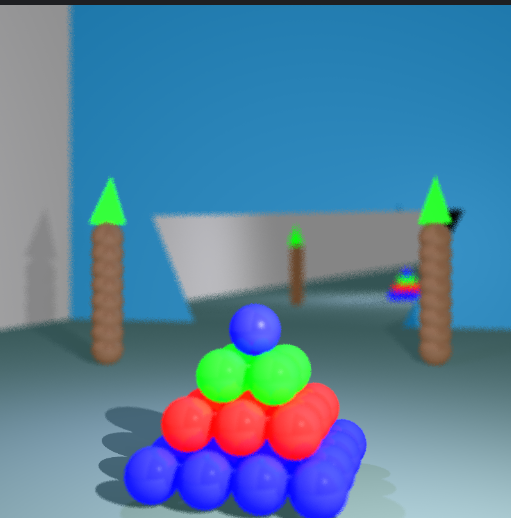
**דו"ח שיפורים**

בס"ד

**מיני פרויקט במבוא להנדסת תוכנה**

****

**מרצה:**

אליעזר גנסבורגר

**מגישים:**

אוהד כחלון 209530500

מאיר רביבו 208700831

עומק השדה - Depth of field3

מימוש הפונקציה3

יצירת תמונה5

**פונקציית הזזת המצלמה** **6**

**Multi-threading7**

**(BVH) Boundary Volume Hierarchy9**

מימוש 9

**דוגמאות לזמני ריצה וסיכום** **12**

**עומק השדה - Depth of field**

**הבעיה:**

בכל התמונות שיצרנו עד עכשיו, כל הפיקסלים בתמונה היו חדים באותה מידה, מה שגרם לתמונה להיראות לא מציאותית. בחיים האמיתיים, כאשר מצלמה מתמקדת על אובייקט מסוים, האובייקטים הקרובים והרחוקים יותר ממנו יוצאים מטושטשים. ללא אפקט עומק השדה (DoF), התמונות שלנו נראות שטוחות וחסרות עומק, ללא הפרדת מיקוד בין האובייקטים השונים.

**הפתרון:**

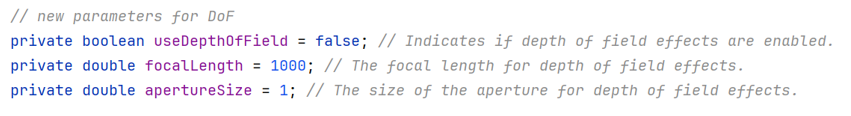
1. הפעלת אפקט עומק השדה:

כדי להתמודד עם הבעיה וליצור תמונות מציאותיות יותר, הוספנו למחלקת Camera את אפשרות הפעלת אפקט עומק השדה -.**useDepthOfField** כאשר אפשרות זו מופעלת, המצלמה תדמה את הטשטוש הנוצר בעדשות אמיתיות על ידי פיזור הקרניים דרך מפתח הצמצם.

בנוסף, הוספנו למחלקת המצלמה עוד 2 תכונות נוספות:

**focalLength** - מגדיר את המרחק מהמצלמה לנקודת המוקד שבה האובייקט נמצא בפוקוס. אורך המוקד קובע איזה חלק מהתמונה יהיה חד וברור, וכל מה שנמצא במרחק שונה מאורך המוקד יהיה מטושטש יותר.

**apertureSize** – גודל הצמצם. מגדיר את גודל הפתח דרכו הקרניים נכנסות למצלמה, ומשפיע על מידת הטשטוש של האובייקטים שמחוץ לפוקוס. צמצם גדול יותר יגרום לטשטוש גדול יותר מחוץ למוקד, וצמצם קטן יותר יגרום לפחות טשטוש.

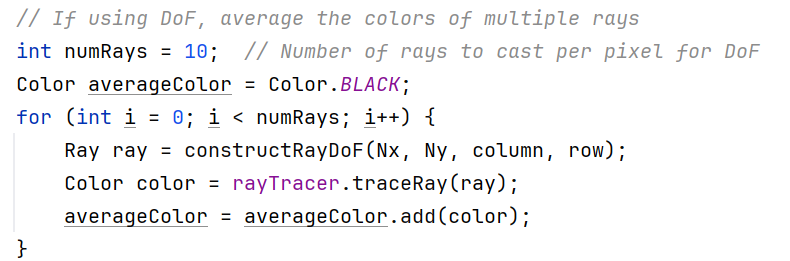


1. בניית קרן ראשית:

בשלב הראשון, אנו בונים קרן ראשית מכל פיקסל בעזרת המתודה constructRay , שמחשבת את הקרן הישירה מהמיקום הנוכחי של המצלמה לעבר הפיקסל במישור המוקד. הקרן הראשית נוצרת מהמיקום הנוכחי של המצלמה לפיקסל מסוים על מישור התמונה. פונקציה זו מחזירה קרן שעוברת דרך אותו פיקסל.

