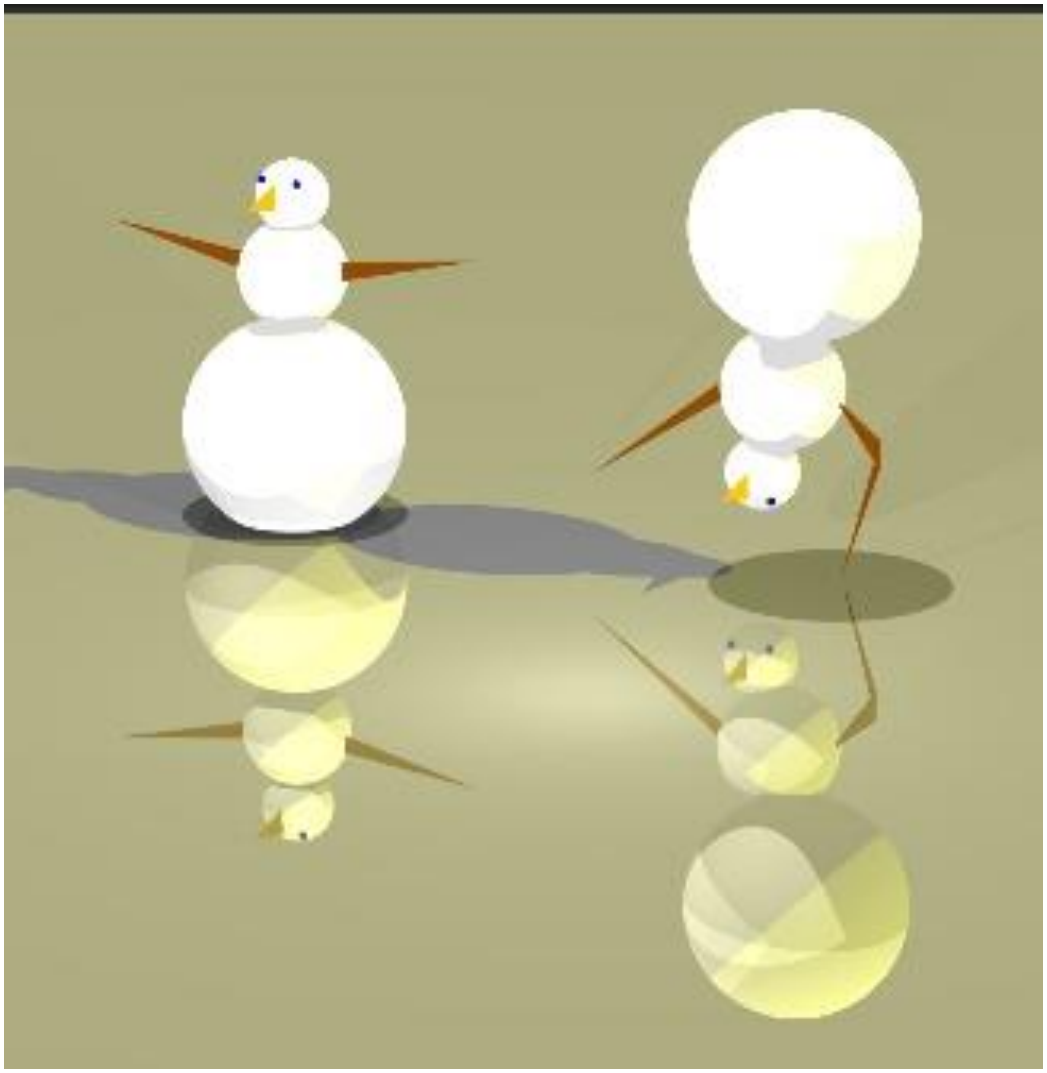
בס"ד

דו"ח פרויקט מפורט: שיפורי רינדור מתקדמים במעקב קרניים

מגישים: צבי ציפל ומשה טולדנו

הרב המרצה: הרב אליעזר גינסבורגר

התמונה בלי השדרוגים:



פרק 1: מבוא וארכיטקטורת המערכת

1.1. רקע כללי ומטרת הפרויקט

מטרתו המרכזית של פרויקט זה היא תכנון ומימוש של מערכת תוכנה ליצירת תמונות פוטוריאליסטיות באמצעות סימולציה של סצנה גרפית וירטואלית תלת-ממדית. התהליך, המכונה "מעקב קרניים" (Ray Tracing), מדמה את התנהגותן של קרני אור במרחב, החל ממקורות אור, דרך פגיעתן באובייקטים ועד הגעתן לעין המתבונן או למצלמה. הפרויקט כולו נכתב בשפת התכנות Java, תוך הקפדה על יישום עקרונות הנדסת תוכנה ותכנון מונחה עצמים (Object-Oriented Design) כפי שנלמדו במסגרת הקורס. דגש מיוחד הושם על יצירת קוד מודולרי, גמיש ובר-הרחבה, המאפשר הוספת תכונות ויכולות חדשות בצורה נקייה ומסודרת. הפיתוח לווה בכתיבת בדיקות יחידה (Unit Tests) מקיפות על מנת להבטיח את נכונות ותקינות הרכיבים המרכזיים במערכת.[1]

1.2. מבנה הפרויקט וחבילות מרכזיות

ארכיטקטורת הפרויקט מבוססת על חלוקה למספר חבילות (Packages), כאשר כל חבילה אחראית על תחום לוגי מוגדר. חלוקה זו מקדמת הפרדת עניינים (Separation of Concerns) ומאפשרת ניהול יעיל של הקוד. להלן פירוט החבילות המרכזיות במערכת:

- חבילת **primitives:** חבילה זו מהווה את הבסיס המתמטי והפיזיקלי של המערכת. היא מכילה את המחלקות הבסיסיות ביותר, המגדירות את המרחב התלת-ממדי ואת תכונות החומרים בו.[1]
  - Point3D: מייצגת נקודה במרחב התלת-ממדי באמצעות שלוש קואורדינטות (x,y,z).
  - Vector: מייצגת וקטור במרחב, המוגדר על ידי כיוון וגודל. משמשת לתיאור כיוונים, מהירויות ונורמלים.
  - Ray: מייצגת קרן אור. מורכבת מנקודת מוצא (Point3D) ווקטור כיוון (Vector), המגדירים יחדיו ישר אין-סופי בכיוון אחד.
  - Color: מייצגת צבע בפורמט RGB. המחלקה תומכת בפעולות אריתמטיות על צבעים, כגון חיבור, כפל בסקלר ומיצוע, שהן חיוניות לחישובי תאורה.
  - Material: מייצגת את תכונות החומר של אובייקט. היא מגדירה כיצד האובייקט מגיב לאור, וכוללת מקדמים לשקיפות (kT), החזרה (kR), החזרה דיפוזית (kD), החזרה ספקולרית (kS) ומידת הברק.
- חבילת **geometries:** חבילה זו מרכזת את כל הצורות הגאומטריות שניתן למקם בסצנה. כל הגאומטריות יורשות ממחלקה אבסטרקטית Geometry, אשר בתורה מממשת את הממשק Intersectable.
  - Intersectable: ממשק (interface) מרכזי במערכת, המגדיר התנהגות אחת ויחידה: היכולת למצוא נקודות חיתוך עם קרן נתונה. הממשק מכיל את הפונקציה findGeoIntersections(Ray ray).

- Geometry: מחלקה אבסטרקטית המספקת התנהגות משותפת לכל הגופים הגאומטריים, כגון החזקת חומר (Material) וצבע פליטה עצמי (emission). בנוסף, היא דורשת מכל מחלקה יורשת לממש את הפונקציה getNormal(Point3D point), המחזירה את הווקטור הנורמלי למשטח הגוף בנקודה נתונה. הנורמל חיוני לחישובי תאורה והחזרה.[1]
- מחלקות קונקרטיות: Sphere, Plane, Triangle, Polygon וכו'. כל אחת מממשת את לוגיקת חיתוך הקרן ואת חישוב הנורמל הספציפיים לה.

- חבילת **elements:** חבילה זו מכילה את הרכיבים הלא-גאומטריים של הסצנה, בעיקר את המצלמה ואת מקורות האור.
- Camera: מייצגת את נקודת המבט של הצופה. היא מוגדרת על ידי מיקום (0p) ושלושה וקטורים אורתוגונליים המגדירים את האוריינטציה שלה במרחב: vTo (כיוון ההסתכלות), vUp (כיוון "למעלה") ו-vRight (כיוון "ימינה").[1]
- LightSource: ממשק המגדיר את ההתנהגות של מקור אור. הוא כולל פונקציות לקבלת עוצמת האור בנקודה מסוימת (getIntensity) ואת כיוון האור מאותה נקודה (getL).[1]
- מחלקות קונקרטיות של תאורה: DirectionalLight (אור המגיע מכיוון אחיד מאינסוף, כמו שמש), PointLight (אור הבוקע מנקודה לכל הכיוונים) ו-SpotLight (פנס, אור הבוקע מנקודה לכיוון מוגדר בזווית מסוימת).

- חבילת **renderer:** זוהי החבילה המרכזית המבצעת את תהליך הרינדור עצמו.
- ImageWriter: מחלקה שתפקידה לכתוב את התמונה הסופית לקובץ. היא מקבלת מטריצה של צבעים (פיקסלים) ומייצרת ממנה קובץ תמונה בפורמט סטנדרטי (למשל, PNG או JPG).[1]
- RayTracerBase: ה"מוח" של תהליך הרינדור. מחלקה זו אחראית על חישוב הצבע הסופי עבור כל קרן. היא מבצעת את אלגוריתם מעקב הקרניים, הכולל חישוב תאורה מקומית (מודל Phong), הצללות, ואפקטים גלובליים כמו השתקפות ושקיפות.[1]

- חבילת **scene:** מחלקת Scene היא אגרגטור (aggregator) המרכז את כל רכיבי הסצנה תחת אובייקט אחד. היא מחזיקה את רשימת הגאומטריות, רשימת מקורות האור, המצלמה, צבע הרקע ומרחק המצלמה ממישור התצוגה.[1]

1.3. תהליך הרינדור הבסיסי

לפני הצגת השיפורים המתקדמים, חיוני להבין את תהליך הרינדור הבסיסי, כפי שמומש בשלבים המוקדמים של הפרויקט. תהליך זה מהווה את נקודת המוצא, והבנת מגבלותיו היא המניע לשיפורים שיפורטו בהמשך. התהליך עבור כל פיקסל בתמונה הסופית מתבצע באופן הבא:

1. בניית קרן ראשונית: עבור כל פיקסל על גבי מישור התצוגה (View Plane), מחלקת ה-Camera בונה קרן (Ray) אחת ויחידה. מקור הקרן הוא במיקום המצלמה, וכיוונה הוא דרך נקודת המרכז המדויקת של אותו פיקסל.
2. מציאת חיתוך: הקרן שנבנתה נשלחת אל הסצנה. המערכת עוברת על כל הגאומטריות בסצנה ומפעילה את פונקציית findGeoIntersections עבור כל אחת מהן. הפונקציה מחזירה רשימה של

כל נקודות החיתוך של הקרן עם אותה גאומטריה.

