים רכים יותר של הצללים ולתחושת תאורה מציאותית יותר.

המערכת שוקלת את פגיעת הקרניים – תוך הבחנה בין קרניים שמוסתרות לאלו שאינן מוסתרות – כדי לקבוע את צבע הפיקסל הסופי.

חישוב עצמת הצללים הרכים:

להבדיל מהשיטה הקלאסית, בה נשלחת קרן צל בודדת לכל מקור אור, אנו מבצעים דגימה מרובת קרני צל באזור שטח האור.

בפונקציה calcLocalEffectsSoftShadows אנו מייצרים רשת של קרני צל היוצאות מנקודת המפגש אל נקודות שונות על פני שטח האור (לפי רדיוס האור והכיוון). לכל קרן אנו בודקים האם היא נחסמת על ידי גיאומטריה כלשהי בעזרת הפונקציה isBlocked.

כדי לייעל את תהליך הבדיקה ולצמצם את זמן החישוב, אנו משתמשים במבנה האצה מסוג **BVH (Bounding Volume Hierarchy)** – עץ היררכי של קופסאות תחומות, שמאפשר סינון מוקדם של גיאומטריות לא רלוונטיות. כך ניתן לבדוק חיתוך רק מול עצמים פוטנציאליים במקום לסרוק את כל הסצנה עבור כל קרן.

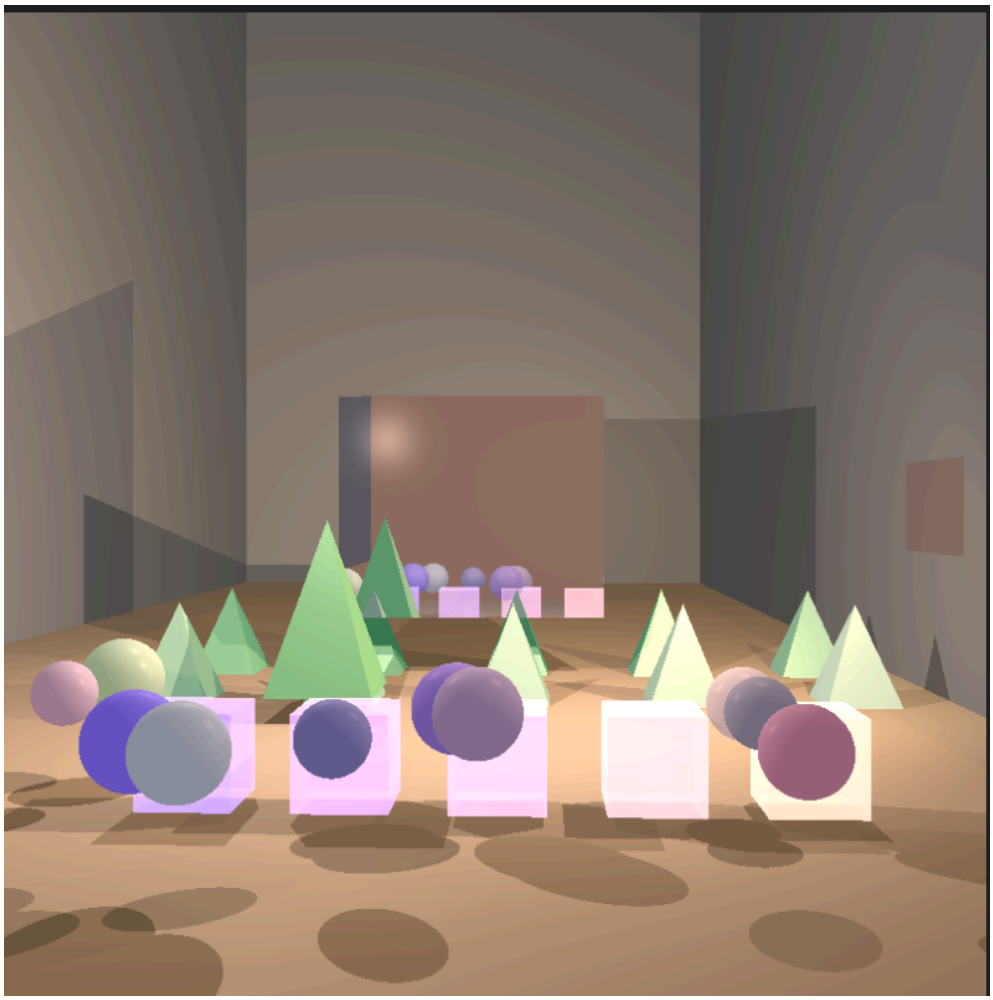
לאחר איסוף תוצאות הקרניים, אנו מחשבים את מקדם השקיפות הממוצע (ערך בין 0 ל־1), אשר מתאר את מידת העמימות של הצללים. ערך זה משמש כמשקל לקביעת עוצמת התאורה הסופית, ומאפשר יצירת הצללות רכות עם שוליים מטושטשים יותר – הקרובים להתנהגות האור הפיזיקלית ומעשירים את תחושת העומק והמציאות.

```java
/**
 * Determines whether a ray is blocked before reaching the light source.
 *
 * @param shadowRay the ray toward the light
 * @param lightDistance distance to light source
 * @return true if the ray is blocked
 */
private boolean isBlocked(Ray shadowRay, double lightDistance) {  1 usage  efratYi +1
    List<Intersection> intersections = scene.geometries.calculateIntersections(shadowRay);
    if (intersections == null) return false;

    for (Intersection i : intersections) {
        if (shadowRay.getHead().distance(i.point) < lightDistance - DELTA &&
                i.geometry.getMaterial().kT.lowerThan(MIN_CALC_COLOR_K)) {
            return true;
        }
    }
    return false;
}
```