1. חישוב נקודת המוקד:

לפונקציה castRay הוספנו בדיקה האם משתמשים בDoF או לא. אם משתמשים בDoF, הפונקציה ממשיכה לתהליך יצירת קרניים נוספות.



מספר הקרניים נקבע על ידי המשתנה numRays (במקרה שלנו הוא 10).

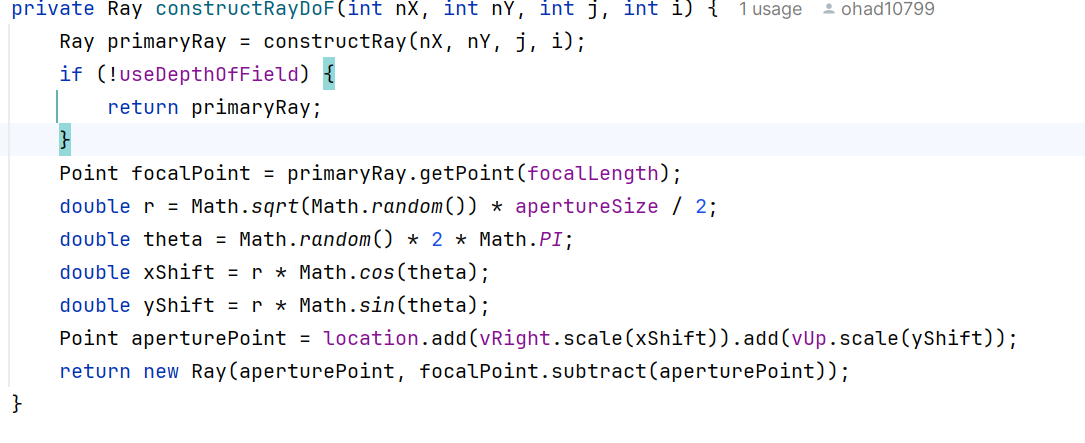
הפונקציה יוצרת צבע ממוצע לכל הקרניים שנוצרו. לכל קרן שנוצרה, היא מחשבת את הצבע בעזרת rayTracer ומוסיפה אותו לצבע ממוצע.

1. יצירת נקודה אקראית במפתח הצמצם:

כדי לדמות את אפקט הצמצם של עדשת מצלמה אמיתית, אנו יוצרים נקודה אקראית בתוך מפתח הצמצם. גודל הצמצם מוגדר על ידי.apertureSize הנקודה האקראית נבחרת על פי פיזור רדיאלי בתוך הצמצם.

1. בניית אלומה של קרניים עם אפקט עומק השדה:

השלבים האלה מתבצעים בתוך הפונקציה ,constructRayDoFשבה בונים קרן חדשה מנקודה אקראית בצמצם לנקודת המוקד. תהליך זה מתבצע מספר פעמים לכל פיקסל, לדוגמה 10 פעמים. הקרניים שנוצרות בכל פעם מתפזרות מעט זו מזו, ומייצגות את האור שנכנס דרך מפתח הצמצם.



