

מיני פרוייקט מבוא להנדסת תוכנה

151055.5780



**מגישות:**

**סבאג תמר 855054**

**שושן גאולה 11826658**

**יוני 2020 | תמוז תש"פ**

**מיני פרוייקט 1 +2**

מיני פרוייקט 1

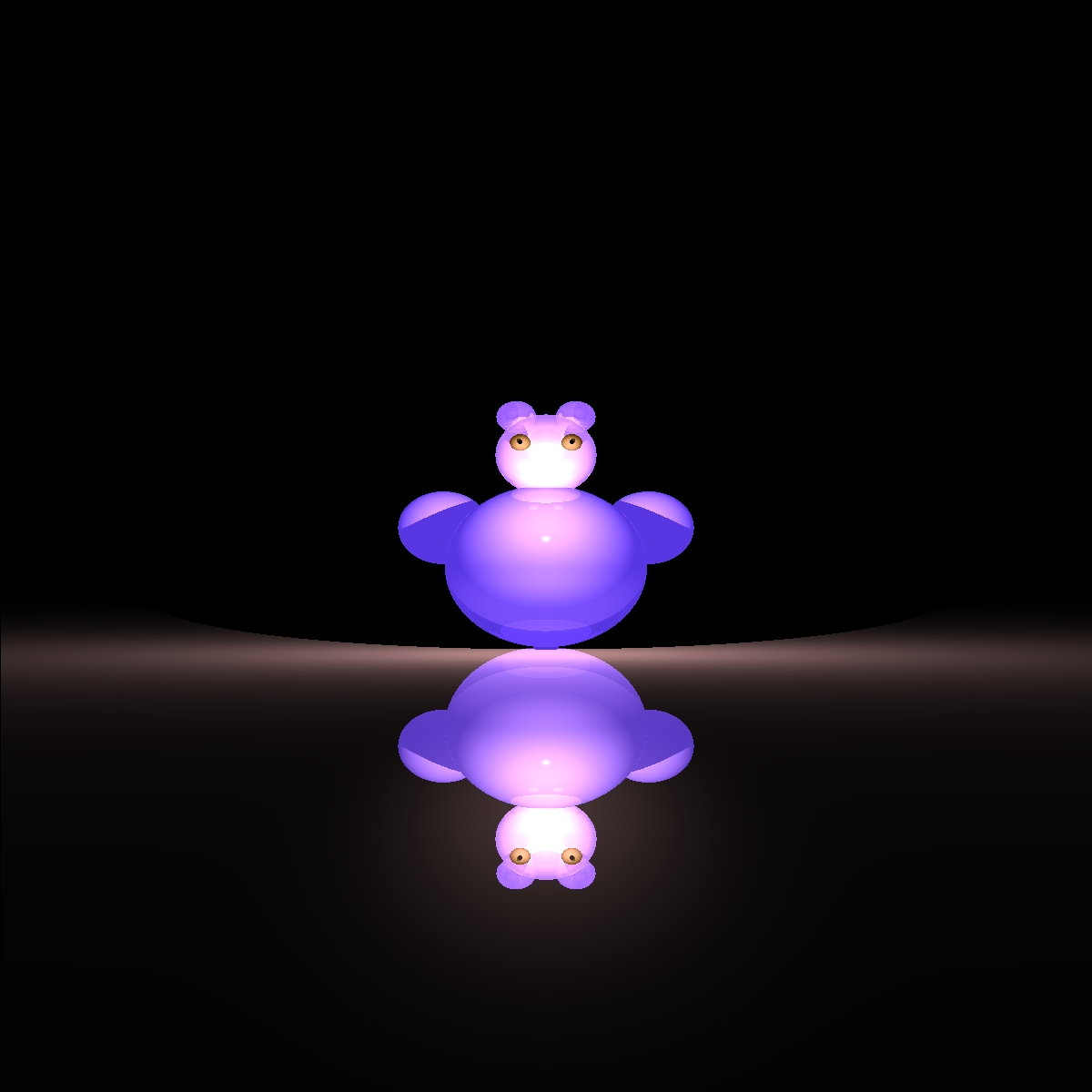
שיפור מיני פרוייקט הוא שיפור התמונה המתקבלת.

החלק שבחרנו בפיתוח הוא שיפור של משטחי הברקה . glossy surface ושיפור דרך זכוכית diffuse glass.

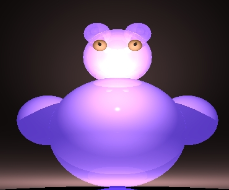
תאור הבעיה

1. משטחים מבריקים glossy , הבעיה הייתה ששכפול האלמט עשה כבייכול COPY לאלמנט עצמו מה שלא מבטא את הצורה הנכונה בפועל, משטח מבריק אפילו עד כדי מראה עדיין לא עדיין צריך להיות הבדל בין המשטח עצמו
2. כנ"ל הבעיה על משטחי זכוכית, אלמנט שנמצא לאחר משטח זכוכית בשקיפות מלאה עדיין לא צריך להראות כאילו המדובר על האלמנט עצמו.