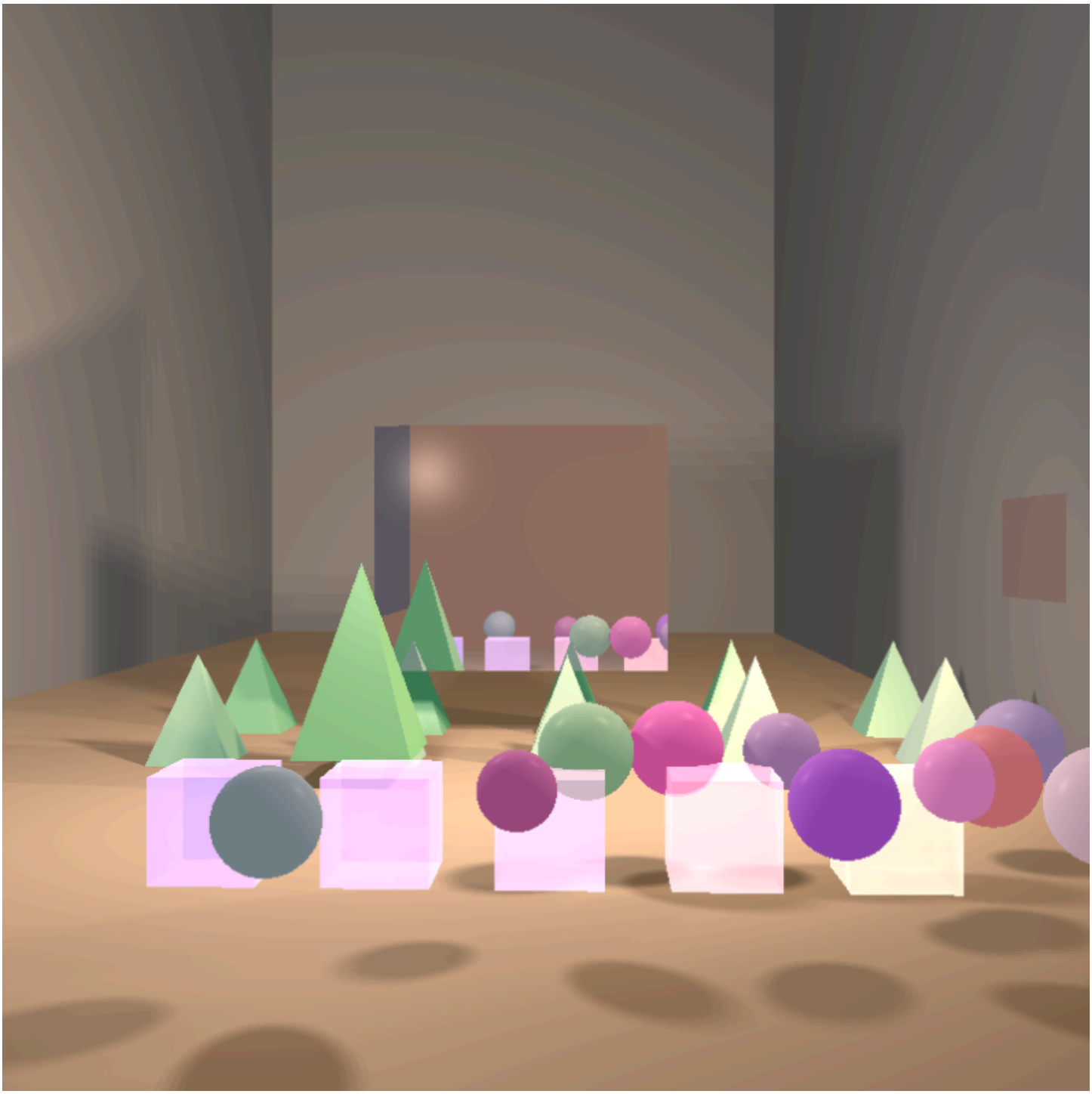
```java
/**
 * Calculates soft shadows using a sampling strategy.
 *
 * @param lightSource area light source
 * @param intersection the intersection
 * @return shadow intensity modifier
 */
private Double3 calcLocalEffectsSoftShadows(PointLight lightSource, Intersection intersection) {  1 usage  efratYi +1
    Vector l = lightSource.getL(intersection.point);
    Vector vUp;

    try {...} catch (Exception e) {...}

    int numSamples = 5;
    TargetArea area = new TargetArea(numSamples, size: lightSource.getRadius() * 2, l.scale(t: -1), vUp, lightSource.getPosition());
    double totalShadow = 0;
    int validRays = 0;

    for (int i = 0; i < numSamples; i++) {...}

    return validRays > 0 ? new Double3( value: 1 - (totalShadow / validRays)) : transparency(intersection);
}
```