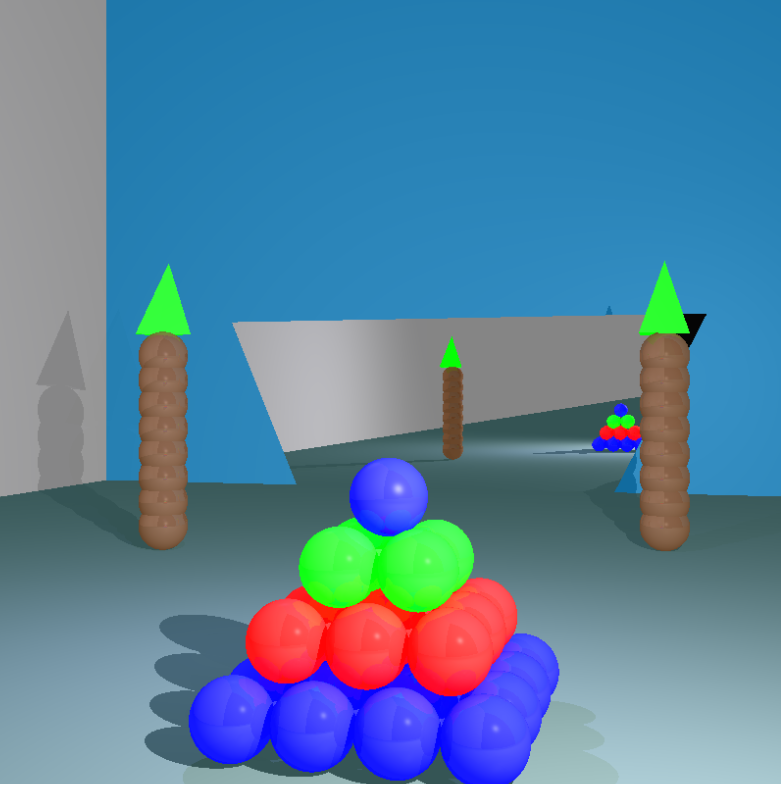
1. מיצוע הצבעים:

כל קרן מחושבת בעזרת המתודה traceRay כדי לקבל את הצבע שלה. לאחר שיצרנו מספר קרניים מכל פיקסל, אנו מחשבים את ממוצע הצבעים שהתקבלו מכל הקרניים כדי לקבל צבע סופי לכל פיקסל. ממוצע הצבעים מדמה את הטשטוש המתקבל בעדשה אמיתית.

1. יצירת התמונה:

בסיום התהליך, התמונה עם אפקט עומק השדה נכתבת לקובץ. התוצאה הסופית היא תמונה עם עומק ומיקוד מציאותיים יותר, המדמה את האופן שבו עין אנושית רואה את העולם.

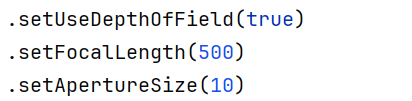
**תמונה שמכילה צעצוע

התיאור נוצר באופן אוטומטי**

**התמונה עם השיפור**

**התמונה ללא השיפור**

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

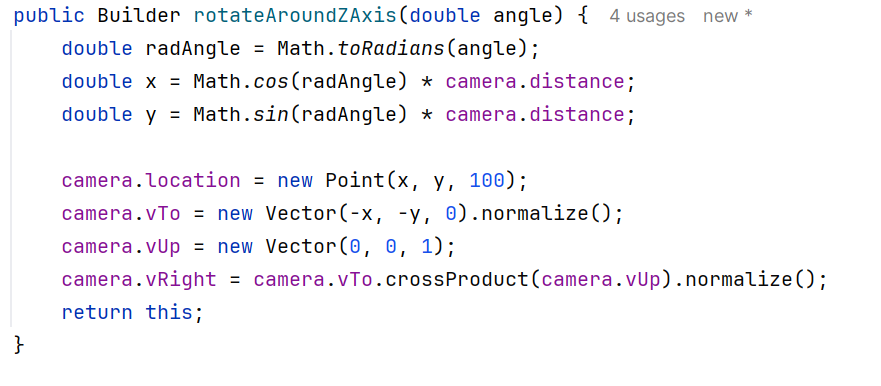
* הפעלה של DoF על ידי שינוי של הפרמטר setUseDepthOfField לtrue.
* קביעת אורך המוקד (פוקוס) setFocalLength.
* קביעת גודל הצמצם setApertureSize.

בתמונה שלנו:

תוספת של פונקציית הזזה:

למחלקת Camera הוספנו את הפונקציה rotateAroundZAxis , שאחראית לסובב את המצלמה סביב ציר ה-Z בזווית נתונה. הפונקציה מקבלת את הזווית במעלות וממירה אותה לרדיאנים. לאחר מכן היא מחשבת את הקואורדינטות החדשות של המצלמה במישור ה.XY מיקום המצלמה החדש מחושב באמצעות קוסינוס וסינוס של הזווית, ומכפלה במרחק הקבוע של המצלמה מהאובייקטים.

כמו פונקציית ההזזה של ציר הZ, הוספנו פונקציות דומות לציר הX והY.



