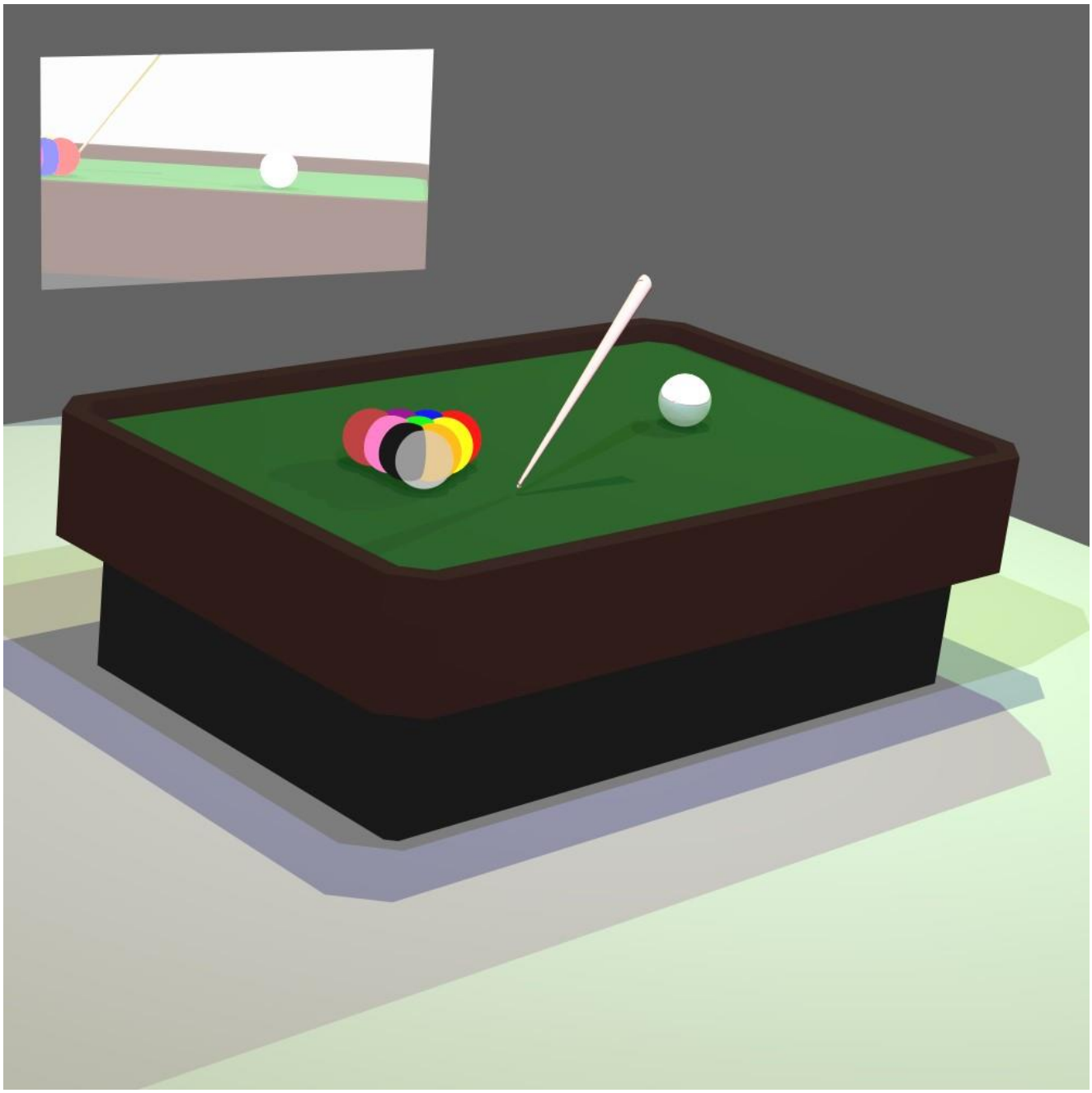
בס"ד

דו"ח שיפורים

מיני פרויקט במבוא להנדסת תוכנה- 151055



מגישים:

בניה טרבלסי – 325836922

יקיר מעודה - 322434549

שיפורי תמונה:

תיאור הבעיה:

בעת עיבוד תמונה, ייתכן שהצורות המתקבלות ייראו מחוספסות ולא חלקות. תופעה זו מתרחשת משום שהצבע מחושב על פי מרכז הפיקסל בלבד, דבר המוביל להבלטת החלוקה לפיקסלים ולגבולות חדים שאינם מציאותיים.

הצעת השיפור:

בכדי לשפר את איכות התמונה ולהפחית את המראה המחוספס, ניישם טכניקה שנקראת "super-sampling" באמצעות אלגוריתם רנדומלי. הרעיון המרכזי בטכניקה זו הוא לשלוח מספר קרניים לכל פיקסל ולחשב את ממוצע הצבע המתקבל מכל הקרניים הללו. בצורה זו, בקצוות הצורה יתקבל צבע משולב והגבולות יהיו "רכים" יותר, מה שיגרום לצורות להיראות חלקות וטבעיות יותר.

תהליך יישום השיפור:

1. שליחת קרניים רנדומליות: לכל פיקסל נשלח מספר קרניים שממוקמות באופן רנדומלי בתוך הפיקסל, בנוסף לקרן המקורית.

2. חישוב ממוצע הצבע: נחשב את ממוצע הצבע מכל הקרניים שנשלחו לפיקסל, כך שהצבע המתקבל יהיה ממוצע של כל הקרניים.

3. שילוב הצבעים: הצבע המשולב שמתקבל יחליף את הצבע המקורי של הפיקסל, ויעניק לתמונה מראה חלק וטבעי יותר.

באמצעות יישום השיפור הנ"ל, נוכל להשיג תמונות בעלות גבולות רכים יותר וצורות חלקות וטבעיות יותר, כך שהמראה הכללי של התמונה ישתפר בצורה ניכרת.

בס"ד

כך מימשנו זאת:

```
private void castRays(int nX, int nY, int j, int i) {  2 usages  Benaya Trabelsi *
    // 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 < numSamples * numSamples; ++k) {
        // Calculate the X and Y offsets for the current sample.
        double x = j + (k % numSamples + (Math.random() - 0.5)) / (double) numSamples;
        double y = i + (k / numSamples + (Math.random() - 0.5)) / (double) numSamples;

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

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

    // Mark the pixel as processed.
    Pixel.pixelDone();
}
```

נציג את ההבדלים:

לפני השיפור

אחרי השיפור

שיפורי זמן ריצה:

Multithreading

תיאור הבעיה:

עד כה, תהליך עיבוד התמונות רץ על גבי תהליכון (thread) יחיד, מה שלא ניצל באופן מיטבי את יכולות המעבד בעל ליבות רבות. כתוצאה מכך, זמן הריצה של עיבוד התמונות היה ארוך מאוד.

הצעת הפתרון:

בכדי לשפר את זמן הריצה ולנצל את יכולות המעבד בצורה מיטבית, ניישם גישת multithreading (ריבוי תהליכונים).

תהליך יישום השיפור:

1. הפרדת התהליכים: נפריד את תהליך עיבוד התמונה לתהליכונים שונים (threads), כך שכל תהליכון יעסוק בצביעה של פיקסלים שונים שאינם חופפים.

2. ניהול הפיקסלים: בכדי לוודא שכל תהליכון אכן צובע פיקסלים שונים ולמנוע חפיפות, יצרנו מחלקה בשם Pixel, האחראית על בחירת הפיקסל הבא שיצבע.

3. סנכרון התהליכונים: הפונקציה המרכזית במחלקה Pixel היא nextPixel. פונקציה זו דואגת להקצות לכל תהליכון את הפיקסל הבא הזמין לצביעה, תוך שמירה על סנכרון ומניעת בעיות חפיפה.