השוואת התמונות:

תמונה לפני שיפור:



תמונה אחרי השיפור:



התוצאה:

הבדל ויזואלי:

צללים רכים מאפשרים קצוות צללים מטושטשים וטבעיים יותר במקום קווים חדים ונוקשים. ההטמעה באמצעות דגימת מספר קרניים מכל מקור אור מדמה את התפשטות האור על פני שטח, וכך מקבלת הצללה הדרגתית והדרגתית יותר.

השלכות ביצועים:

דגימה מרובת קרניים לכל מקור אור מגדילה את עלות החישוב, אך התוצאה הויזואלית המשופרת מצדיקה את ההוצאה החישובית.

מודולריות ונגישות:

הפעלת הצללים הרכים מתבצעת דרך פרמטר רדיוס האור – רדיוס אפס לא מפעיל את השיפור (הצללים קשים),

וככל שהרדיוס גדול יותר הצללים רכים יותר, מפעיל דגימה ליצירת צללים רכים. ניתן לשלב ולהתאים בקלות בפרמטרים השונים.

חלק 2:

הגדרת הבעיה – זמן רינדור ארוך

לאחר שיישמנו שיטות לשיפור איכות התמונה באמצעות Soft Shadows, זיהינו כי זמן הרינדור לכל תמונה עלה במידה משמעותית.
לדוגמה:
רינדור עם 16 קרניים לכל פיקסל (4x4) עשוי להאריך את זמן החישוב פי עשרה ואף יותר, בהשוואה לרינדור בסיסי.
בסצנות מורכבות הכוללות שקיפויות, החזרים ותאורות מרובות, כל קרן נוספת מגדילה משמעותית את העומס החישובי.
למעשה, איכות התמונה באה על חשבון ביצועים וזמן ריצה.

