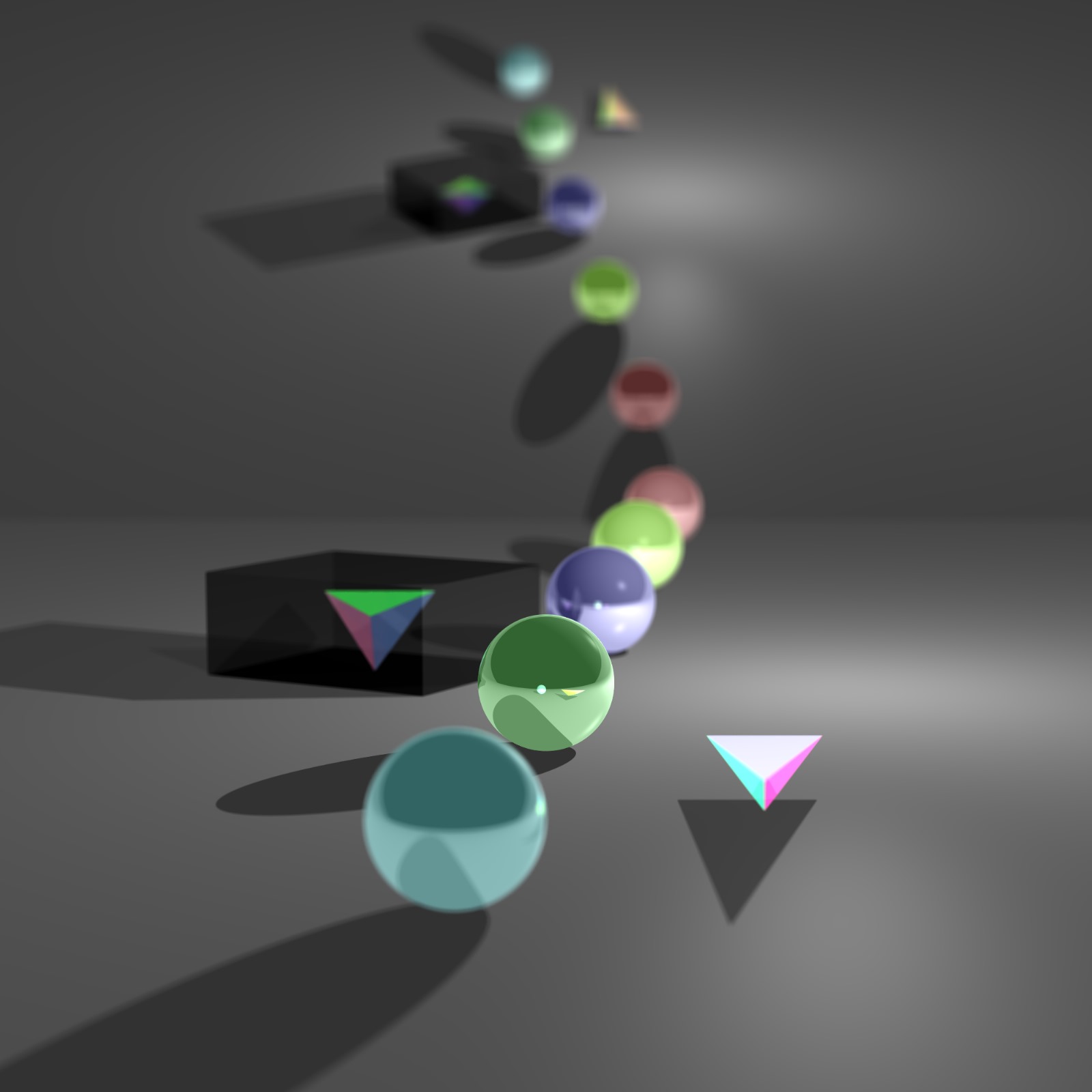
**דו"ח למיני פרויקט במבוא להנדסת תכנה**



**מגישים:**

**אשר אבן צור, ת.ז: 215340118**

**שמעון חקשור, ת.ז: 328308721**

**תוכן עניינים**

[מיני פרויקט 1 3](#_Toc202217963)

[דגימה עם רעש (Jittered Sampling) 3](#_Toc202217964)

[מימוש בקוד 4](#_Toc202217965)

[עומק השדה (Depth of Field / DOF) 7](#_Toc202217966)

[כיצד ממומש בפועל? 7](#_Toc202217967)