להמחשת הפיתוח יצרנו סטס הממחיש השתקפות עם משטח הברקה כפי שניתן לראות אם התמונה ובמצב התחלתי ההשתקפות ממש כמו התמונה האמיתית כך שזה לא משקף כלל את השתקפות אלמנטים בעולם האלמנטים שהן נראות מטושטשות יותר

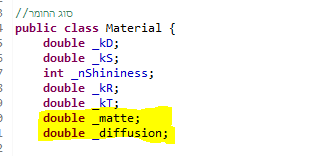


זה התמונה המשתקפת – כלל לא נראה שזה משתקף מתוך משטח הברקה

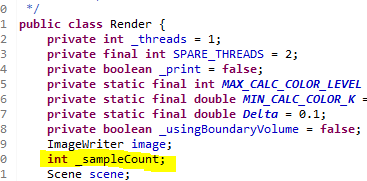


מהלך הפיתוח

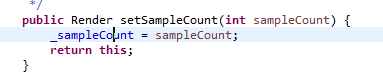
כקדם פיתוח הוספנו למחלקת material מקדמים המשקפים את הרדיוס של בדיקת הקרניים.



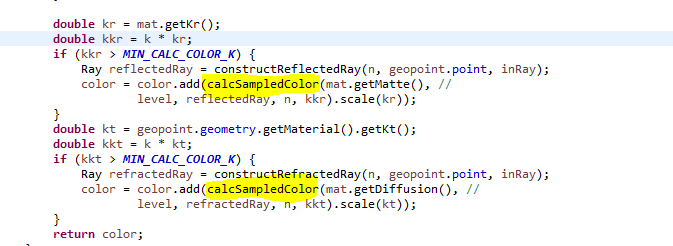
כמו כן במחלקת RENDER הוספנו משתנה המציין את מספר הקרניים למשלוח



והsetter להטמעת מספר הקרניים למשלוח



כהוספה על החישוב הקיים של שקיפות ומבריקות הוספנו קריאה לפונקציה, calcSampledColor, שמקבלת את הקרן השקיפות או ההשתקפות, וכוללת את מקדם הרדיוס שהוסף למחלקת material



**private** Color calcSampledColor(**double** radius, **int** level, Ray refRay, Vector n, **double** k) {

Point3D p0 = refRay.get\_POO();

Vector dir = refRay.get\_direction();

Point3D head = dir.get\_head();

**double** ndir = primitives.Util.*alignZero*(n.dotProduct(dir));

**if** (ndir == 0)

**return** Color.***BLACK***;

GeoPoint gp = findClosestIntersection(refRay);

Color bg = scene.getBackground();

Color color = gp == **null** ? bg : calcColor(gp, refRay, level - 1, k);

**if** (radius == 0)

**return** color;

**int** samples = **this**.getSampleCount();

**if** (samples == 0)

**return** color;

**double** x = head.get\_x().get();

**double** y = head.get\_y().get();

**double** z = head.get\_z().get();

Vector v = **null**;

**if** (x <= y) {

**if** (x <= z)

v = **new** Vector(0, z, -y).normalize();

**else**

v = **new** Vector(-y, x, 0).normalize();

} **else** {

**if** (y <= z)

v = **new** Vector(-z, 0, x).normalize();

**else**

v = **new** Vector(-y, x, 0).normalize();

}

Vector u = v.crossProduct(dir).normalize();

Point3D pc = p0.add(dir.scale(d));

**for** (**int** i = **this**.getSampleCount() - 1; i > 0; --i) {

Point3D p;

Vector w;

**double** nw;

**do** {

p = pc;

x = ***RANDOM***.nextDouble() \* 2 - 1;

y = Math.*sqrt*(1 - x \* x);

**if** (!Util.*isZero*(x))

p = p.add(v.scale(x));

**if** (!primitives.Util.*isZero*(y))

p = p.add(u.scale(y));

z = radius \* (***RANDOM***.nextDouble() \* 2 - 1);

p = pc.add(p.subtract(pc).scale(z));

w = p.subtract(p0);

nw = primitives.Util.*alignZero*(n.dotProduct(w));

} **while** (ndir > 0 && nw < 0 || ndir < 0 && nw > 0);

Ray ray = **new** Ray(p0, w);

gp = findClosestIntersection(ray);

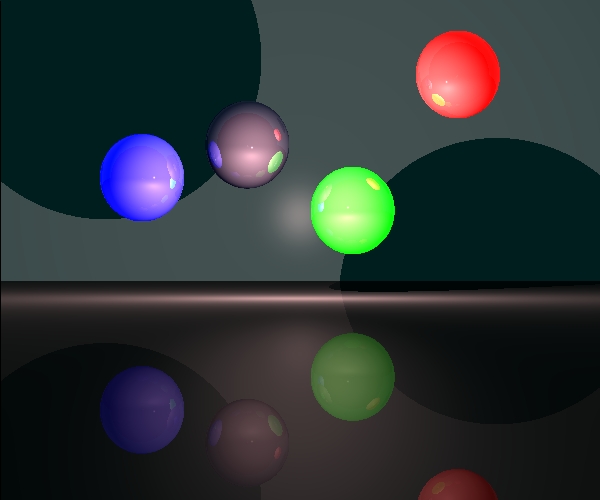
color = color.add(gp == **null** ? bg : calcColor(gp, ray, level - 1, k));

