דוח מס' 2: שיפורי זמן

מגישות: חיה יזרסקי 5876 ונחמה ברון 8830

תיאור הבעיה: לוקח למחשב הרבה זמן לייצר את התמונה, במיוחד אם השתמשנו בשיפורי תמונה.

פתרונות שבחרנו:

- Multi-threading
- Adaptive Super Sampling

בחרנו בשיפור זמן של Adaptive Super Sampling כי השתמשנו בgrid בשיפורי תמונה.

עכשיו נחלק כל פיקסל בתמונה לרבעים ברקורסיה רק אם יש באמת צורך ושוני בין הפינות של הפיקסל.
שיפרנו רק לAnti-aliasing, כי מותר לשפר רק לשיפור תמונה אחד.

MULTI-THREADING:

עכשיו נפרט על Multi-threading- שימוש בכמה תהליכונים בו זמנית, בשביל לחשב את צבעי התמונה. מה זה ריבוי תהליכונים? חלוקה של העבודה לחישוב (התהליך) לחלקים יותר קטנים, כדי שיהיה אפשר לחשב אותם במקביל. וכאן באמת אפשר לחשב את צבעי הפיקסלים במקביל, כי אין תלות בין חישוב הצבע של כל פיקסל ופיקסל.

נוסיף את הקוד הבא בתוך פונקציית renderImage:

```
if (_MultiThreadingButton) {
    Pixel.initialize(_imageWriter.getNy(), _imageWriter.getNx(), printInterval); //debug print is pri
    int threadsCount = 3;
    while (threadsCount-- > 0) {
        new Thread(() -> {
            for (Pixel pixel = new Pixel(); pixel.nextPixel(); Pixel.pixelDone()) {
                if(_adaptiveSuperSampling) {
                    //implementing the adaptive super sampling time improvement of part 9
                    Color thisPixelColor = castRay(pixel.col, pixel.row);
                    _imageWriter.writePixel(pixel.col, pixel.row, thisPixelColor);
                }
                else{
                    Color thisPixelColor = castRayOld(pixel.col, pixel.row);
                    _imageWriter.writePixel(pixel.col, pixel.row, thisPixelColor);
                }

            }
        }).start();

    } Pixel.waitToFinish();
```

אם הכפתור של ריבוי תהליכונים לא דלוק, נבצע את הקוד הישן.
בנוסף, נחלק לאם Adaptive Super Sampling דלוק או לא. נפרט על כך יותר בהמשך.

איך הקוד עובד:

בשלב הראשון, נאתחל את מספר השורות ומספר העמודות שיש בתמונה. נצטרך אותם בשביל לדעת כמה עבודה של פיקסלים יש לעשות, וכל כמה שניות אנחנו רוצים להדפיס כמה אחוז עבודה כבר סיימנו.

אחר כך ניצור תהליכונים לפי הכמות של מונה התהליכונים. כל תהליכון כזה יבצע את הלולאה הבאה:

התהליכון מקבל את הפיקסל הבא שצריך לחשב עבורו צבע בעזרת קריאה לפונקציה nextPixel.
התהליכון מבצע את פונקציות חישוב הצבעים לפי הכפתורים הדלוקים ולבסוף קובע את הצבע של אותו

פיקסל. לאחר מכן התהליכון מודיע על סיום העבודה עם אותו פיקסל בעזרת פונקציית pixelDone.
הפונקציה הזו מעדכנת את מספר הפיקסלים שסיימו ומאפשרת הדפסה של אחוז העבודה שכבר הסתיים.

על כל פיקסל שנוצר צריך לקרוא סטרט בשביל להתחיל את התהליכון.

כפתורים כדי להדליק ולכבות את השינויים שעשינו :

```
private static boolean _MultiThreadingButton = false;
private static int NUM_OF_THREADS = 1;
private double printInterval = 0;
```

סטרים לשימוש בשיפור של ריבוי תהליכונים-

```
public Camera setMultithreading(int i) {
   if (i > 1)
      _MultiThreadingButton = true;
   NUM_OF_THREADS = i;
   return this;
}
```

ניתן להגדיר כמה תהליכונים אנחנו רוצים בו זמנית, ואם הוא גדול מאחד, אז נדליק את כפתור השימוש בריבוי תהליכונים ונאתחל את מספר התהליכונים.