הפתרון – מעבר לרינדור מקבילי (Parallel Rendering)

כדי להתגבר על הבעיה, הטמענו מנגנון רינדור במקביל באמצעות תהליכונים (Threads).
יישמנו שלוש גישות עיקריות:

- **renderImageNoThreads()** – רינדור סידרתי, פיקסל אחר פיקסל (ברירת מחדל).
- **renderImageRawThreads()** – יצירת תהליכונים ידנית, כאשר ניתן להגדיר את מספר התהליכונים.
- **renderImageStream()** – שימוש ב-Java Parallel Streams לביצוע רינדור במקביל באופן אוטומטי.

שליטה במערכת

הפרמטר threadsCount מגדיר את אופן הרינדור:

- 0 → רינדור סידרתי רגיל
- -1 → שימוש ב-Parallel Streams
- כל ערך חיובי → יצירת מספר תהליכונים ידני לפי המספר שהוגדר

ארכיטקטורת המערכת

PixelManager – מחלקת עזר לניהול חלוקת העבודה בין התהליכונים:

- מנהלת את חלוקת הפיקסלים לעיבוד בצורה בטוחה
- מוודאת שתהליכונים שונים לא יעבדו על אותו פיקסל בו זמנית
- מסנכרנת בין התהליכונים לשמירת יציבות ודיוק בתהליך

מנגנון renderImageRawThreads()

- יוצרת רשימת תהליכונים (Threads)
- כל Thread מקבל תור לעיבוד פיקסלים דרך PixelManager ופועל באופן עצמאי

- נעשה שימוש ב-thread.join() על מנת להמתין לסיום כל התהליכונים לפני המשך הרינדור

```java
/**
 * Render image using multi-threading by creating and running raw threads* @return the camera object itself
 */
private Camera renderImageRawThreads() {  1 usage  ruchamabricker
    var threads = new LinkedList<Thread>();
    while (threadsCount-- > 0)
        threads.add(new Thread(() -> {
            PixelManager.Pixel pixel;
            while ((pixel = pixelManager.nextPixel()) != null)
                castRay(pixel.col(), pixel.row());
        }));
    for (var thread : threads) thread.start();
    try {
        for (var thread : threads) thread.join();
    } catch (InterruptedException ignore) {}
    return this;
}
```

renderImageStream():

מממש רינדור מקבילי על ידי שימוש ב־IntStream.range(...).parallel() שמאפשר לעבד במקביל את כל שורות ועמודות הפיקסלים.

היתרון בשיטה זו הוא פשטות וקומפקטיות הקוד.

החיסרון הוא שליטה מוגבלת על אופן תזמון התהליכונים והיכולת לעקוב אחריהם באופן מדויק.

```java
/**
 * Render image using multi-threading by creating and running raw threads* @return the camera object itself
 */
public Camera renderImageStream() {  1 usage  ruchamabricker
    IntStream.range(0, nY).parallel() //
            .forEach( int i -> IntStream.range(0, nX).parallel() //
                    .forEach( int j -> castRay(j, i)));
    return this;
}
```

שדרוגים ב־Builder – ניהול ביצועים חכם

כדי לאפשר למשתמש לשלוט באופן גמיש בשיטת הרינדור, הוספנו ב־Builder של המצלמה אפשרות להפעיל רינדור במקביל באמצעות מספר תהליכונים שנקבע על ידי המשתמש.