}

**return** color.reduce(samples + 1);

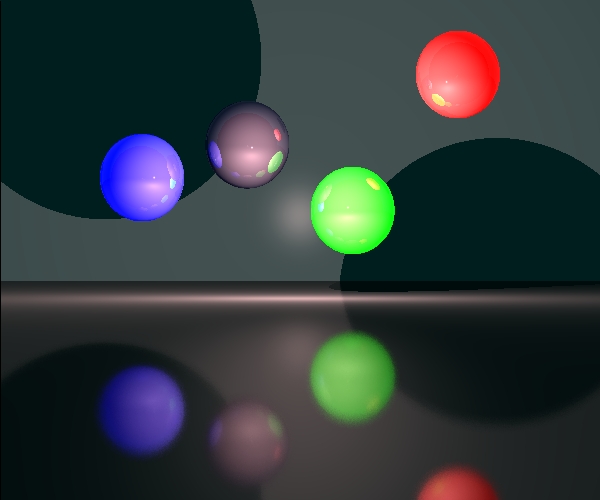
}

טסט משטח הברקה, בדוגמא ללא התייחסות מקדם



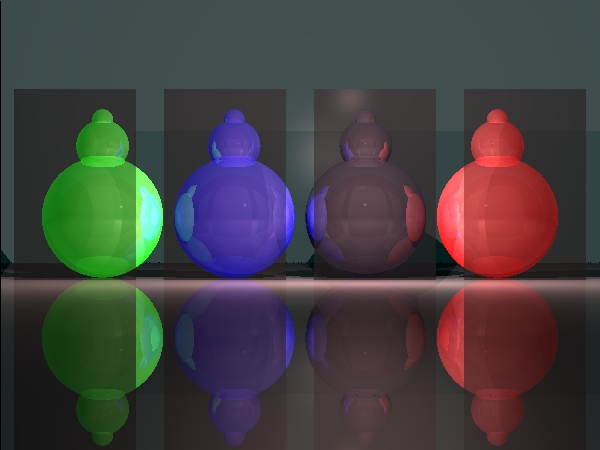
כפי שניתן לראות האלמנטים שמוצגים מתוך משטח ההברקה נראים בצורה רגילה ולא ניתן להבחין שמדובר על אלמנטים מעל משטח.

במשלוח 80 קרניים כפי ההמלצה עם רדיוס של 0.4 ניתן להבחין שהאלמנטים מוצגים בטשטוש, ואכן ניתן להבחין שהם אינם מוצגים טבעיים אלא משתקפים מתוך משטח מראה, או משטח מבריק



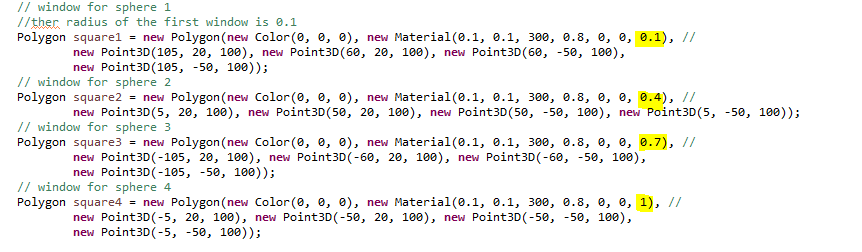
טסט מקדמי שקיפות

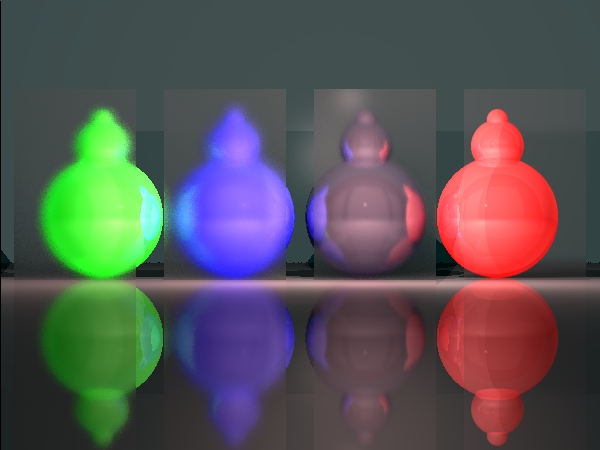
כפי שניתן לראות מדובר על צורות מאחרי משטח זכוכית שקוף, אך בפועל להוציא את הצבע של משטח הזכוכית לא ניתן להבחין כלל שמדובר על אלמנטים מאחורי משטח זכוכית



בטסט זה בדקמנו משטחי זכוכית עם רדיוס שקיפות משתנה משטח ימני עם רדיוס 0.1 עד 1

וכך ניתן ממש להבחין איך כאשר הרדיוס גדל התמונה מטשטשת יותר וכן הגבולות של האלמנט מטשטשים.





מיני פרוייקט 2

בשונה משלב ראשון של שיפור איכות התמונה, שלב שני היה שיפור יעילות טעינה.

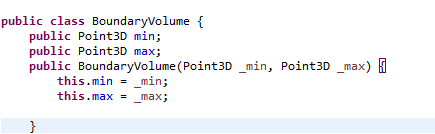
בפועל בכל פיקסל נבחר נבדקת חיתוך עם כל האלמנטים, חיתוך עם אלמנטים לעיתים דורשת חישוב ובתמונה הכוללת הרבה מאוד אלמנטים מדובר על פעולה מורכבת מאוד

ואם ניקח דוגמא לתמונה שהאלמנטים נמצאים בצד ימין תחתון של התמונה אך עבור כל הפיקסלים גם השמאלי וגם ימין עליון נבדקות נקודות חיתוך עם האלמנטים פעולה מבזבזת לחלוטין