פונקציה נוספת, setDebugPrint מגדירה כל כמה שניות (שולחים שניות) יש להדפיס כמה אחוזים מהעבודה כבר עשינו.

```
/***
 * set button for multi-threading (part 9) this allows for easier debugging
 * @param v interval of how often to print
 * @return the camera
 */
public Camera setDebugPrint(double v) {
   printInterval = v;
   return this;

}
```

נראה את הקריאה לסטרים בטסט-

```
private final Camera camera = new Camera(new Point(0, 0, -1000), new Vector(0, 0, 1), new
Vector(0, 1, 0)) //
      .setVPDistance(1000).setVPSize(200, 200) //
      .setImageWriter(imageWriter) //
      .setMultithreading(3).setDebugPrint(0.1);
```

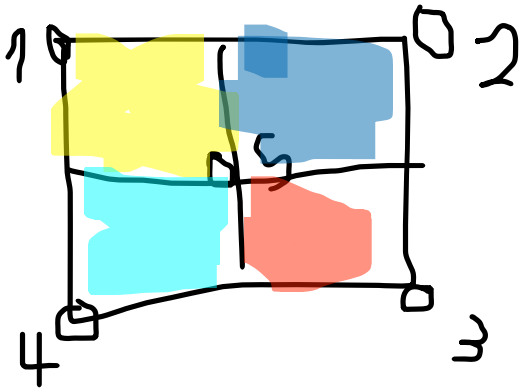
זה החלק שמסומן בצהוב.
הרצנו את התמונה שנתנו לנו בשביל התהליכונים-

בלי השיפור לוקח 2 דקות להריץ.

עם השיפור לוקח 1 להריץ.

ADAPTIVE SUPER SAMPLING:

עכשיו נפרט על Adaptive Super Sampling. כשיש פיקסל שרובו צבע אחד, כשנחלק לאזורים קטנים בתוכו בפועל נחשב שוב ושוב את אותו צבע בדיוק. כדי למנוע חזרה על חישוב של אותו צבע כמה פעמים, נחלק לאזורים קטנים רק אם יש הבדל בין הפינות והמרכז של פיקסל מסויים. רק נכנס לחישוב רקורסיבי כשיש צורך בכך.



בציור פה למשל, אם הצבעים של 1 ו5 שונים, נשלח לרקורסיית חישוב צבע את הרבע הצהוב.

ואת הצהוב נחשב באותה דרך, חישוב ארבע פינות מול המרכז, וכן הלאה. אבל בעצם נקודה מספר אחת תחושב גם בחישוב של הפינות ברקורסיית חישוב הרבע הצהוב, לכן הרקורסיה שלנו מקבלת הרבה פרמטרים, בשביל לחסוך חישוב של נקודות וצבעים כל פעם מחדש. בסופו של דבר לכל רבע ניתן אחוז יחסי בצבע הסופי.

גם לשיפור הזה, יצרנו כפתור שבודק אם דלוק, וסטר שמפעיל אותו. השדות הרלוונטיים:

```
/**
 * feature of part 9
 * /ON/OFF button default is off
 */
private boolean _adaptiveSuperSampling = false;
/***
 * maximum recursion, for button
 */
public static int MAX_RECURSION = 3;
```

והנה הסטרים שלהם-

```
public Camera setAdaptiveSuperSampling(boolean adaptiveSuperSampling) {
    _adaptiveSuperSampling = adaptiveSuperSampling;
    return this;
}
```