[מימוש בקוד 8](#_Toc202217968)

[השוואת תמונה ללא האפקט לתמונה עם האפקט 10](#_Toc202217969)

[החלקת עקומות (Anti-Aliasing / AA) 11](#_Toc202217970)

[כיצד ממומש בפועל? 11](#_Toc202217971)

[מימוש בקוד 12](#_Toc202217972)

[השוואת תמונה ללא האפקט לתמונה עם האפקט 15](#_Toc202217973)

[השוואת תמונה ללא האפקט לתמונה עם האפקט, ממבט קרוב 16](#_Toc202217974)

[שילוב AA יחד עם DOF 17](#_Toc202217975)

[כיצד ממומש בפועל? 17](#_Toc202217976)

[מימוש בקוד 17](#_Toc202217977)

[הדגמה של השילוב בין האפקטים AA ו DOF 20](#_Toc202217978)

[מיני פרויקט 2 21](#_Toc202217979)

[ריבוי דגימות מסתגל (Adaptive Super Sampling) 21](#_Toc202217980)

[כיצד ממומש בפועל? 21](#_Toc202217981)

[מימוש בקוד 23](#_Toc202217982)

[מימוש ריבוי דגימות מסתגל עבור אפקט החלקת עקומות 23](#_Toc202217983)

[מימוש ריבוי דגימות מסתגל עבור אפקט עומק השדה 26](#_Toc202217984)

[השוואת תמונה ללא ההאצה לתמונה עם ההאצה 30](#_Toc202217985)

[השוואת תמונה ללא ההאצה לתמונה עם ההאצה, ממבט קרוב 31](#_Toc202217986)

[הדגמת השיפור בזמני הריצה בעקבות ההאצה 32](#_Toc202217987)

[רשימת בונוסים שעשינו 35](#_Toc202217988)

# **מיני פרויקט 1**

## **דגימה עם רעש (Jittered Sampling)**

השיפורים במיניפ 1 (שאנחנו הוספנו 2 מתוכם לפרויקט שלנו) מבוססים עלsuper sampling משמע החלפה של נקודה מסוימת באוסף נקודות שיאפשר לנו לבצע מספר רב של דגימות שיחליפו את הדגימה הבודדת שבוצעה עבור הנקודה המקורית, התהליך הזה מאפשר שיפור משמעותי של איכות התמונה (כמו למשל בשיפור של anti-aliasing) או במקרים מסוימים אפקט חדש לחלוטין שכלל לא היה אפשרי קודם לכן (כמו למשל בשיפור של עומק השדה).

את הדגימה המרובה אנחנו בחרנו לבצע ע"י Jittered sampling כלומר דגימות מרקדות.

**כיצד ממומש בפועל?**

ראשית נוסיף שדה boardShape שעל פיו נקבע אם אזור הדגימה יהיה בצורת של עיגול או ריבוע.

שלבי האלגוריתם:

1. מחלקים את איזור הדגימה ל numOfSamples אזורים כאשר numOfSamples מייצג את מספר הדגימות שנרצה לקחת (זוהי חלוקה לוגית בפועל בקוד לא באמת עושים חלוקה אלא פשוט דואגים שהתחומים שייקבעו ל"ריקוד" של הנקודה יהיו תואמים לאזור כזה)
2. מאתרים את מרכז האזור.
3. על סמך ערכים רנדומליים מאפשרים לנקודה "לרקד" בתחומי האזור שנקבע לה, כלומר מזיזים את הנקודה ממרכז האזור על ציר איקס ועל ציר וואי על סמך ערכים רנדומליים באופן כזה שתשאר בתחומי האזור ולא תצא ממנו.

חשוב לציין:

* מספר האזורים הוא כמספר הנקודות שברצוננו לדגום.
* מכל אזור נלקחת נקודת דגימה אחת בלבד.
* האזורים זרים זה לזה כלומר אין חפיפות ביניהם.
* כל האזורים מוכלים בשטח הדגימה המקורי.
* איחוד האזורים שווה לשטח הדגימה המקורי.