3. בחירת הנקודה הקרובה: מבין כל נקודות החיתוך שנמצאו מכל הגאומטריות, המערכת בוחרת את זו הקרובה ביותר למקור הקרן (למצלמה). אם לא נמצאו חיתוכים כלל, הפיקסל ייצבע בצבע הרקע של הסצנה.
4. חישוב צבע: במידה ונמצאה נקודת חיתוך, היא מועברת למחלקת RayTracerBase לחישוב הצבע. החישוב מתבסס על מודל התאורה של Phong, וכולל:
   - תאורה סביבתית (**Ambient**): תאורה בסיסית ואחידה המונעת מאזורים מוצלים להיות שחורים לחלוטין.
   - תאורה דיפוזית (**Diffuse**): מדמה את האופן שבו אור מתפזר באופן אחיד ממשטחים מחוספסים. עוצמתה תלויה בזווית בין הנורמל למשטח לבין כיוון האור.
   - תאורה ספקולרית (**Specular**): מדמה את הברק או ההחזר המבריק ממשטחים חלקים. עוצמתה תלויה בזווית בין כיוון ההחזרה המושלמת לבין כיוון הצופה.
   - הצללה (**Shadows**): לפני חישוב תרומתו של כל מקור אור, נשלחת "קרן צל" מנקודת החיתוך אל מקור האור. אם קרן זו נחסמת על ידי אובייקט אחר בדרך, אותו מקור אור לא יאיר את הנקודה.
5. כתיבת הפיקסל: הצבע הסופי שחושב מועבר למחלקת ImageWriter, אשר כותבת אותו למיקום המתאים במאגר הנתונים (buffer) של התמונה.

תהליך זה, על אף שהוא מייצר תמונה בסיסית, סובל ממגבלה מהותית הנובעת מהשלב הראשון: דגימת צבע אחת בלבד ממרכז הפיקסל. מגבלה זו מובילה לפגמים ויזואליים משמעותיים, כפי שיפורט בפרק הבא.

פרק 2: שיפור איכות התמונה: החלקת קצוות (**Anti-Aliasing**)

2.1. הבעיה: עיוות דגימה (**Aliasing**)

האלגוריתם הבסיסי שתואר לעיל מייצר פגם ויזואלי נפוץ הידוע בשם "עיוות דגימה" או Aliasing. מכיוון שצבעו של כל פיקסל נקבע על סמך קרן בודדת העוברת דרך מרכזו, נוצרת תמונה בעלת קצוות חדים, משוננים ולא טבעיים, המכונים לעיתים "מדרגות" הבעיה בולטת במיוחד באזורים של קווים אלכסוניים או בקצוות של צורות עגולות, שם המעבר החד בין צבע האובייקט לצבע הרקע יוצר מראה "מפוקסל".

הסיבה הטכנית לבעיה נעוצה בתהליך הדגימה. פיקסל אינו נקודה אינסופית, אלא שטח ריבועי. במציאות, פיקסל הממוקם על גבול של אובייקט עשוי לכסות בחלקו את האובייקט ובחלקו את הרקע. החלטה בינארית לקבוע את צבע הפיקסל כולו על סמך דגימה בודדת במרכזו מתעלמת ממידע זה וגורמת לאיבוד פרטים. כתוצאה מכך, "צורות שאינן ישרות יראו מחוספסות והחלוקה לפיקסלים תבלוט".

2.2. הפתרון: דגימת-על (**Super-Sampling**)

הפתרון המקובל לבעיית ה-Aliasing הוא טכניקה הנקראת "דגימת-על" (Super-Sampling, או SS). הרעיון המרכזי הוא להפסיק להתייחס לפיקסל כאל נקודה בודדת, ובמקום זאת, לדגום את צבעו במספר מיקומים שונים בתוך שטחו הריבועי. לאחר מכן, הצבע הסופי של הפיקסל נקבע כממוצע של כל צבעי הדגימות שנלקחו. בדרך זו, פיקסל הממוקם על גבול של אובייקט יקבל צבע ביניים, המהווה שילוב של צבע האובייקט וצבע הרקע. התוצאה היא "גבולות 'רכים' יותר" ומעברים הדרגתיים וחלקים יותר בין צבעים, המפחיתים משמעותית את המראה המשונן.

קיימות מספר גישות למימוש דגימת-על, כאשר שתי הנפוצות הן:

1. דגימת רשת (**Grid Sampling**): בשיטה זו, כל פיקסל מחולק באופן וירטואלי לרשת של "תת-פיקסלים" (למשל, רשת של 3×3, 5×5, וכו'). קרן נשלחת דרך מרכזו של כל תת-פיקסל ברשת. שיטה זו סימטרית ומדויקת, ומבטיחה כיסוי אחיד של שטח הפיקסל.
2. דגימה רנדומלית (**Random Sampling**): בשיטה זו, במקום רשת קבועה, הקרניים נשלחות למיקומים אקראיים בתוך שטח הפיקסל. שיטה זו יכולה לעיתים להפיק תוצאות טובות יותר בהמרת תבניות Aliasing לתבניות רעש, הנתפסות כפחות מפריעות לעין האנושית.