הפיתוח בוצע ב3 שלבים

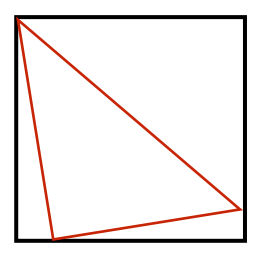
שלב ראשון

לצורך הפתרון יצרנו מחלקה המשקפת את המעטפת הויראטלית הכוללת 2 נקודות, מינימום ומקסימום



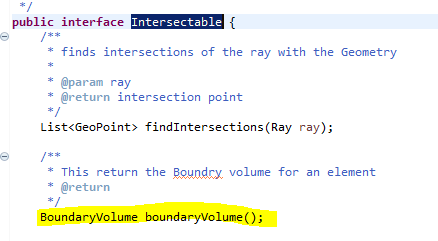
הקפנו כל אלמנט בכעין מלבן עוטף וירטואלי עם נקודת מינימום ונקודת מקסימום, ואז בודקים אם יש חיתוך במעטפת הוירטואלית, שזו פעולה פשוטה אז לבדוק חיתוך עם האלמנט עצמו.

לדוגמא

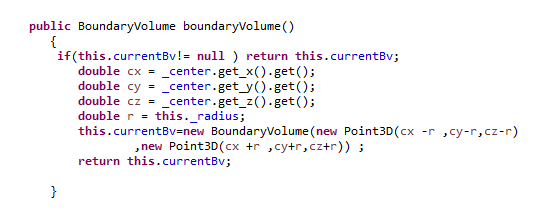


רק אם יש חיתוך עם המעטפת נבדוק חיתוך עם האלמנט עצמו.

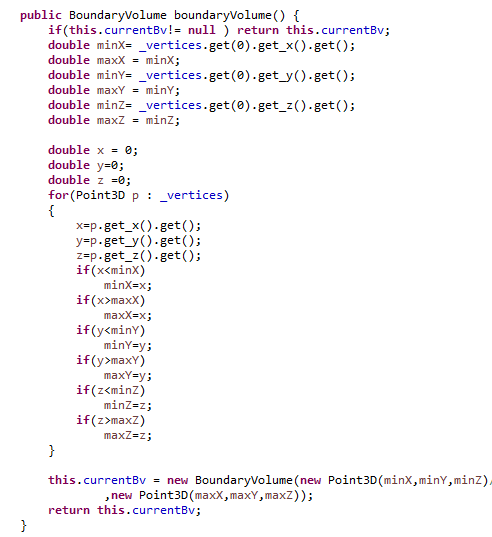
באמצעות קוד הוספנו לאינטרפייס של הצורות את הצורך לממש מעטפת וירטואלית, וממשנו את זה בכל הצורות



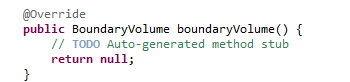
למשל **בכדור** (כפי שניתן לראות ברגע שבצענו את המימוש לא מחשבים כל פעם אלא חישוב המעטפת הויראטואלית ממומשת פעם אחת)



מימשנו גם על **פוליגון** (שכולל את המשולש) כנ"ל חישוב המעטפת הויראטואלית מבוצע רק פעם אחת