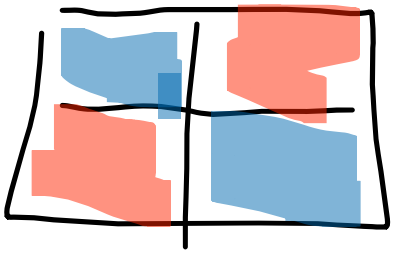
```
/***
 * on/off button for adaptive super sampling (part 9)
 * @param adaptiveSuperSampling on/off
 * @param recursion max amount of recursions
 * @return the camera
 */
public Camera setAdaptiveSuperSampling(boolean adaptiveSuperSampling, int recursion) {
    _adaptiveSuperSampling = adaptiveSuperSampling;
    MAX_RECURSION = recursion;
    return this;
}
```

עשינו פה שני סטרים, אחד שמקבל רק אם להדליק או לא, ואז כמות הרקורסיה היא לפי ברירת המחדל. ועשינו סטר אחד שמקבל גם את מספר הרקורסיות, וגם שאכן הדלקנו את הכפתור, שאנחנו רוצים אדפטיב סופר סמפלינג.

המימוש בפועל קורה בפונקציית castRay.

```java
if (_JaggedEdgesButton) {
    //initial color for average calculation
    double colorX = 0;
    double colorY = 0;
    double colorZ = 0;

    //creating the initial parameters for the recursive calculation

    Point thisPixelPoint = createMiddlePixel(_imageWriter.getNx(), _imageWriter.getNy(), j, i);
    // out.print(thisPixelPoint);

    //creating a point in the top left corner
    Point topLeftCorner = thisPixelPoint;
    Color topLeftColor = _rayTracerBase.traceRay(new Ray(_centerCam, topLeftCorner.subtract(_centerCam).normalize()));


    Point bottomRightCorner = thisPixelPoint.add(_Vup.scale((_heightVP / _imageWriter.getNy()) * -1));
    bottomRightCorner = bottomRightCorner.add(_Vright.scale((_widthVP / _imageWriter.getNx())));
    Color bottomRightColor = _rayTracerBase.traceRay(new Ray(_centerCam, bottomRightCorner.subtract(_centerCam).normalize()));

    //calling recursive function to calculate the color for this pixel
    Color fullPixelColor = recursivePixelColor(topLeftCorner, topLeftColor, bottomRightCorner, bottomRightColor,
            _Vup, _Vright, leftRight: "left", recursionNum: 0, (_heightVP / _imageWriter.getNy()), (_widthVP / _imageWriter.getNx()));

    return fullPixelColor;


}
return Color.BLACK;
```

הסבר: אם כפתור jaggedEdges דלוק - כלומר אנחנו רוצים לטפל בבעיה של צורות חדות בעזרת anti-aliasing. נשתמש בפונקצייה createMiddlePixel כדי ליצור את נקודת הקצה השמאלי העליון של הפיקסל, בעזרתו ובעזרת כיווני המצלמה וגודל הפיקסל נשיג את הקצה הימני התחתון, נחשב את הצבעים של הנקודות, ונשלח לרקורסייה. הרקורסייה גם מקבלת איזה סוג של פיקסל היא קיבלה, כי יש שני סוגים:
left and right:



הרבעים האדומים הם רבעים שמקבלים נקודות ימנית עליונה ושמאלית תחתונה מחושבות מראש, והרבעים הכחולים הם רבעים שמקבלים נקודות ימנית תחתונה ושמאלית עליונה. עשינו את החלוקה הזו, כדי שנוכל לשלוח נקודות שכבר חישבנו, ונדע איזה סוג כל נקודה. נסמן את ההבדל "left" ו-"right" שנשלחים כמחרוזת. ברמה הראשונה הסוג הוא "left".

הרקורסייה גם מקבלת וקטורי כיוון, רמת רקורסיה, רוחב וגובה הפיקסל(השלושה אחרונים משתנים בכל רמה).

אופן עבודת הרקורסיה:

תנאי עצירה:

```java
if (recursionNum >= MAX_RECURSION)
    return upperColor;
```

אם נכנסנו למספיק רמות רקורסיה, תחזיר את הצבע העליון כצבע של הרבע. מקסימום רקורסיות זה שדה של המחלקה Camera.