**Multi-threading**

**הבעיה:**

עד עכשיו לא ניצלנו את יכולות המעבד עד הסוף מכיוון שהתהליך של הרינדור עובר על כל הפיקסלים באופן סדרתי, כלומר כל פיקסל מעובד אחד אחרי השני. תהליך זה יכול להיות איטי במיוחד כאשר יש הרבה פיקסלים, ואנחנו לא מנצלים את כל היכולות של המעבד.

**הפתרון:**

כדי לשפר את מהירות הרינדור, נפריד את עיבוד הפיקסלים למספר תהליכונים בעזרת .Multi-threading נשתמש במחלקה חדשה בשם ,PixelManager שתפקידה לנהל את תהליך הקצאת הפיקסלים לעיבוד. כל תהליכון יקבל פיקסל לעיבוד ויעבוד עליו במקביל לתהליכונים אחרים, מה שיאפשר ניצול מלא יותר של המעבד.

במחלקת Camera נממש את Multi-threading באמצעות הקוד הבא:

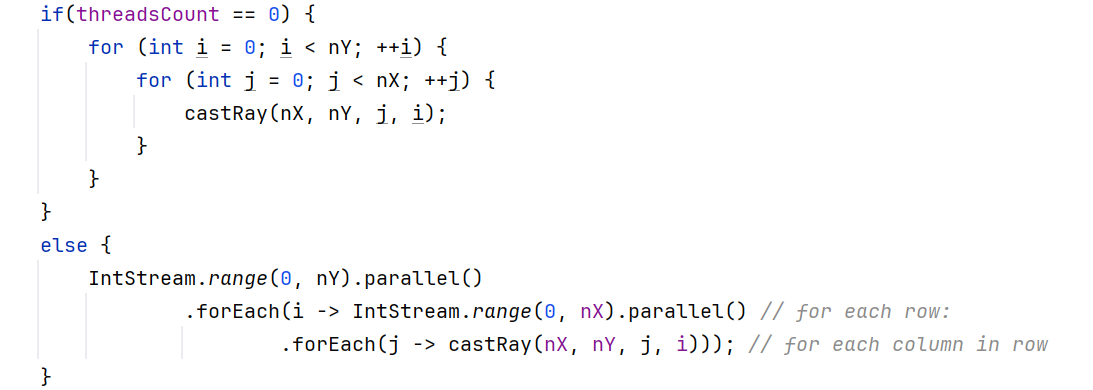
תמונה שמכילה טקסט, גופן, קו, צילום מסך

התיאור נוצר באופן אוטומטי

נבדוק את מספר התהליכונים שהמשתמש הגדיר:

* + אם מספר התהליכונים קטן מ-1 (למשל 0): נבצע את הרינדור באופן סדרתי, כמו שהיה קודם.
  + אם מספר התהליכונים גדול מ-1: נשתמש ב- IntStream.parallel() , כדי לבצע את עיבוד הפיקסלים במקביל, תוך כדי שימוש ב PixelManager לניהול הקצאת הפיקסלים ותיאום התהליכונים.
  + אם המשתמש הגדיר את מספר התהליכונים כמינוס 2, המערכת תבחר את מספר התהליכונים בצורה אוטומטית על פי המערכת, כדי לוודא שהעיבוד יתבצע בצורה מיטבית ללא עומס יתר על המערכת.

נגדיר את השימוש ב Multi-threading בפונקציה renderImage:



עם ריבוי תהליכונים

ללא ריבוי תהליכונים

השימוש ב-Multi-threading מאפשר לרינדור להתרחש מהר יותר מכיוון שהמעבד יכול לעבד מספר פיקסלים בו-זמנית במקום לעבד כל פיקסל אחד אחרי השני.

**:(BVH) Boundary Volume Hierarchy**

**הבעיה:**

בגרפיקה ממוחשבת, כאשר יש הרבה אובייקטים בסצנה, נדרש לבדוק לכל קרן האם היא חותכת כל אחד מהאובייקטים. אם יש אלפי אובייקטים, כמו במקרה של יער עם הרבה עצים, זה יכול להוביל למספר עצום של בדיקות, מה שמאט את זמן החישוב.