באמצעות יישום שיפור זה, נוכל לחלק את עבודת עיבוד התמונה בין מספר תהליכונים הפועלים במקביל, מה שיביא לשיפור משמעותי בזמן הריצה ולניצול מיטבי של משאבי המעבד.

בס"ד

פונקציית nextPixel:

```
static Pixel nextPixel() {  1 usage  Benaya Trabelsi
    synchronized (mutexNext) {
        if (cRow == maxRows) return null;
        ++cCol;
        if (cCol < maxCols) return new Pixel(cRow, cCol);
        cCol = 0;
        ++cRow;
        if (cRow < maxRows) return new Pixel(cRow, cCol);
    }
    return null;
}
```

לאחר מכן, התאמנו את הפונקציה `renderImage` במחלקה `Camera` כך שתבצע רינדור באמצעות מספר התהליכונים שנבחרו.

```
public Camera renderImage() {  19 usages  Benaya Trabelsi
    int ny = imageWriter.getNy();
    int nx = imageWriter.getNx();
    Pixel.initialize(ny, nx, printInterval);

    if (threadsCount == 0) {
        for (int i = 0; i < ny; ++i)
            for (int j = 0; j < nx; ++j)
                castRays(nx, ny, j, i);
        return this;
    }
    List<Thread> threads = new LinkedList<>();
    int availableProcessors = threadsCount == -1 ? Runtime.getRuntime().availableProcessors()
            : threadsCount;

    for (int t = 0; t < availableProcessors; t++) {
        threads.add(new Thread(() -> {
            Pixel pixel;
            while ((pixel = Pixel.nextPixel()) != null)
                castRays(nx, ny, pixel.col(), pixel.row());
        }));
    }
    for (var thread : threads)
        thread.start();
    try {
        for (var thread : threads)
            thread.join();
    } catch (InterruptedException ignore) {
    }

    return this;
}
```

מקרא של בחירת מספר תהליכונים:

```
-2 auto, -1 range/stream, 0 no threads, 1+ number of threads
```

היררכיית גופים תוחמים (BVH)

תיאור

תיאור הבעיה:

בסצנה המכילה גופים רבים, חלקם קטנים או מורכבים מאוד, כל קרן בודקת חיתוכים עם כל הגופים. ברוב המקרים, לא יימצא חיתוך עם רוב הגופים.

הצעת הפתרון:

ניצור היררכיית גופים תוחמים המוגדרים כקופסאות סביב האובייקטים. בכל פעם נבדוק האם הקרן נחתכת עם הקופסאות (חישוב פשוט ומהיר לעומת חישוב מלא). אם לא נחתך, נעצור את הבדיקה; אם כן, נמשיך בהיררכיה לבדוק האם הקרן נחתכת עם האובייקטים עצמם.

עקרון הפעולה:

העיקרון מאחורי BVH הוא לבנות תיבות שעוטפות את הגופים ולבדוק חיתוך עם הגופים שבתוך התיבה שנחתכה ע"י הקרן. כל קופסה נמצאת בתוך קופסה גדולה יותר והחישוב נעשה באופן רקורסיבי.

תהליך יישום השיפור:

1. מחלקת BoundingBox: נוסיף מחלקה בשם `BoundingBox`. מחלקה זו תכיל שתי נקודות: נקודת מינימום ונקודת מקסימום, כך שלמעשה המחלקה מייצגת תיבה (ערכי ברירת מחדל – מינוס אינסוף ופלוס אינסוף בהתאמה).