במסגרת פרויקט זה, נבחרה למימוש גישת דגימת הרשת, בשל פשטות המימוש והתוצאות העקביות והאיכותיות שהיא מפיקה.

2.3. מימוש (**Implementation**)

מימוש דגימת-העל דרש שינויים והוספת פונקציונליות חדשה, בעיקר במחלקת Camera ובמחלקת Renderer.

2.3.1. שינויים במחלקת **Camera**

כדי לתמוך בדגימת-על, נוספה למחלקת Camera פונקציה חדשה, constructRaysThroughPixelGrid, אשר מחליפה את הפונקציה המקורית constructRayThroughPixel. בניגוד לקודמתה שהחזירה קרן בודדת, פונקציה זו מחזירה רשימה של קרניים (List<Ray>) עבור פיקסל נתון. הפונקציה מקבלת, בנוסף לפרמטרים הרגילים, את מספר הדגימות הרצוי בכל ציר (numOfRaysInRowCol).

בס"ד

להלן קטע קוד המדגים את לוגיקת המימוש של הפונקציה, בהשראת המימוש המוצג ב-:[1]

Java

```java
/**
 * Calculates all the rays from the camera's starting point to a grid within a single pixel.
 * Instead of one ray through the center, this creates a grid of rays (e.g., 9x9).
 *
 * @param nX Number of pixels in the x-axis (columns)
 * @param nY Number of pixels in the y-axis (rows)
 * @param j Column index of the current pixel
 * @param i Row index of the current pixel
 * @param screenDistance Distance between the camera and the view plane
 * @param screenWidth The width of the view plane
 * @param screenHeight The height of the view plane
 * @param numOfRaysInRowCol The number of samples per row/column in the grid (e.g., 9 for an 81-ray grid)
 * @return A list of rays for the specified pixel
 */
public List<Ray> constructRaysThroughPixelGrid(int nX, int nY, int j, int i,
                                               double screenDistance, double screenWidth, double screenHeight,
                                               int numOfRaysInRowCol) {
    List<Ray> allTheRays = new ArrayList<>();
    Point3D Pc = _p0.add(_vTo.scale(screenDistance)); // Center of the view plane

    double Ry = screenHeight / nY; // Height of one pixel
    double Rx = screenWidth / nX;  // Width of one pixel

    double subPixelStep = 1.0d / numOfRaysInRowCol;

    // Iterate through the sub-pixel grid
    for (double rowOffset = 0; rowOffset < 1.0; rowOffset += subPixelStep) {
        for (double colOffset = 0; colOffset < 1.0; colOffset += subPixelStep) {
            Point3D Pij = new Point3D(Pc);

            // Calculate the precise coordinate for the sub-pixel
            double yi = ((i - nY / 2.0d) * Ry) + (rowOffset * Ry);
            double xj = ((j - nX / 2.0d) * Rx) + (colOffset * Rx);

            if (!isZero(xj)) {
                Pij = Pij.add(_vRight.scale(xj));
            }
            if (!isZero(yi)) {
```

```
Pij = Pij.add(_vUp.scale(-yi));
}

Vector Vij = Pij.subtract(_p0);
Ray myRay = new Ray(_p0, Vij);
allTheRays.add(myRay);
}
}
return allTheRays;
}
```

הפונקציה מחשבת תחילה את גודל הפיקסל (Rx, Ry). לאחר מכן, היא רצה בשתי לולאות מקוננות כדי לעבור על כל תת-פיקסל ברשת. עבור כל תת-פיקסל, היא מחשבת את מיקומו המדויק על מישור התצוגה ויוצרת קרן מהמצלמה אל נקודה זו.

2.3.2. שינויים במחלקת **Renderer**

במחלקת הרינדור הראשית, הלולאה המרכזית שונתה כדי להתמודד עם רשימת הקרניים במקום עם קרן בודדת. נוספה פונקציית עזר חדשה, calcPixelColorSuperSampled, האחראית על תהליך המיצוע.

Java

```
/**
* Calculates the final color for a pixel using super-sampling.
* It traces each ray in the list and averages the resulting colors.
*
* @param rays The list of rays generated for a single pixel.
* @return The final, averaged color for the pixel.
*/
private Color calcPixelColorSuperSampled(List<Ray> rays) {
Color accumulatedColor = Color.BLACK;
Color backgroundColor = _scene.getBackground();

for (Ray ray : rays) {
GeoPoint closestPoint = findClosestIntersection(ray);
if (closestPoint == null) {
```