**הפתרון:**

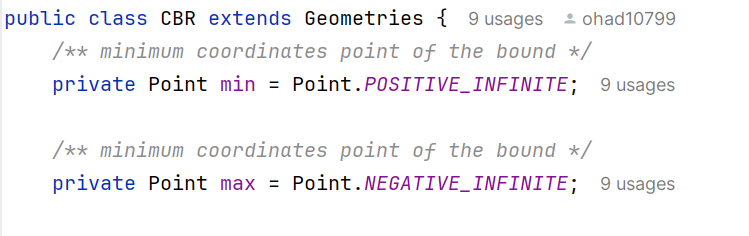
כדי להפחית את מספר הבדיקות הנדרש, משתמשים בBVH.

BVHהוא מבנה נתונים המארגן את האובייקטים בסצנה בהיררכיה של תיבות סגורות. במקום לבדוק כל אובייקט בנפרד, קודם בודקים האם הקרן חותכת תיבה סגורה גדולה שמקיפה קבוצת אובייקטים. אם כן, בודקים את תיבות המשנה שמכילות חלקים קטנים יותר, וכך הלאה. אם אין חיתוך עם התיבה הסגורה, ניתן לדלג על כל האובייקטים שבתוכה.

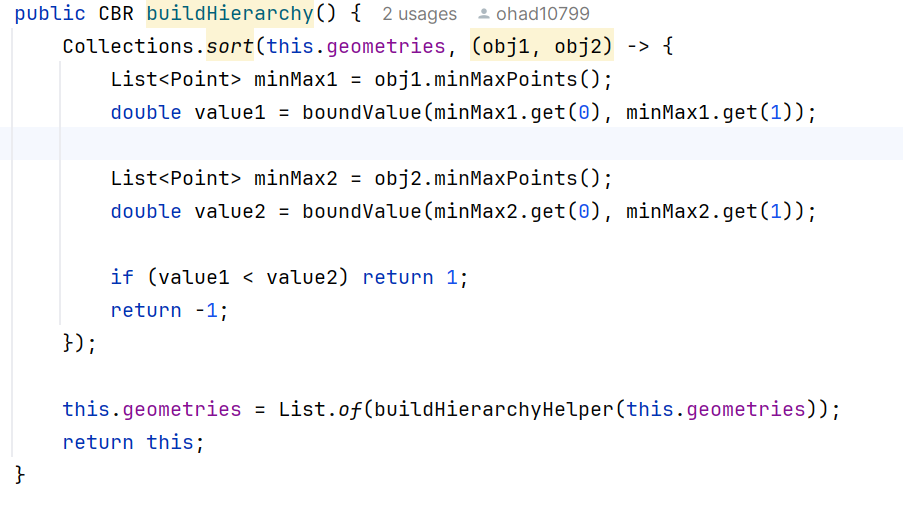
**דרך המימוש:**

הוספנו מחלקה בשם CBR. במחלקה מימשנו מנגנון ליצירת עץ BVH.

**מימוש תיבות סגורות (Bounding Boxes):** במחלקה CBR, כל קבוצה של אובייקטים מוקפת בתיבה סגורה המוגדרת על ידי שתי נקודות: min (הנקודה המינימלית) ו-max (הנקודה המקסימלית). נקודות אלו מייצגות את הקצוות של התיבה שמקיפה את כל האובייקטים בקבוצה.



**בניית היררכיית BVH:** מימשנו פונקציה בשם buildHierarchy, אשר מחלקת את האובייקטים לתתי-קבוצות ומקיפה כל תת-קבוצה בתיבה סגורה משלה. כך נבנית היררכיית תיבות סגורות, בה כל תיבה ברמה גבוהה יותר בהיררכיה מקיפה תיבות קטנות יותר.

תמונה שמכילה טקסט, צילום מסך, גופן, מספר

התיאור נוצר באופן אוטומטי

תחילה, הפונקציה ממיינת את רשימת האובייקטים הגיאומטריים לפי ערך שנקבע על ידי פונקציית boundValue.