יש לנו משתנים כלליים לפונקציה, ששומרים את הנקודות בפינות ובמרכז ואת הצבעים שלהם:

```
Point leftTopPoint;
Point rightTopPoint;
Point leftBottomPoint;
Point rightBottomPoint;
Point middlePoint;

Color leftTopColor;
Color rightTopColor;
Color leftBottomColor;
Color rightBottomColor;
```

אם קיבלנו רבע מסוג "left"- נבצע את הקוד הבא-

```
if (leftRight == "left") {
    middlePoint = upperCorner.add(_Vup.scale(heightOfPixelSection * -0.5));
    middlePoint = middlePoint.add(_Vright.scale(widthOfPixelSection * 0.5)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);

    rightBottomPoint = bottomCorner.movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    leftTopPoint = upperCorner.movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    rightTopPoint = upperCorner.add(_Vright.scale(widthOfPixelSection)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    leftBottomPoint = upperCorner.add(_Vup.scale(-1 * heightOfPixelSection)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);

    leftTopColor = upperColor;
    rightBottomColor = bottomColor;
    rightTopColor = _rayTracerBase.traceRay(new Ray(_centerCam, rightTopPoint.subtract(_centerCam).normalize()));
    //if(!rightTopColor.equals(Color.MAGENTA)){
    //  out.print(rightTopColor);
    //}
    leftBottomColor = _rayTracerBase.traceRay(new Ray(_centerCam, leftBottomPoint.subtract(_centerCam).normalize()));
```

כלומר, ימנית תחתונה ושמאלית עליונה כבר נתונות לנו. נזיז את הנקודות באופן רנדומלי לרוחב ולגובה, אבל נזיז מקסימום ברבע מהרוחב והגובה, כי יש גם את נקודת המרכז שנצטרך להזיז, ורק בשבילה נצטרך חצי מהגובה והרוחב, שתוכל לזוז לכל כיוון. הסיבה שלא נתנו מעבר היא כדי שישאר הפרופורציה בין ימנית תחתונה ושמאלית עליונה, שלא יתהפכו והשמאלית תהיה מימין לימנית וכן הלאה. את החישוב של הנקודות החסרות נעשה בעזרת תזוזה בכיווני הוקטורים שקיבלנו. נחשב את הצבעים לנקודות החדשות בעזרת קריאה לtraceRay, שמקבל את הקרן דרך הנקודה של הפיקסל, ומחזיר צבע שרואים מהנקודה.

אם קיבלנו רבע מסוג "right", מבצעים את הקוד הבא-

```
} else {
    middlePoint = upperCorner.add(_Vup.scale(heightOfPixelSection * -0.5));
    middlePoint = middlePoint.add(_Vright.scale(widthOfPixelSection * -0.5)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);

    leftBottomPoint = bottomCorner.movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    rightTopPoint = upperCorner.movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    leftTopPoint = upperCorner.add(_Vright.scale(-1 * widthOfPixelSection)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);
    rightBottomPoint = upperCorner.add(_Vup.scale(-1 * heightOfPixelSection)).movePointRandom(_Vup, _Vright, aperture: widthOfPixelSection / 4);


    rightTopColor = upperColor;
    leftBottomColor = bottomColor;
    leftTopColor = _rayTracerBase.traceRay(new Ray(_centerCam, leftTopPoint.subtract(_centerCam).normalize()));
    rightBottomColor = _rayTracerBase.traceRay(new Ray(_centerCam, rightBottomPoint.subtract(_centerCam).normalize()));
}
    Color leftBottomColor
    ISE5782_8830_5876
```

שמבצע כמעט אותו דבר, אבל הפעם הנקודות שכבר יש לו הם הימנית העליונה והשמאלית התחתונה, וצריך לחשב את הנקודות האחרות, ואת הצבעים שלהם, שיחושבו בעזרת תזוזה מהנקודות הקיימות. כיווני התזוזה יהיו בהתאם לנקודה שנצטרך והנקודה ממנה התחלנו.