2. שדה BoundingBox במחלקה Intersectable: במחלקה `Intersectable` נוסיף שדה `protected` מסוג `BoundingBox` שייצג את הקופסה התוחמת של הגוף. בכל אחד מהבנאים של הגופים הסופיים (כדור ומצולע) נחשב את גודל הקופסה ונשמור בשדה זה.

```
/**
 * The bounding box of the object.
 */
protected BoundingBox box;  5 usages
```

3. <u>בדיקת חיתוך עם BoundingBox:</u> נוסיף ל-`BoundingBox` מתודה `hasIntersection` כדי לבדוק האם יש חיתוכים של הקרן עם הקופסה. נשתמש במתודה זו בתוך `findGeoIntersections` של `Intersectable`. אם אין חיתוכים עם הקופסה, לא ננסה לחשב חיתוכים עם הגופים.

4. <u>בניית היררכיית BVH:</u> כדי לאחד את הגופים לקבוצות נוסיף ל-`BoundingBox` מתודה ציבורית סטטית `buildBVH`, שתתקבל רשימת גופים ותחזיר רשימה חדשה בצורת BVH.

לסיכום, הוספת היררכיית גופים תוחמים (BVH) משפרת את ביצועי הבדיקות של חיתוכי הקרניים עם האובייקטים על ידי צמצום מספר הבדיקות הנדרשות. באמצעות שיטת זו, נוכל לבצע בדיקות חיתוך בצורה יעילה ומהירה יותר.

נוסיף במחלקה Geometries מתודה קצרה לשמירת כל הגופים מחדש בצורה של BVH:

```
public void makeBVH() {   no usages   Benaya Trabelsi
    List<Intersectable> intersectables = BoundingBox.buildBVH(geometries);
    geometries.clear();
    geometries.addAll(intersectables);
}
```

לאחר בניית הסצנה בבדיקות והכנסת כל הגופים, נוכל לצמצם אותם ל-BVH וכך לחסוך זמן חישוב במהלך הרינדור. לשימוש ב-BVH נוסיף בהצהרת הבדיקה את ההצהרה הבאה:

```
scene.geometries.makeBVH();
```

מימוש האלגוריתם buildBVH:

1. הפרדת הגופים: נתחיל בהפרדת רשימת הגופים לשתי רשימות – גופים סופיים וגופים אינסופיים.

2. מיון הגופים: נמיין את הגופים הסופיים לפי סדר מסוים. בדוגמה הנוכחית, המיון נעשה לפי ציר X, אך ניתן למיין לפי פרמטרים אחרים בהתאם לצורך.

3. חלוקת הגופים: נחלק את רשימת הגופים הסופיים לשני חלקים שווים.

4. יצירת BVH חדש: עבור כל חלק ניצור BVH חדש, תוך חישוב גודל הקופסה לכל חלק בנפרד באמצעות רקורסיה.

5. איחוד הגופים: ניצור אובייקט `Geometries` חדש שיכיל את שני החלקים הקודמים ונחשב עבורו את גודל הקופסה.

6. חזרה לתוצאה: נאחד את רשימת הגופים האינסופיים עם הגופים הסופיים (המיוצגים כאובייקט `Geometries` שהוא BVH) ונחזיר את התוצאה.