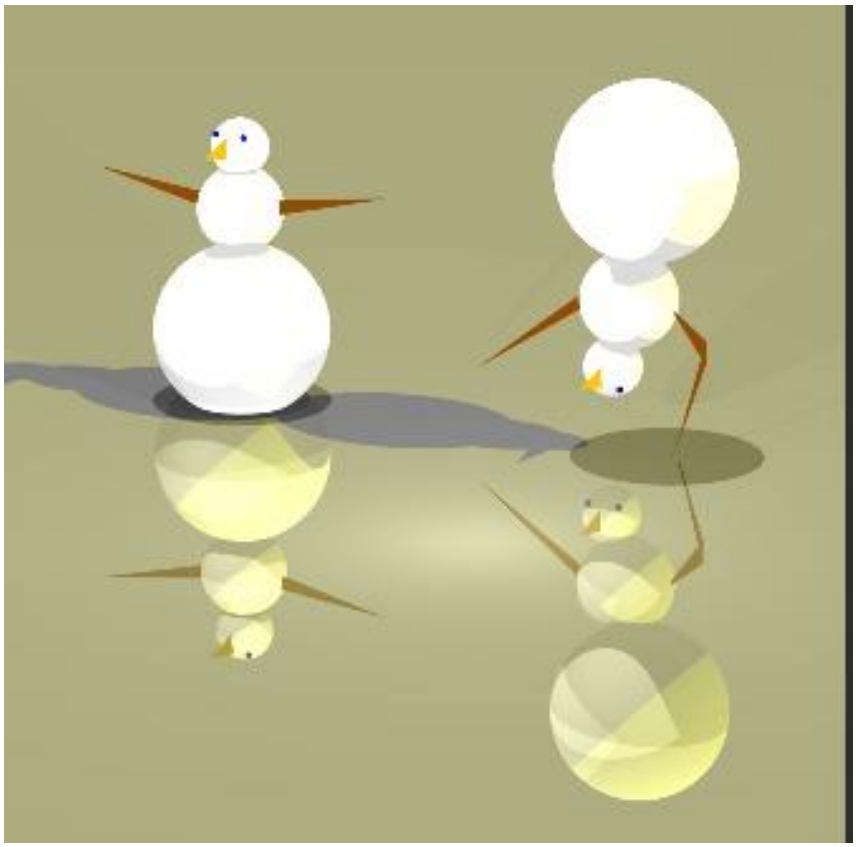
```
// If a ray hits nothing, it contributes the background color
accumulatedColor = accumulatedColor.add(backgroundColor);
} else {
// Otherwise, it contributes the calculated color of the intersection point
accumulatedColor = accumulatedColor.add(calcColor(closestPoint, ray));
}
}

// Average the colors by dividing by the total number of rays
return accumulatedColor.reduce(rays.size());
}
```

פונקציה זו מקבלת את רשימת הקרניים שנוצרה על ידי המצלמה. היא עוברת על כל קרן ברשימה, מוצאת את נקודת החיתוך הקרובה ביותר, ומחשבת את צבעה באמצעות פונקציית calcColor הקיימת. כל הצבעים המתקבלים נצברים, ובסוף התהליך, הצבע המצטבר מחולק במספר הקרניים הכולל כדי לקבל את הממוצע.[1]

2.4. תוצאות

התמונה עם Anti-Aliasing:



פרק 3: אופטימיזציה ושיפור ביצועים: דגימת-על אדפטיבית (**Adaptive Super-Sampling**)

3.1. הבעיה: עלות חישובית גבוהה

למרות השיפור הניכר באיכות התמונה, לטכניקת דגימת-העל הנאיבית (Naive Super-Sampling) שהוצגה בפרק הקודם יש חיסרון משמעותי: עלות חישובית גבוהה מאוד. שליחת עשרות, ולעיתים מאות, קרניים עבור כל פיקסל בתמונה, ללא קשר לתוכן שלו, היא בזבזנית ביותר. "באזור שבו צבע הפיקסלים אחיד, נוצר מצב שנשלחות קרניים רבות ללא צורך, מה שגורם לזמן ריצה גבוה מאוד".[1] לדוגמה, פיקסל הממוקם במרכזו של קיר לבן ואחיד אינו זקוק לדגימה צפופה; כל הקרניים שישלחו דרכו יחזירו את אותו הצבע. דגימת-על נאיבית מבצעת עבודה מיותרת זו עבור רוב הפיקסלים בתמונה, ומבזבזת משאבי עיבוד יקרים. הבעיה מחריפה ככל שרזולוציית התמונה ומספר הדגימות לפיקסל עולים, והופכת את זמני הרינדור לבלתי מעשיים עבור סצנות מורכבות.

3.2. הפתרון: דגימה מבוססת שונות

הפתרון לבעיית הביצועים הוא שימוש בגישה חכמה יותר: דגימת-על אדפטיבית (Adaptive Super-Sampling, או ASS). הרעיון המרכזי הוא לרכז את מאמץ הדגימה רק היכן שהוא נחוץ באמת – באזורים בעלי שונות צבע גבוהה, כלומר, בקצוות של אובייקטים, במרקמים מורכבים ובגבולות בין צל לאור. באזורים אחידים, המערכת תסתפק במספר דגימות קטן.

האלגוריתם פועל בדרך כלל בצורה רקורסיבית:

1. דגימה ראשונית: עבור פיקסל נתון, מתחילים בדגימה מינימלית, למשל, שליחת קרניים דרך ארבע פינות הפיקסל ומרכזו.
2. בדיקת שונות: המערכת בודקת את הצבעים שהתקבלו מהדגימות. אם כל הצבעים דומים מאוד זה לזה (ההפרש ביניהם קטן מסף (threshold) שנקבע מראש), ההנחה היא שהפיקסל מכסה אזור אחיד. במקרה זה, הרקורסיה נעצרת, וצבע הפיקסל נקבע כממוצע של הדגימות המעטות שבוצעו.
3. חלוקה רקורסיבית: אם נמצאה שונות משמעותית בין הצבעים, המשמעות היא שהפיקסל נמצא על גבול או אזור מורכב. במקרה זה, הפיקסל מחולק באופן וירטואלי לארבעה רבעים (תת-פיקסלים), והאלגוריתם נקרא באופן רקורסיבי עבור כל אחד מהרבעים הללו.
4. תנאי עצירה נוסף: כדי למנוע רקורסיה אינסופית, מוגדר עומק רקורסיה מרבי. אם מגיעים לעומק זה, התהליך נעצר והצבע מחושב על סמך הדגימות שבוצעו עד כה.