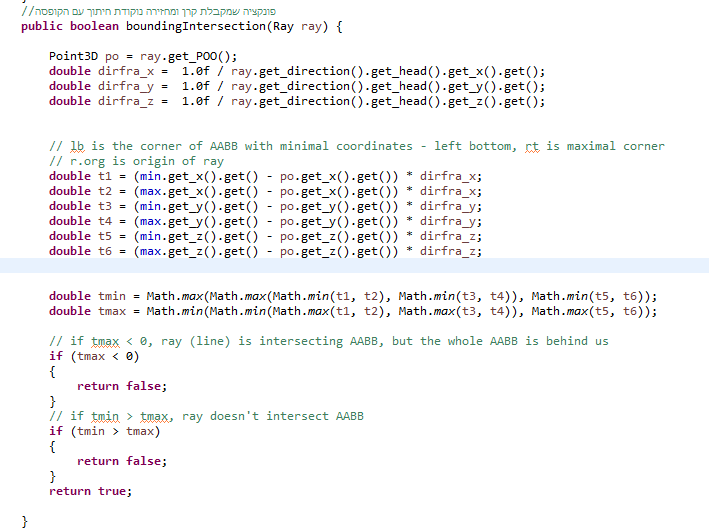


לעומת זאת במשטח שהוא לכל שטח התמונה לא החזרנו מעטפת

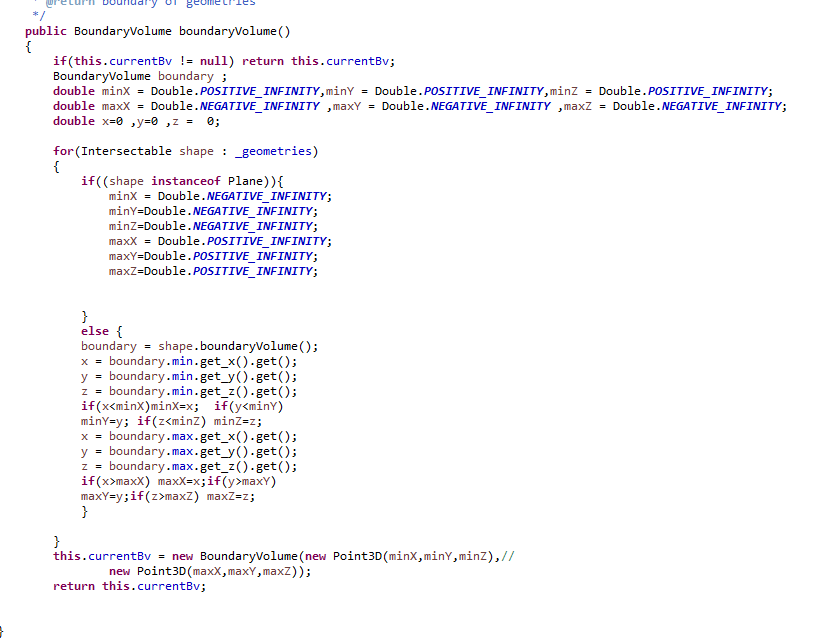


על מנת לסיים את השלב הוספנו את הבדיקה של חיתוך קרן עם מעטפת ורטואלית

למחלקת המעטפת שהוספה הוספנו פונקציה למציאת נקודות חיתוך עם המעטפת, הפונקציה הזו למרות שהיא ארוכה אינה עוסקת בחישובים והחישוב שלה פשוט.

שלב שני של הפיתוח היא יצירת מעטפת בנוסף מאלמנט בודד גם על קבוצה של אלמנטים ולבצע את אותה ההתנהלות רק במידה וקיים חיתוך במעטפת של האלמנטים עצמם יבוצע חיתוך עם המעטפות הפנימיות וכנ"ל השלב הקודם (חיתוך עם מעטפת אלמנט ורק אם כן חיתוך עם אלמנט עצמו)

לצורך זה הוספנו גם לGeometries חישוב מעטפת. כאשר היא מחשבת את נקודת המינימום מן כל הצורות הפנימיות והמקסימום של כל הצורות הפנימיות.



ורק אם קיימת נקודת חיתוך בקבוצה אז בודקים את האלמנטים בתוך הקבוצה.

לבדיקת טסט זה יצרנו תמונה הכוללת הרבה מאוד אלמנטים אך הם חולקו לקבוצות.

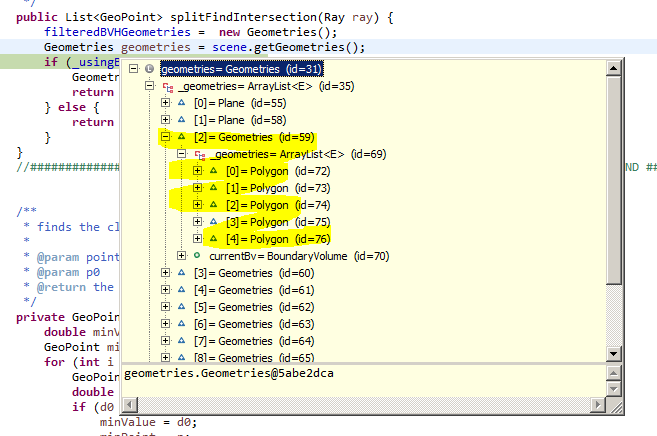


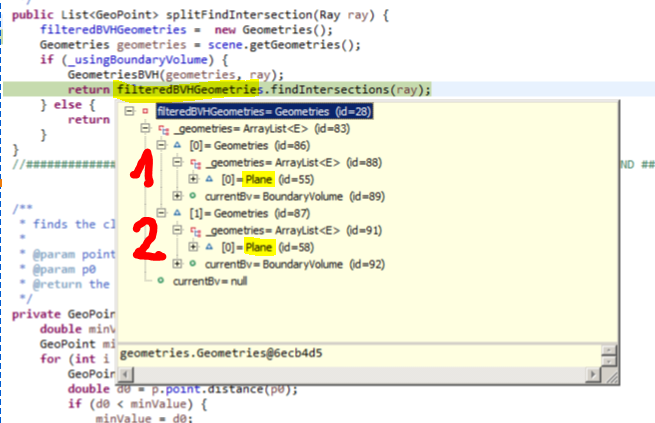
כל אות הכנסנו לקבוצה אחת וכך במקום לבדוק את כל האלמנטים בדקנו חיתוך עם הקבוצה במידה ואין חיתוך ממשיכים הלאה.

רק לשם המחשה על היעילות יצרנו BREAKPOINT בנקודה (999,999) שבה אין חיתוך עם שום אלמנט

מספר האלמנטים הקיימים הוא 11 קבוצות של אלמנטים (שמשקפות אותיות) ובכל קבוצה בממוצע של 4 אלמנטים

כך זה כל קבוצות האלמנטים,

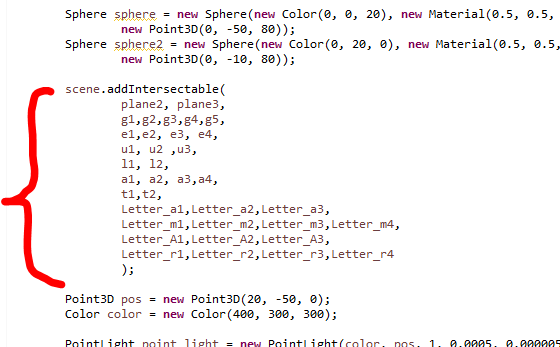
 אך לאחר סינון האלמנטים איתם אנו מעוניינים לבצע חיתוך נמצאו רק 2 אלמנטים שאלו המשטחים בלבד