נעזרנו בקוד של אריאל חלילי נ"י על מנת לבנות את המחלקה בצורה המיטבית:

```
1. /**
    * Creates a new bounding box that is the union of this
   bounding box and another.
    *
    * @param box the bounding box to union with
    * @return a new bounding box that encompasses both bounding
   boxes
    */
   private BoundingBox union(BoundingBox box) {
       return new BoundingBox(
                new Point(Math.min(min.getX(), box.min.getX()),
   Math.min(min.getY(), box.min.getY()), Math.min(min.getZ(),
   box.min.getZ())),
                new Point(Math.max(max.getX(), box.max.getX()),
   Math.max(max.getY(), box.max.getY()), Math.max(max.getZ(),
   box.max.getZ()))
       );
   }
```

```java
/**
 * Builds a bounding volume hierarchy (BVH) from a list of
intersectable geometries.
 *
 * @param intersectableList the list of intersectable
geometries
 * @return a list of intersectable geometries organized in a
BVH
 */
static public List<Intersectable> buildBVH(List<Intersectable>
intersectableList) {
    // If the list has one or zero elements, return it as is
    if (intersectableList.size() <= 1) {
        return intersectableList;
    }

    // Separate geometries without a bounding box
    List<Intersectable> infiniteGeometries = new ArrayList<>();
    intersectableList.removeIf(g -> {
        if (g.getBoundingBox() == null) {
            infiniteGeometries.add(g);
            return true;
        }
        return false;
    });

    // Sort geometries by the X coordinate of their bounding
box centers
    intersectableList.sort(Comparator.comparingDouble(g ->
g.getBoundingBox().getCenter().getX()));

    // Split the list into two halves and recursively build the
BVH
    int mid = intersectableList.size() / 2;
    List<Intersectable> leftGeometries =
buildBVH(intersectableList.subList(0, mid));
    List<Intersectable> rightGeometries =
buildBVH(intersectableList.subList(mid,
intersectableList.size()));

    // Create bounding boxes for the left and right halves
    BoundingBox leftBox = getBoundingBox(leftGeometries);
    BoundingBox rightBox = getBoundingBox(rightGeometries);

    // Combine the left and right geometries into a single
Geometries object
    Geometries combined = new Geometries(leftGeometries);
    combined.add(rightGeometries);
    combined.box = leftBox.union(rightBox);

    // Combine the infinite geometries and the BVH into a
single list
    List<Intersectable> result = new
ArrayList<>(infiniteGeometries);
    result.add(combined);
    return result;
}

// Calculates the bounding box for a list of intersect
geometries.
static private BoundingBox getBoundingBox(List<Intersectable>
```

```
intersectableList) {
    if (intersectableList.isEmpty()) {
        return null;
    }

    // Initialize the bounding box with the first geometry's
bounding box
    BoundingBox boundingBox =
intersectableList.get(0).getBoundingBox();
    for (Intersectable g : intersectableList) {
        boundingBox = boundingBox.union(g.getBoundingBox());
    }

    return boundingBox;
}
```

בס"ד

תוצאות סופיות x10001000:

תוצאות ללא שיפורים:

5 sec 615 ms

תוצאות עם אנטי-אליאסינג ברמה 9 (81 קרניים לכל פיקסל):

13 min 41 sec

תוצאות עם אנטי-אליאסינג ברמה 9 ועם תהליך רב-תהליכים (3 תהליכים):

3 min 52 sec

תוצאות עם אנטי-אליאסינג ברמה 9 ועם היררכיית גופים תוחמים (BVH):