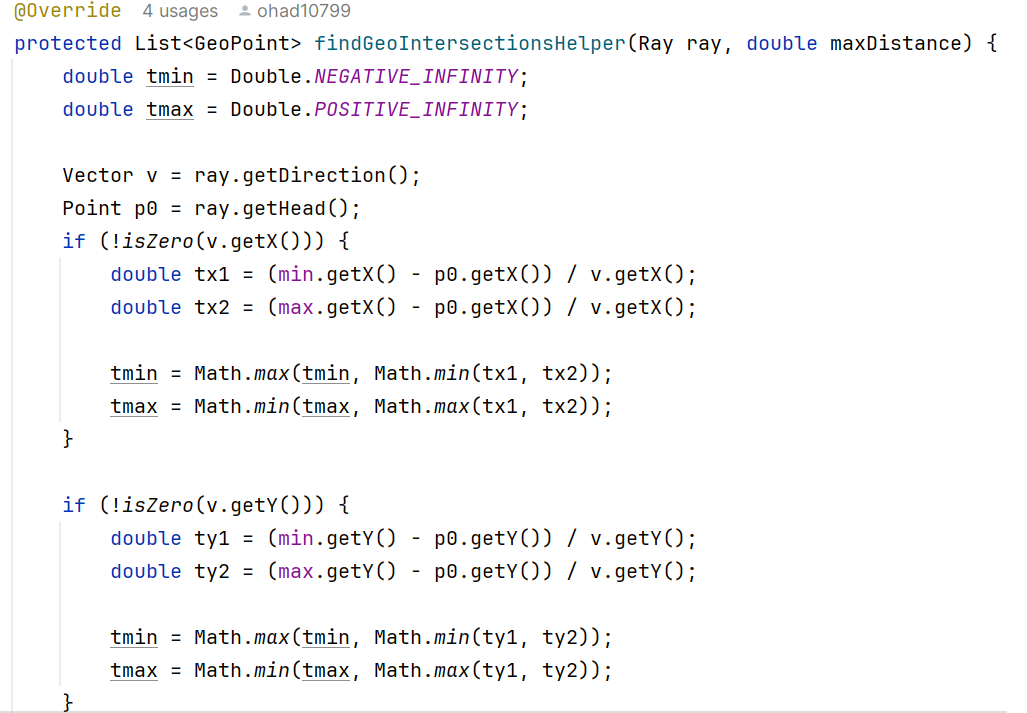
boundValue מחשבת את הסכום של קואורדינטות מרכז התיבה הסגורה שמקיפה כל אובייקט.

המיון מבוצע כך שהאובייקטים עם ערך מרכזי גבוה יותר (קרוב יותר למקור) יופיעו תחילה.

לאחר המיון, הפונקציה מחלקת את הרשימה הממוינת לשני חלקים שווים, ולאחר מכן יוצרת עץ BVH באופן רקורסיבי.

הפונקציה buildHierarchyHelper אחראית על הבנייה הרקורסיבית של העץ.

**בדיקות חיתוך:** מימשנו את הפונקציה findGeoIntersectionsHelper, שבודקת האם קרן מסוימת חותכת את התיבה הסגורה. אם הקרן חותכת את התיבה, נמשיך לבדוק תיבות פנימיות יותר בהיררכיה. אם לא, ניתן לדלג על כל האובייקטים שבתוך התיבה, מה שחוסך הרבה בדיקות חיתוך מיותרות**.**

****

**תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי**

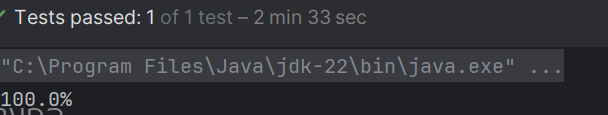
כדי לבדוק את השיפורים, יצרנו תמונה שלוקחת את העץ מהתמונה הראשונה, ומשכפלת אותו 200 פעמים. כתוצאה מכך, נוצרת תמונה שמכילה מאות גופים, ושזמן היצירה שלה ארוך.

יצירת התמונה ללא שימוש בתהליכונים:

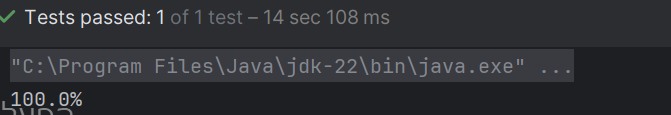
תמונה שמכילה טקסט, גופן, צילום מסך

התיאור נוצר באופן אוטומטי

יצירת התמונה עם שימוש בתהליכונים:



יצירת תמונה הכוללת ריבוי תהליכונים וגם שימוש בBVH:



לסיכום, הוספת תהליכונים ושיפורי יעילות של BVH משפרים משמעותית את זמן יצירת התמונה והופכים את התהליך למהיר יותר, יעיל יותר וחסכוני יותר.