פונקציה: setMultithreading(int threads)

```java
public Builder setMultithreading(int threads) {  5 usages  ruchamabricker
    if (threads < -2) throw new IllegalArgumentException("Multithreading must be -2 or higher");
    if (threads >= -1) camera.threadsCount = threads;
    else { // == -2
        int cores = Runtime.getRuntime().availableProcessors() - SPARE_THREADS;
        camera.threadsCount = cores <= 2 ? 1 : cores;
    }
    return this;
}
```

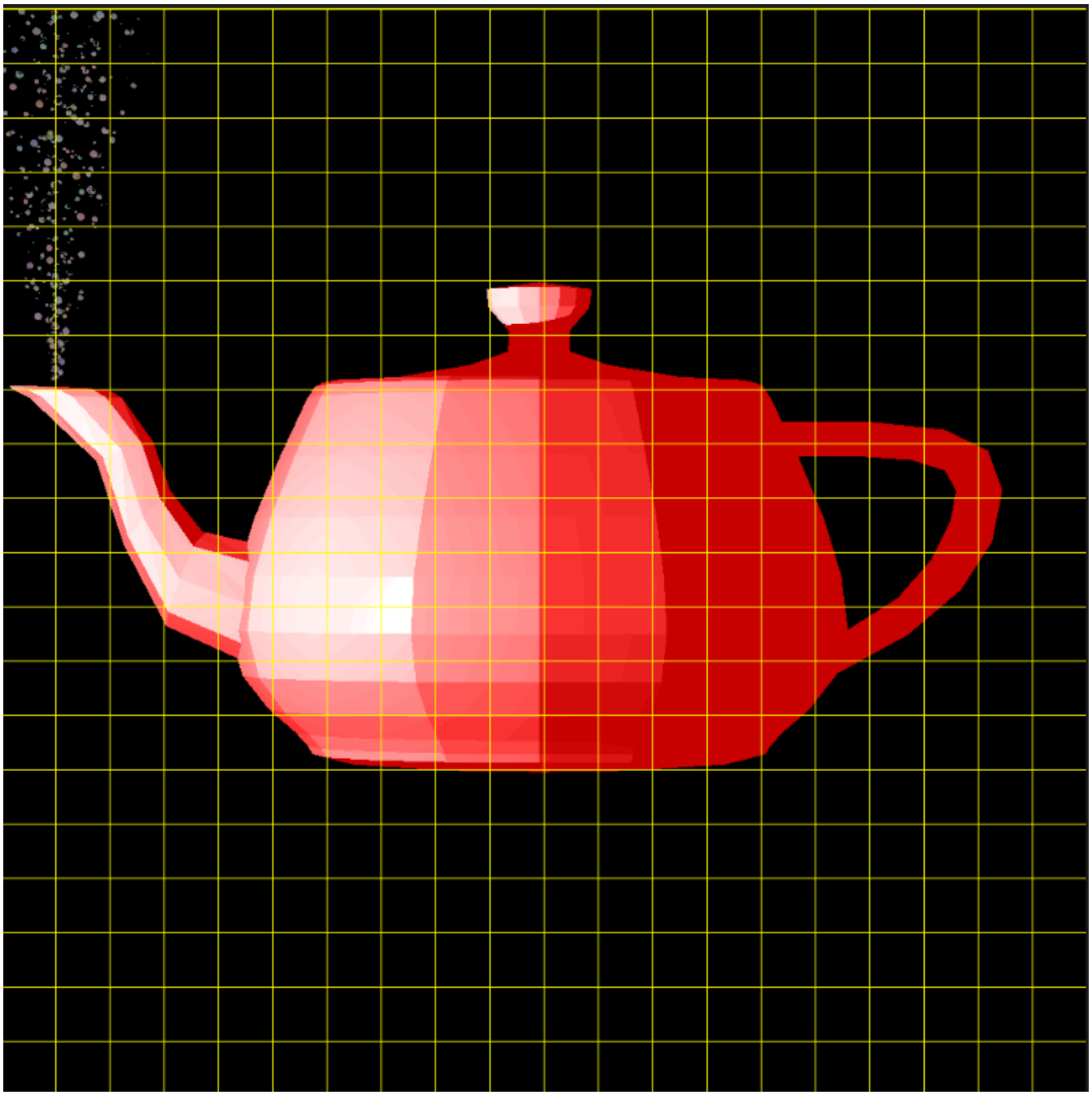
פונקציה: setDebugPrint(double interval)

מאפשרת להציג עדכוני התקדמות בזמן הרינדור ישירות בקונסולה.