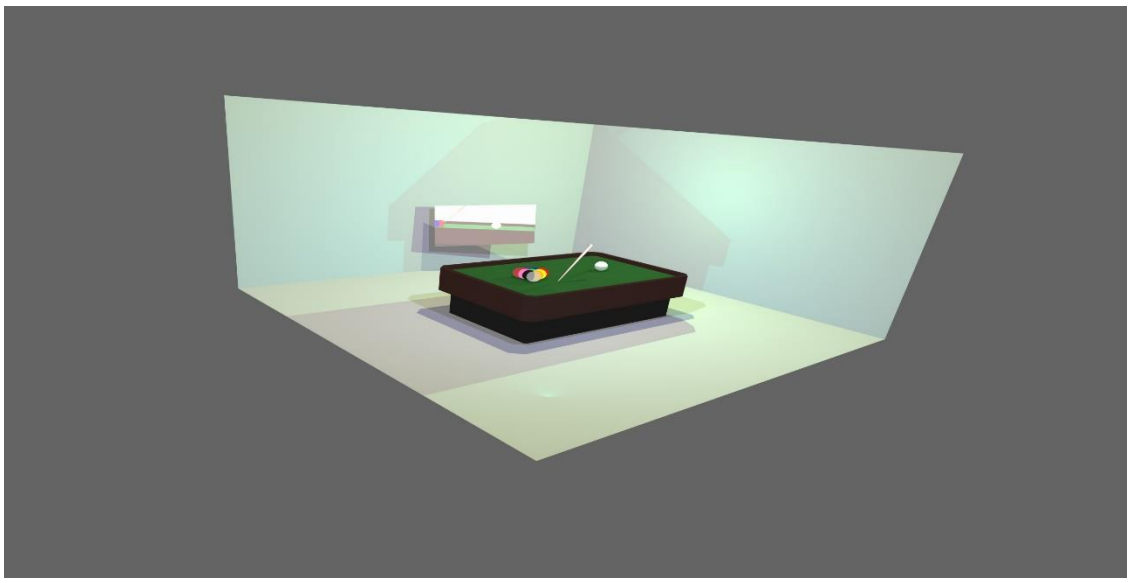
5 min

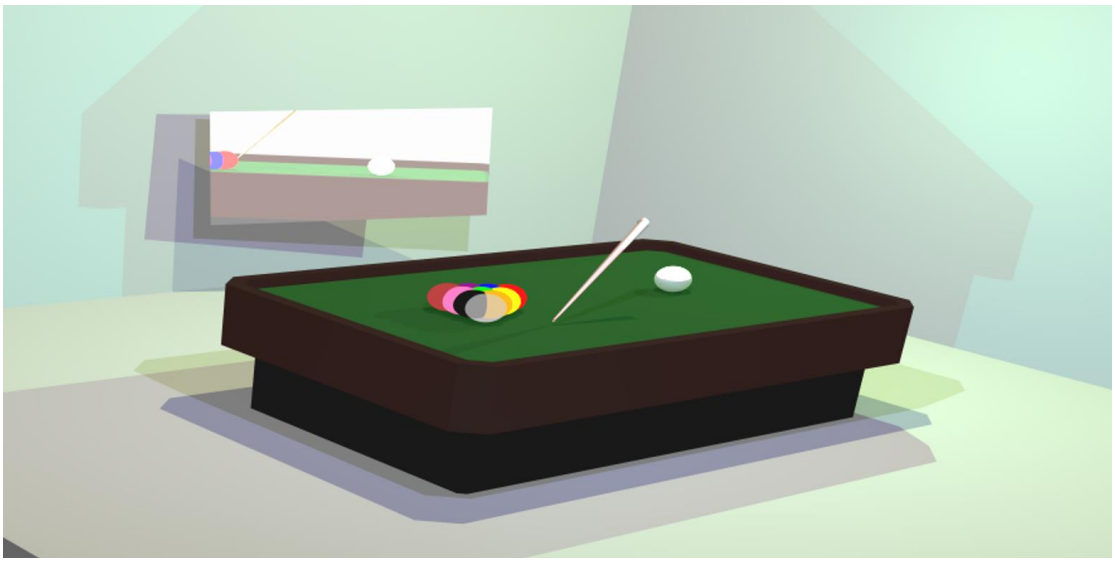
תוצאות עם אנטי-אליאסינג ברמה 9, תהליך רב-תהליכים (3 תהליכים) והיררכיית גופים תוחמים (BVH):

2 min 8 sec

בס"ד

תמונות נוספות:





בס"ד

רשימת בונוסים

1. נורמל לגליל סופי- 1 נק'
2. חיתוך עם מצולע- 1 נק'
3. ספוט ממוקד ("פנס")- 1 נק'
4. תמונה עם 10+ עצמים- 1 נק'
5. פתרון בעיית מרחק בהצללה בדרך 2- 1 נק'
6. Jitter – 2נק'
7. מצלמה על פי נקודה ווקטור- 1 נק'

לאורך כל הפרויקט נעזרנו במגוון חומרי עזר מהמודל, כולל מצגות הקורס התיאורטי והמעשי, וקבצי קוד לדוגמה. כמו כן, אנו מבקשים להודות למרצה היקר, מר אליעזר גינסבורגר, על העזרה, הליווי והתמיכה המתמשכת שלו לאורך כל הפרויקט.