נקודת המרכז תחושב תמיד אותו דבר.

```
Color middleColor = _rayTracerBase.traceRay(new Ray(_centerCam,
middlePoint.subtract(_centerCam).normalize()));
```

כעת, שמורים לנו בשני המקרים כל הנקודות בתור איזה נקודה הם, ימין שמאל +למעלה או למטה. המשך הקוד לא תלוי באופן מסירת הנקודות.

```
if (!middleColor.equals(rightTopColor)) {

    topRight = recursivePixelColor(rightTopPoint, rightTopColor, middlePoint, middleColor, Vup, Vright,  leftRight: "right",
            recursionNum: recursionNum + 1,  widthOfPixelSection: widthOfPixelSection / 2,  heightOfPixelSection: heightOfPixelSection / 2);
    // out.print(topRight);
}

Color topLeft = middleColor;
if (!middleColor.equals(leftTopColor)) {
    topLeft = recursivePixelColor(leftTopPoint, leftTopColor, middlePoint, middleColor, Vup, Vright,  leftRight: "left",
            recursionNum: recursionNum + 1,  widthOfPixelSection: widthOfPixelSection / 2,  heightOfPixelSection: heightOfPixelSection / 2);
}
//great
Color bottomRight = middleColor;
if (!middleColor.equals(rightBottomColor)) {
    bottomRight = recursivePixelColor(middlePoint, middleColor, rightBottomPoint, rightBottomColor, Vup, Vright,  leftRight: "left",
            recursionNum: recursionNum + 1,  widthOfPixelSection: widthOfPixelSection / 2,  heightOfPixelSection: heightOfPixelSection / 2);
}

Color bottomLeft = middleColor;
if (!middleColor.equals(leftBottomColor)) {
    bottomLeft = recursivePixelColor(middlePoint, middleColor, leftBottomPoint, leftBottomColor, Vup, Vright,  leftRight: "right",
            recursionNum: recursionNum + 1,  widthOfPixelSection: widthOfPixelSection / 2,  heightOfPixelSection: heightOfPixelSection / 2);
}
```

לכל רבע מהפיקסל נבדוק האם הצבע של נקודת הקצה שווה לזו של נקודת המרכז. אם הם שווים, נקבע את הצבע של הרבע הנוכחי להיות הצבע של המרכז. אם הם שונים נחשב את הצבע של אותה רבע בעזרת קריאה רקורסיבית : רמת הרקורסייה תעלה באחד, והאורך והגובה יחולקו בשניים.