בדרך זו, המערכת "מתאימה" (adapts) את צפיפות הדגימה לתוכן התמונה, וחוסכת כמות עצומה של

חישובים מיותרים.

3.3. מימוש (**Implementation**)

מימוש ASS הוא מורכב יותר מזה של SS רגיל ודורש הוספת לוגיקה רקורסיבית למערכת.

3.3.1. פונקציית ניהול רקורסיה

בדומה למימושים הקודמים, נוספה פונקציית כניסה חדשה, renderImageAdaptiveSuperSampling, אשר קוראת לפונקציית עזר לכל פיקסל. פונקציית העזר, castBeamAdaptiveSuperSampling, מכינה את הקריאה הרקורסיבית הראשונה. היא בונה את הקרן המרכזית וקוראת לפונקציה הרקורסיבית הראשית.[1]

Java

```
/**
 * Initiates the adaptive super-sampling process for a single pixel.
 * It calls the main recursive function to calculate the pixel's color.
 * @param j Column index
 * @param i Row index
 * @return The final color for the pixel, calculated adaptively.
 */
private Color castBeamAdaptiveSuperSampling(int j, int i) {
    // Construct the pixel's boundaries and initial parameters for the recursion
    double pixelWidth = _width / _imageWriter.getNx();
    double pixelHeight = _height / _imageWriter.getNy();

    // Calculate the center point of the pixel on the view plane
    Point Pij_center =... // Logic to find the center point of pixel (j,i)

    // Start the recursive calculation
    return calcAdaptiveSuperSamplingRec(Pij_center, pixelWidth, pixelHeight,
                                        MAX_RECURSION_DEPTH);
}
```

3.3.2. הפונקציה הרקורסיבית

לב המימוש הוא הפונקציה הרקורסיבית, calcAdaptiveSuperSamplingRec. פונקציה זו מקבלת את מרכז האזור הנוכחי (שיכול להיות פיקסל שלם או תת-פיקסל), את מידותיו, ואת רמת הרקורסיה הנוכחית.

להלן הקוד המדגים את הלוגיקה הרקורסיבית:

Java

```
/**
 * Recursively calculates the color of a pixel area using adaptive super-sampling.
 *
 * @param centerPoint The center of the current area (pixel or sub-pixel).
 * @param width The width of the current area.
 * @param height The height of the current area.
 * @param level The current recursion depth.
 * @return The calculated color for the area.
 */
private Color calcAdaptiveSuperSamplingRec(Point3D centerPoint, double width, double height, int level)
{
    // --- Stopping Condition 1: Max recursion depth reached ---
    if (level == 0) {
        Ray centerRay = new Ray(_p0, centerPoint.subtract(_p0));
        return _rayTracerBase.traceRay(centerRay);
    }

    // Sample the four corners of the current area
    Point3D corner1 = centerPoint.add(_vRight.scale(-width / 2)).add(_vUp.scale(height / 2));
    Point3D corner2 = centerPoint.add(_vRight.scale(width / 2)).add(_vUp.scale(height / 2));
    Point3D corner3 = centerPoint.add(_vRight.scale(-width / 2)).add(_vUp.scale(-height / 2));
    Point3D corner4 = centerPoint.add(_vRight.scale(width / 2)).add(_vUp.scale(-height / 2));

    Color color1 = _rayTracerBase.traceRay(new Ray(_p0, corner1.subtract(_p0)));
    Color color2 = _rayTracerBase.traceRay(new Ray(_p0, corner2.subtract(_p0)));
    Color color3 = _rayTracerBase.traceRay(new Ray(_p0, corner3.subtract(_p0)));
    Color color4 = _rayTracerBase.traceRay(new Ray(_p0, corner4.subtract(_p0)));
```

```
// --- Stopping Condition 2: Colors are similar ---
if (color1.isSimilar(color2) && color1.isSimilar(color3) && color1.isSimilar(color4)) {
    return color1; // Or an average of the four
}

// --- Recursive Step: Colors are different, so recurse on sub-quadrants ---
double halfWidth = width / 2;
double halfHeight = height / 2;

// Calculate centers of the four sub-quadrants
Point3D center1 = centerPoint.add(_vRight.scale(-width / 4)).add(_vUp.scale(height / 4));
Point3D center2 = centerPoint.add(_vRight.scale(width / 4)).add(_vUp.scale(height / 4));
Point3D center3 = centerPoint.add(_vRight.scale(-width / 4)).add(_vUp.scale(-height / 4));
Point3D center4 = centerPoint.add(_vRight.scale(width / 4)).add(_vUp.scale(-height / 4));

Color c1 = calcAdaptiveSuperSamplingRec(center1, halfWidth, halfHeight, level - 1);
Color c2 = calcAdaptiveSuperSamplingRec(center2, halfWidth, halfHeight, level - 1);
Color c3 = calcAdaptiveSuperSamplingRec(center3, halfWidth, halfHeight, level - 1);
Color c4 = calcAdaptiveSuperSamplingRec(center4, halfWidth, halfHeight, level - 1);

// Combine results from recursive calls
return c1.add(c2, c3, c4).reduce(4);
}
```