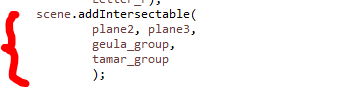
מדובר על חיסכון משמעותי במקום לחפש חיתוך עם כל האלמנטים נבדק חיתוכים רלוונטיים בלבד

לבדיקת היעילות יצרנו את אותו הטסט ואותה התמונה פעם עם חיתוך "רגיל" של כל האלמנטים ופעם מקובץ, והתוצאות גם בטסט יחסית מינורי היו **משמעותיות**

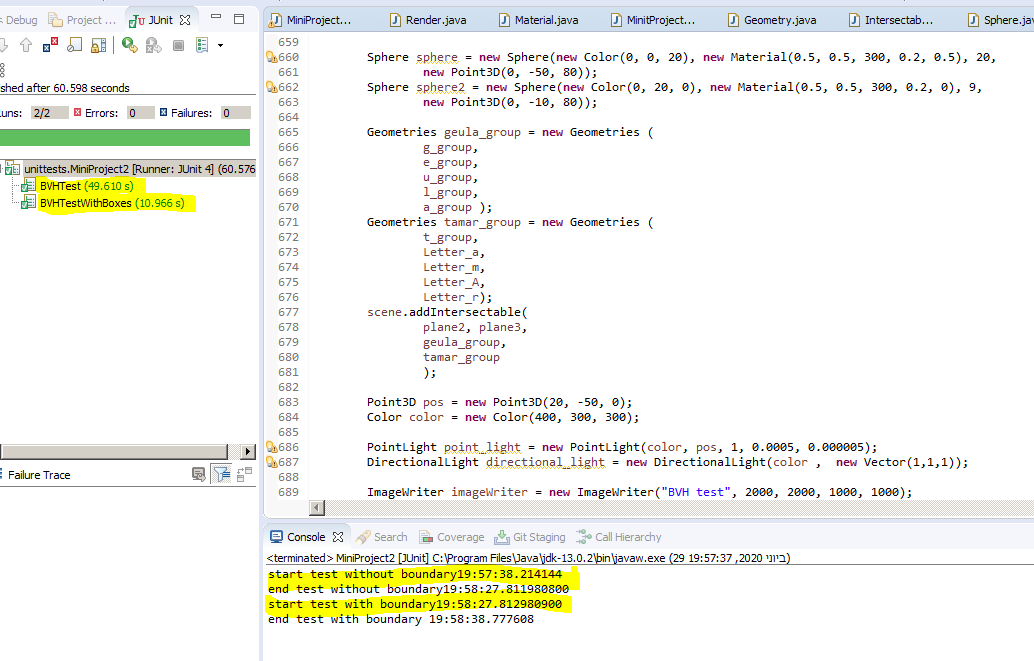
כך הכנסנו את הצורות בטסט הרגיל



וכך הכנסנו בטסט מקובץ



והמהירות גם באופן הרגיל הייתה פי 5 יותר איטית

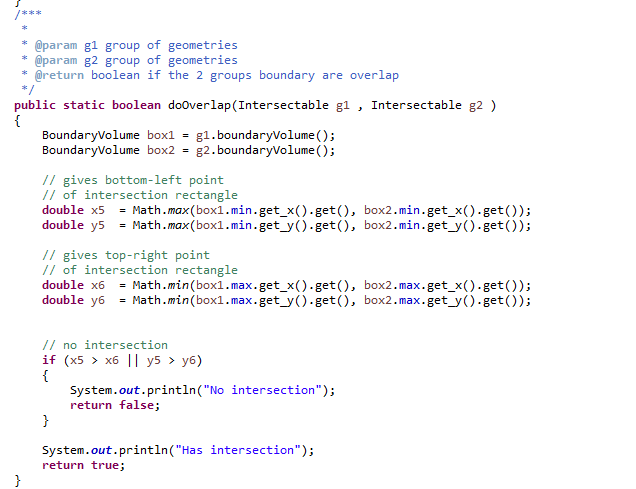


**שלב השלישי** הוא לבצע את שלב חלוקת הקבוצות באופן אוטומטי

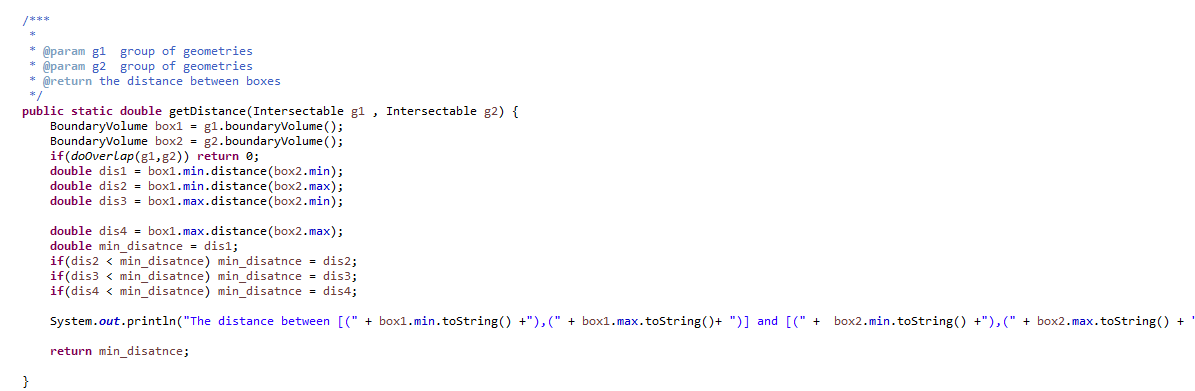
הוספנו פונקציה המחשבת מרחק בין 2 מעטפות וירטואליות שכאמור הן עוטפות את האלמנטים וכן קבוצות של אלמנטים