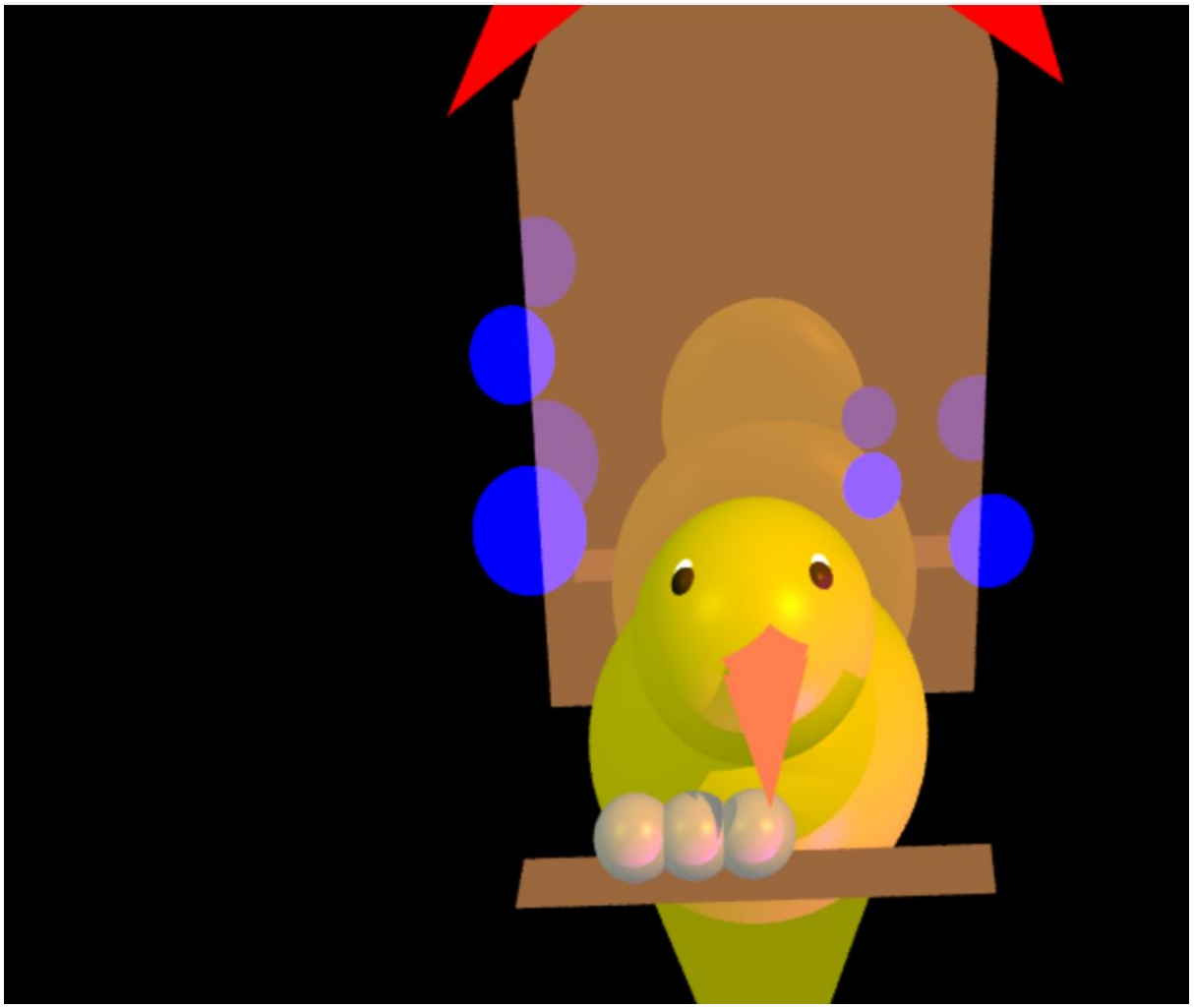
לדוגמא, בתנאי הראשון, נבדוק האם צבע הימנית העליונה שונה מנקודת המרכז, ואם כן, אז לרבע הזה נצטרך לחשב את הצבע, ששונה מהצבע שנמצא בעליונה עכשיו (כל פינה מייצגת את הצבע של הרבע המתאים), לכן נחשב את צבע הרבע בעזרת קריאה רקורסיבית לפונקצייה.

במקרה הנ"ל נשלח סוג ימין לרקורסיה, (הסוג הוא לפי איזה צד הנקודה העליונה), ולכן נשלח את הנקודה הנוכחית כנקודה העליונה, ונקודת המרכז כנקודה התחתונה. יישלחו גם הצבעים המתאימים.

בסופו של דבר חישבנו ממוצע בין הצבעים של הרבעים, לכל רבע נתנו משקל שווה.