3.3.3. שיקולי יעילות מתקדמים: מניעת חישובים כפולים

מימוש רקורסיבי כפי שתואר לעיל, על אף יעילותו, עדיין סובל מבעיה פוטנציאלית: חישובים כפולים. כאשר האלגוריתם פועל על שני רבעי-פיקסל סמוכים, הוא עלול לחשב את צבען של קרניים העוברות דרך הגבול המשותף שלהם פעמיים – פעם אחת עבור כל קריאה רקורסיבית. מכיוון שחישוב צבע של קרן (traceRay) הוא הפעולה היקרה ביותר במערכת, הימנעות מחישובים כפולים אלו יכולה להוביל לשיפור ביצועים נוסף ומשמעותי.[1]

פתרון אלגנטי לבעיה זו הוא שימוש בטכניקת **Memoization** (או Caching). הרעיון הוא לשמור את תוצאות החישוב של כל קרן במבנה נתונים, כגון Map, הממפה בין אובייקט Ray (או ייצוג ייחודי שלו) לאובייקט Color המתאים לו. לפני ביצוע חישוב traceRay יקר, המערכת תבדוק תחילה אם צבעה של

הקרן כבר חושב ונשמר במטמון. אם כן, הערך יישלף ישירות מהמטמון; אם לא, החישוב יתבצע, ותוצאתו תישמר במטמון לשימוש עתידי.

דו"ח לדוגמה מתאר גישה זו: "יצרנו מחלקה חדשה ColorRay שמחזיקה Color ו-Ray ביחד... ולכן בכל פעם שנשתמש בקרן שכבר השתמשנו בה לא נצטרך לבדוק את הצבע שלה כי ברגע שחשבנו אותו בפעם הראשונה הוא כבר נשמר. שיפור זה לבד חסך כ-50% זמן ריצה".[1] הכללת אופטימיזציה כזו מדגימה חשיבה מעמיקה על יעילות אלגוריתמית והבנה של צווארי הבקבוק החישוביים במערכת.

3.4. תוצאות וניתוח ביצועים

התוצאה הוויזואלית של שימוש ב-Adaptive Super-Sampling זהה כמעט לחלוטין לזו של Super-Sampling רגיל עם מספר דגימות גבוה. שתי השיטות מפיקות תמונה חלקה ואיכותית. עם זאת, ההבדל הדרמטי טמון בזמן הריצה.

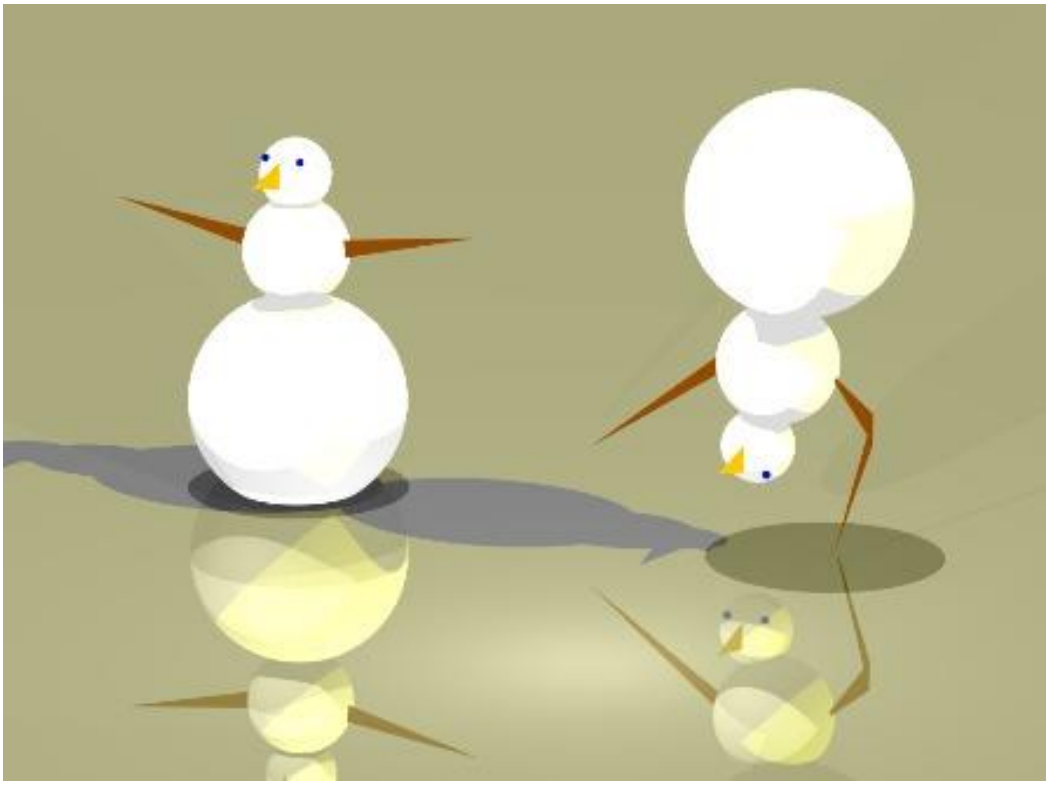
כדי לכמת את שיפור הביצועים, נערכה השוואה בין שלוש שיטות הרינדור על אותה סצנה באותה רזולוציה. התוצאות מוצגות בטבלה הבאה.