את פונקצית חישוב מרחק הפרדנו ל-2, פונקציה הבודקת האם האלמנטים חופפים, ואז מחזירה מרחק 0, ואם הם לא חופפים מחשבת את המרחק הוירטואלי בינהם

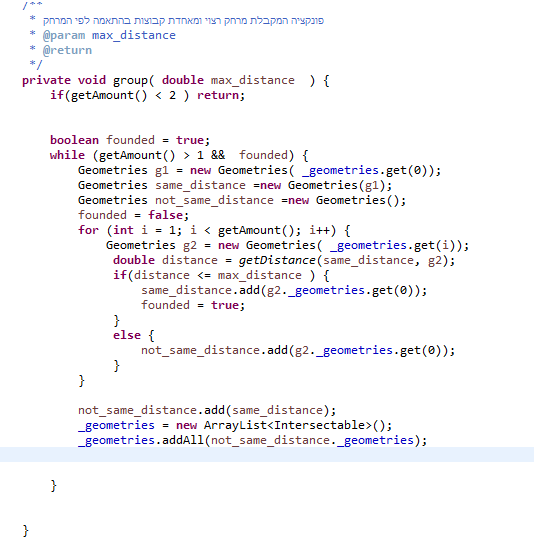
פונקציה הבודקת האם האלמנטים חופפים



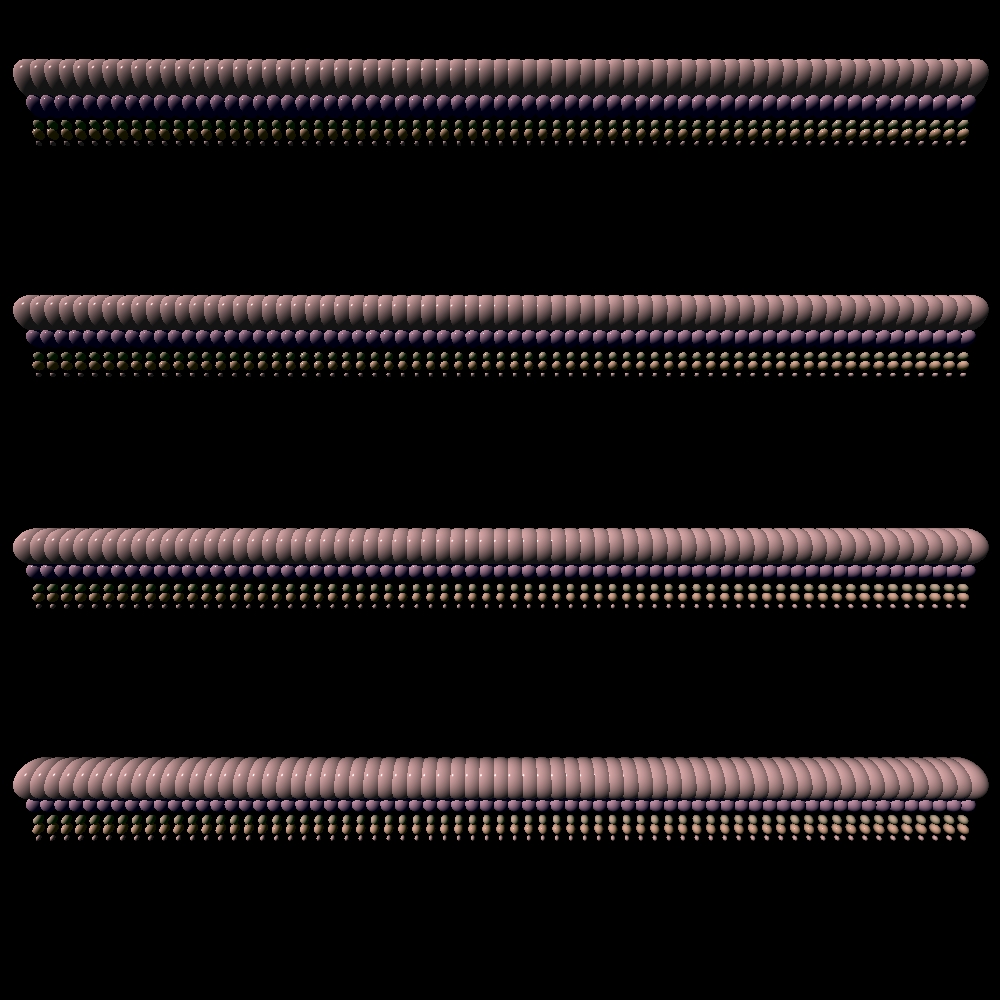
ופונקציה הבודקת מרחק בין האלמנטים

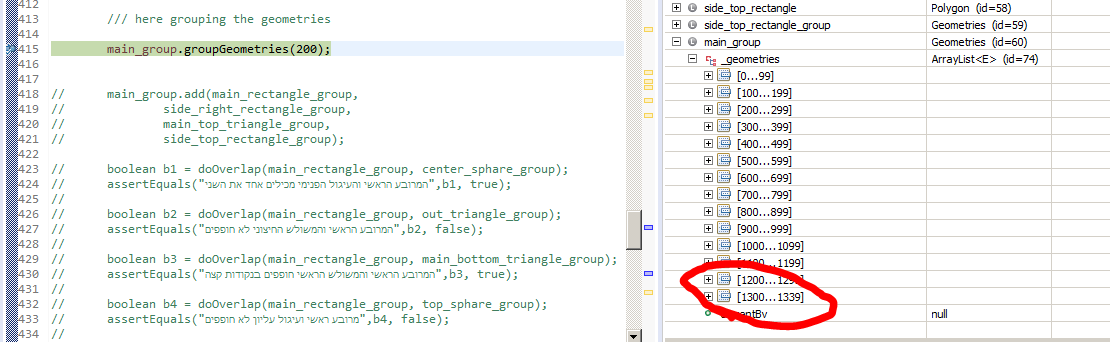