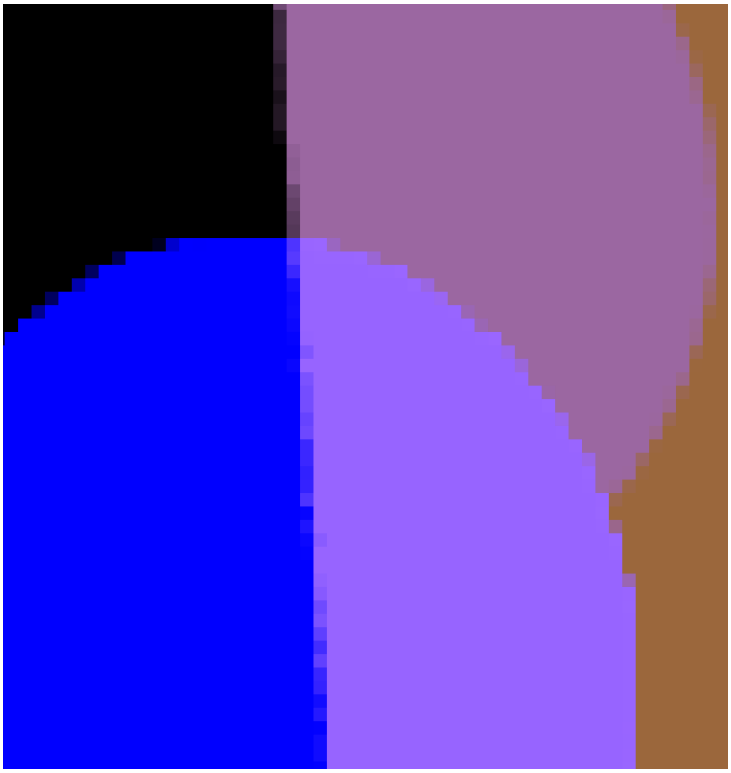
```
Color totalColor = topRight.add(topLeft, bottomRight, bottomLeft);
Color averageColor = totalColor.scale(0.25);
return averageColor;
```

הרצנו את בדיקת הקוד על הטסט שלנו שמשתמש בJaggedEdges. נראה איך נראית התמונה לאחר הדלקה של הכפתור :

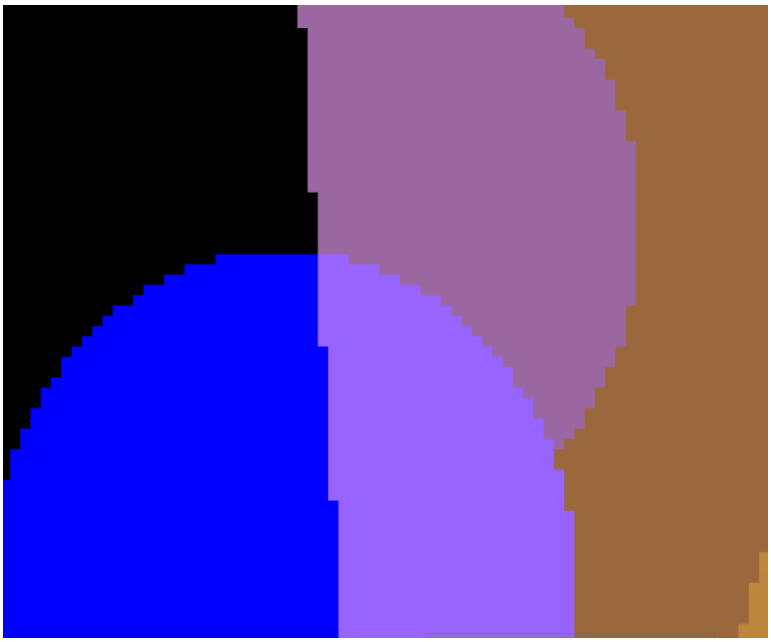


ניתן לראות שהוא עם שיפור תמונה אם נצלם יותר מקרוב. אף על פי שעשינו 400 קרניים (20 *20), ההבדל לא עד כדי כך ברור לעין.

נצלם יותר מקרוב-



לעומת



השוואת זמני ריצה :

זמני ריצה בלי שיפור זמן כלל : 19 דקות
(מספר קרניים מקסימום : 400)

זמני ריצה עם שיפור :

רמת רקורסיה : 5
זמן : 8 דקות
(מספר קרניים מקסימום : 1024)

רמת רקורסיה : 4
זמן : 4 דקות
(מספר קרניים מקסימום : 256)

כלומר, השיפור חוסך זמן באופן מהותי!