טבלה 1: השוואת ביצועי שיטות רינדור

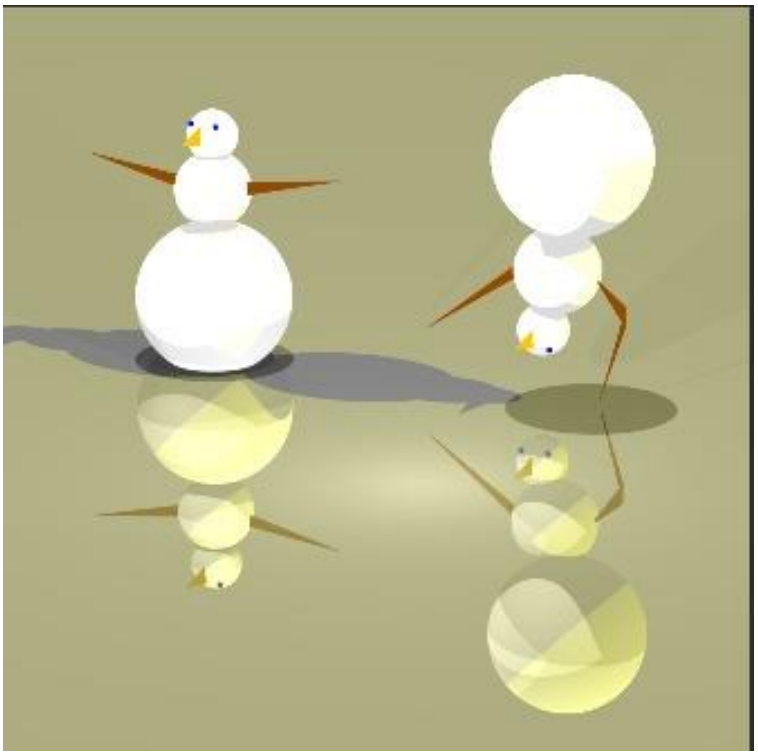
שיטת רינדור (Rendering Method)	זמן ריצה (Render Time)	איכות ויזואלית (Visual Quality)	הערות (Notes)
ללא החלקה (No Anti-Aliasing)	22 שניות	קצוות חדים ומפוקסלים	רינדור מהיר אך באיכות נמוכה
דגימת-על (Super-Sampling 9x)	28 דקות ו-15 שניות	קצוות חלקים, איכות גבוהה	איכות גבוהה אך זמן ריצה ארוך מאוד ולא יעיל
דגימת-על אדפטיבית (ASS)	1 דקה ו-12 שניות	קצוות חלקים, איכות גבוהה	איכות זהה לדגימת-על אך בשיפור ביצועים של פי ~23

הנתונים בטבלה מדגימים באופן חד משמעי את כוחה של הגישה האדפטיבית. היא מצליחה לספק את אותה איכות ויזואלית גבוהה של דגימת-על מלאה, אך בזמן ריצה קצר משמעותית, מה שהופך אותה לפתרון המעשי והיעיל ביותר להשגת תמונות איכותיות. הנתונים תומכים בממצאים דומים מדו"חות אחרים, המצביעים על שיפורי ביצועים בסדרי גודל של פי 18 ויותר.

תמונה עם Adaptive Super-Sampling ובלי Anti-Aliasing:



תמונת הדגמה סופית עם ASS ו-AA:



בתמונה הסופית ניתן להבחין בבירור בתוצאות העבודה. הקצוות של כל האובייקטים, נראים חלקים וטבעיים. ניתן לראות בבירור את ההשתקפות. משחקי האור והצל על הרצפה, הנובעים ממקורות האור, מעניקים לסצנה עומק ותחושת ריאליזם. התוצאה הסופית מדגימה את יכולתה של המערכת לייצר תמונות מורכבות ואיכותיות בזמן ריצה סביר, הודות לשילוב בין אלגוריתמים גרפיים מתקדמים וטכניקות אופטימיזציה יעילות.

פרק 5: סיכום

5.1. סיכום

במסגרת פרויקט זה תוכננה ומומשה מערכת מתקדמת למעקב קרניים בשפת Java. הפרויקט התמקד לא רק במימוש האלגוריתם הבסיסי, אלא גם בפיתוח והטמעה של שני שיפורים מרכזיים שנועדו להתמודד עם אתגרים מהותיים בתחום הגרפיקה הממוחשבת.

השיפור הראשון, דגימת-על (**Super-Sampling**), נתן מענה לבעיית ה-Aliasing (עיוות דגימה), אשר יצרה תמונות בעלות קצוות משוננים ולא מציאותיים. על ידי דגימת מספר רב של קרניים בכל פיקסל ומיצוע צבעיהן, הושגה החלקה משמעותית של קצוות האובייקטים, והתקבלה תמונה איכותית ונעימה יותר לעין.

עם זאת, שיפור זה בא על חשבון זמני ריצה ארוכים מאוד. כדי לפתור בעיית ביצועים זו, מומש השיפור השני והמשמעותי יותר: דגימת-על אדפטיבית (**Adaptive Super-Sampling**). אלגוריתם רקורסיבי זה ממקד את מאמץ הדגימה רק באזורים הבעייתיים בתמונה (קצוות ומרקמים מורכבים), תוך ביצוע דגימה מינימלית באזורים אחידים. כפי שהודגם בניתוח הביצועים, גישה זו סיפקה תוצאה ויזואלית זהה באיכותה לדגימת-על מלאה, תוך קיצוץ דרמטי בזמן הרינדור והפיכתו למעשי.

הפרויקט מדגים בהצלחה את השילוב בין הבנה תיאורטית של אלגוריתמים גרפיים לבין יישומם המעשי תוך שימוש בעקרונות הנדסת תוכנה נכונים. התוצר הסופי הוא מערכת מודולרית וגמישה, המסוגלת להפיק תמונות פוטוריאליסטיות מורכבות ואיכותיות ביעילות חישובית.