בנוסף יצרנו פונקציה המקבלת את המרחק אותו אנחנו רוצים לאחד, לדוגמא אנו דורשים שהמערכת תאחד אלמנטים שהמרחק שלהם פחות מ10, הנתון הזה כמובן משתנה בין גודל התמונה וגודל האלמנטים

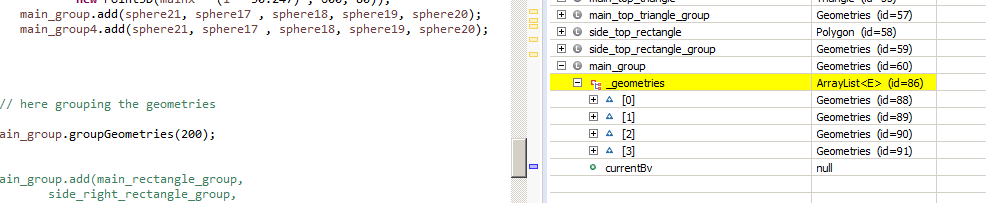


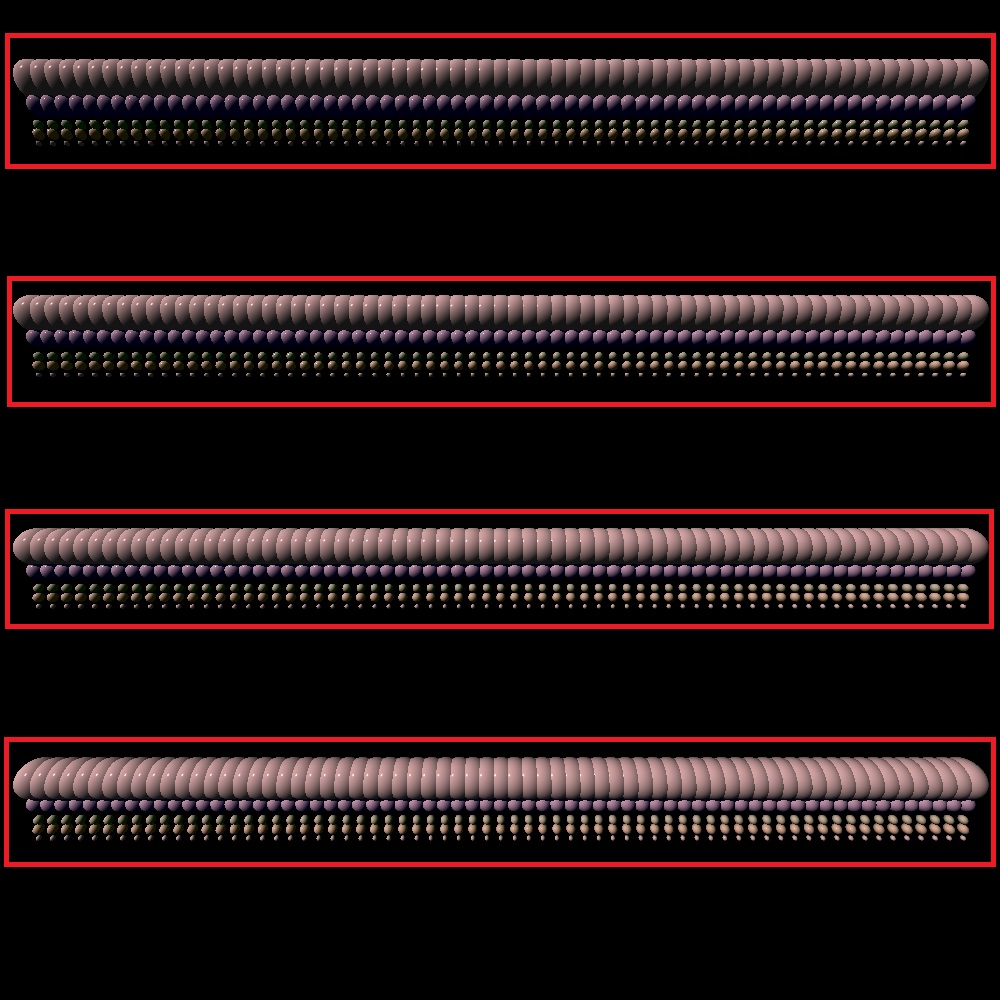
יצרנו טסט עם מספר גאומטריות מאוד גדול 1339 גאומטריות





ולאחר פעולת איחוד קבוצות המנגנון איחד ל-4 קבוצות





## והתוצאה בלתי תאמן, 53 שניות כאשר אנו מקבצים מול 364 שניות ללא קיבוץ