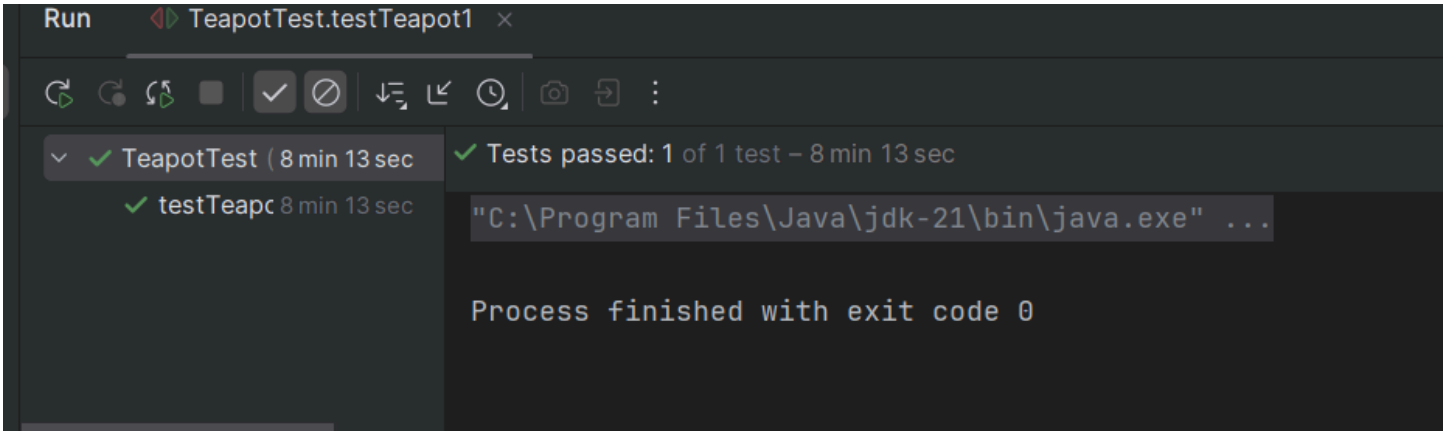
לדוגמה, ערך של 0.5 יגרום להדפסת אחוזי ההתקדמות כל חצי שנייה.
פונקציה זו שימושית במיוחד בעת דיבוג סצנות מורכבות, כדי לוודא שהרינדור אכן מתקדם בצורה תקינה.

```
/**
 * Sets the interval for printing progress percentage.
 *
 * @param interval the interval in seconds
 * @return this Builder instance
 * @throws IllegalArgumentException if the interval is negative
 */
public Builder setDebugPrint(double interval) {  4 usages  ruchamabricker
    if (interval < 0) throw new IllegalArgumentException("Interval value must be non-negative");
    camera.printInterval = interval;
    return this;
}
```

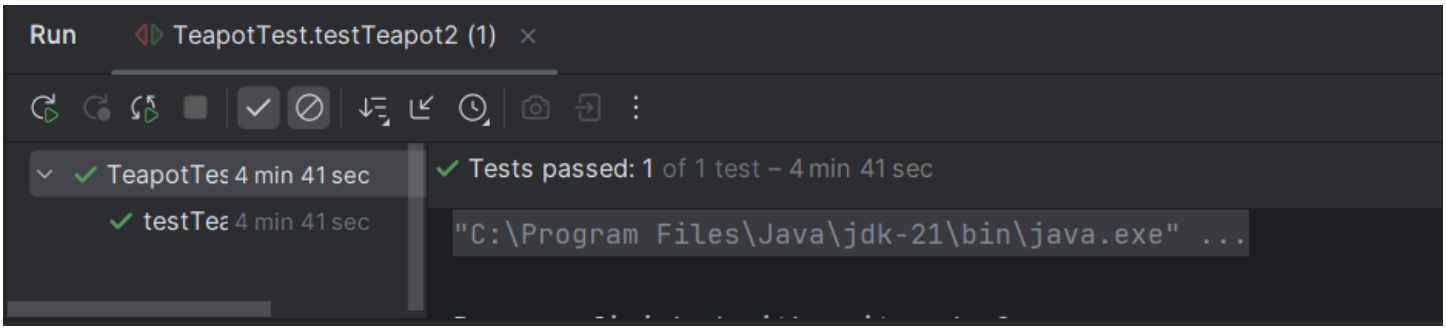
תמונות מרובות אובייקטים, הבדלים בזמן ריצה:



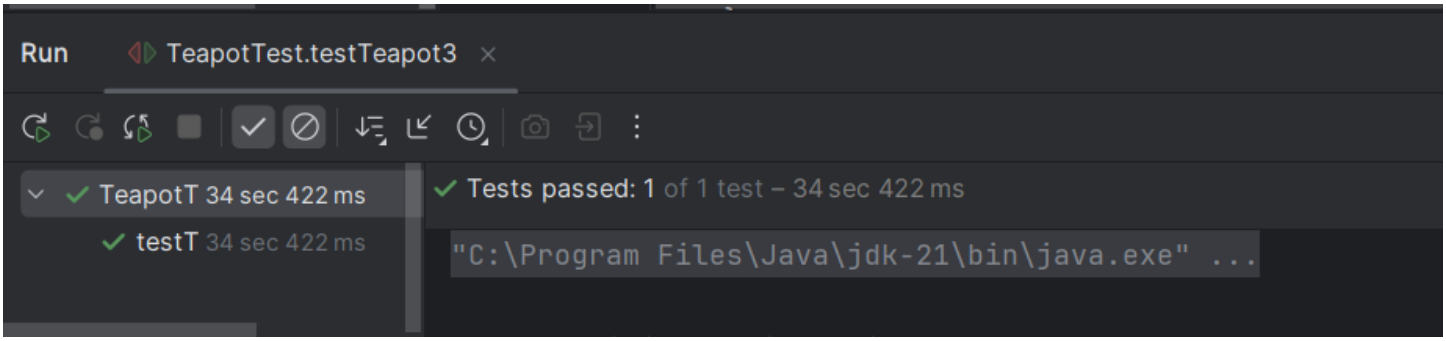
ללא האצות כלל:



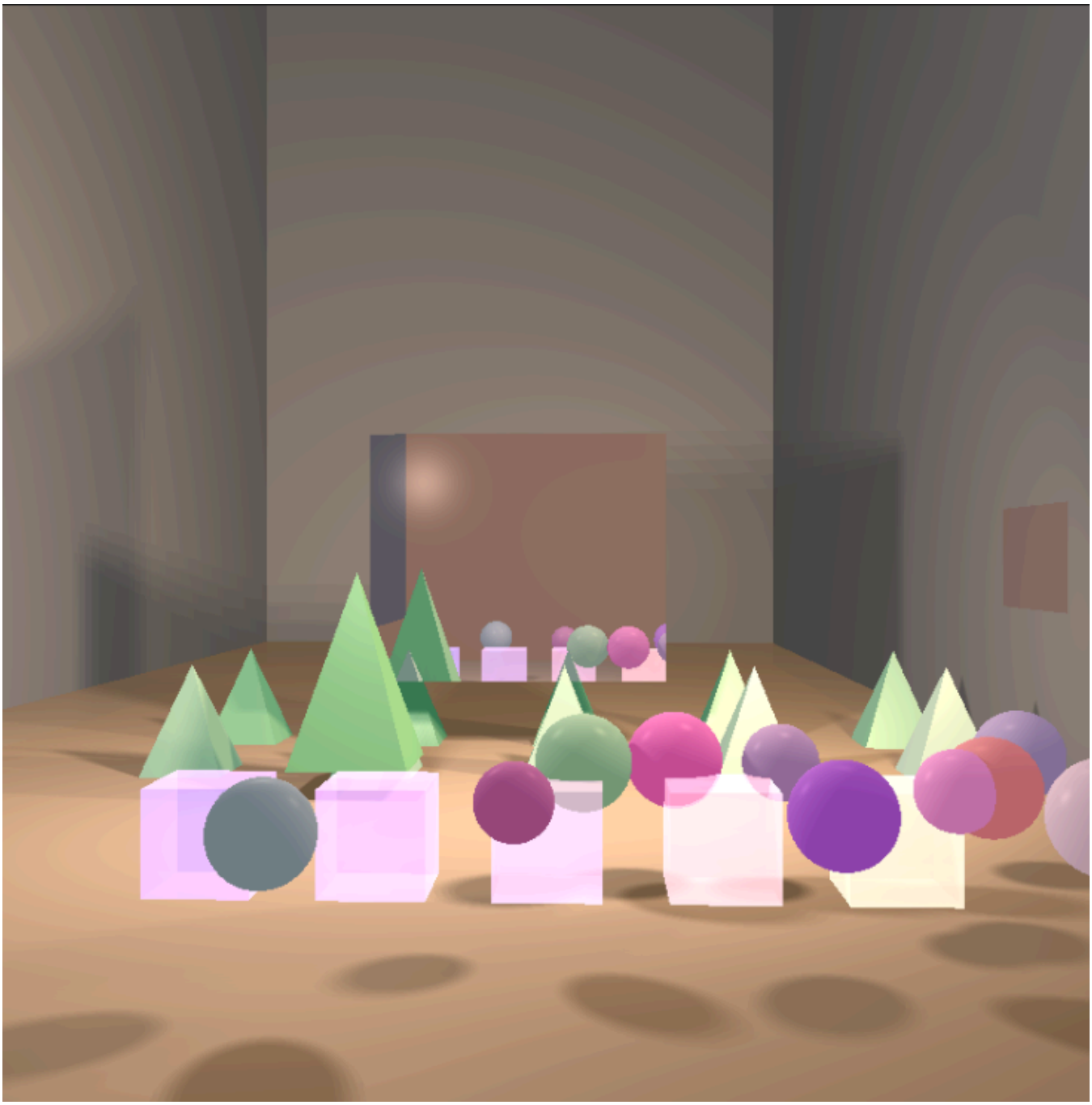
עם הצאת CBR:



עם האצת BVH:



תמונה נוספת:



בלי האצה:

עם האצת תהליכונים:

```
Run    Minip2.createSceneWithMultipleGeometriesSoftShadows_...
Minip2 (renderer)                4 min 3 sec
  createSceneWithMulti 4 min 3 sec
Tests passed: 1 of 1 test – 4 min 3 sec
C:\Users\molev\.jdks\openjdk-21.0.2\bin\java.exe ...
100.0%
Process finished with exit code 0
```

עם האצת BVH:

```
Run    Minip2.createSceneWithMultipleGeometriesSoftShadows...
Minip2 (renderer)                2 min 45 sec
  createSceneWithMulti 2 min 45 sec
Tests passed: 1 of 1 test – 2 min 45 sec
C:\Users\molev\.jdks\openjdk-21.0.2\bin\java.exe ...

Process finished with exit code 0
```

עם האצת BVH + תהליכונים מרובים:

```
Run    Minip2.createSceneWithMultipleGeometriesSoftShadows_...
Minip2 (renderer)                42 sec 746 ms
  createSceneWithMu 42 sec 746 ms
Tests passed: 1 of 1 test – 42 sec 746 ms
C:\Users\molev\.jdks\openjdk-21.0.2\bin\java.exe ...
100.0%
Process finished with exit code 0
```

עם האצת CBT + תהליכונים מרובים:

```
Run    Minip2.createSceneWithMultipleGeometriesSoftShadows_...
Minip2 (renderer)                50 sec 117 ms
  createSceneWithMul 50 sec 117 ms
Tests passed: 1 of 1 test – 50 sec 117 ms
C:\Users\molev\.jdks\openjdk-21.0.2\bin\java.exe ...
100.0%
Process finished with exit code 0
```