התנאים הנ"ל מאפשרים פיזור אחיד של הדגימות על פני שטח הדגימה תוך כדי שימוש בערכים רנדומליים לקביעת מיקום הדגימות, מה שהופך את האלגוריתם הזה לשילוב האולטימטיבי של שני האלגוריתמים האחרים לבניית אוסף נקודות הדגימה (רשת רגילה ללא רנדומליות, רשת מבוססת רנדומליות מלאה ללא שמירה על פיזור אחיד)

### **מימוש בקוד**

*/\*\*  
 \* Generates jittered samples on a board (square or disk) using polar mapping of a jittered grid.  
 \*  
 \* @param center center of the disk  
 \* @param vRight right-direction vector (X axis on disk)  
 \* @param vUp up-direction vector (Y axis on disk)  
 \* @param radius if boardShape is SQUARE, this is half the side length of the square,  
 \* If boardShape is CIRCLE, this is the radius of the disk.  
 \* @param numPoints number of jittered points (should be square number, like 81)  
 \* @param boardShape shape of the board (SQUARE or CIRCLE)  
 \* @return list of 3D points on the disk  
 \*/*public static List<Point> generateJitteredSamples(Point center, Vector vRight, Vector vUp, double radius, int numPoints, BoardShape boardShape) {  
 List<Point> points = new ArrayList<>();  
 int gridSize = (int) Math.*sqrt*(numPoints);  
  
 if (gridSize \* gridSize != numPoints)  
 throw new IllegalArgumentException("numPoints must be a perfect square (e.g. 81, 100, 121)");  
  
 for (int i = 0; i < gridSize; i++) {  
 for (int j = 0; j < gridSize; j++) {  
  
 switch (boardShape) {  
 case *SQUARE* -> {  
 // Jittered sample in [-1,1]^2 square  
 double x = ((i + *random*.nextDouble()) / gridSize) \* 2 - 1;  
 double y = ((j + *random*.nextDouble()) / gridSize) \* 2 - 1;  
  
 Point p = center;  
 if (!*isZero*(x)) p = p.add(vRight.scale(x \* radius));  
 if (!*isZero*(y)) p = p.add(vUp.scale(y \* radius));  
 points.add(p);  
 }  
  
 case *CIRCLE* -> {  
 // Compute uniform jittered samples in polar coordinates inside the unit disk  
  
 // u, v are jittered normalized coordinates in [0,1) along grid axes  
 double u = (i + *random*.nextDouble()) / gridSize;  
 double v = (j + *random*.nextDouble()) / gridSize;  
  
 // r is radius for point inside disk,  
 // Using r = u will not cover the entire disk uniformly,  
 // since the area of each "layer" increases with r.  
 // r = sqrt(u) ensures uniform distribution,  
 // by "pushing" the points away from the center, compared to a linear distribution (r = u).  
 double r = Math.*sqrt*(u);  
  
 // theta is angle, distributed uniformly in [0, 2\*PI)  
 double theta = 2 \* Math.*PI* \* v;  
  
 // Convert polar coordinates (r, theta) to Cartesian coordinates (x,y) on unit disk  
 double x = r \* Math.*cos*(theta);  
 double y = r \* Math.*sin*(theta);  
  
 // Scale (x,y) by radius of the disk and translate by center along vRight and vUp axes  
 Point p = center;  
 if (!*isZero*(x)) p = p.add(vRight.scale(x \* radius));  
 if (!*isZero*(y)) p = p.add(vUp.scale(y \* radius));  
  
 points.add(p);  
 }  
 }  
 }  
 }  
  
 return points;  
}

## **עומק השדה (Depth of Field / DOF)**

הוספת אפקט של עומק השדה משמעו יצירת מצב שבו מסביב למצלמה ישנו מעין כדור לוגי שהמצלמה היא המרכז שלו וכל הנקודות על אותו כדור הן נקודות פוקוס שחדות התמונה בהן תהיה זהה לחדות המקורית לפני הוספת האפקט, חדות התמונה בכל נקודה אחרת במרחב תלויה במרחקה מנקודת הפוקוס הקרובה ביותר אליה – ככל שהנקודה רחוקה יותר מנקודת הפוקוס הקרובה ביותר אליה כך חדות התמונה בה תהיה חלשה יותר.

בין היתר הוספת האפקט הזה מחזקת אצל הצופה בתמונה את ההרגשה שהוא צופה במיצג תלת ממדי בעל עומק.

### **כיצד ממומש בפועל?**

ראשית נוסיף מספר שדות הכרחיים:

1. useDOF – שדה שייקבע האם יהיה שימוש ב dof בתמונה הנוכחית.
2. amountOfRays\_DOF – כמות הקרניים שנירה עבור כל פיקסל כדי ליצור את האפקט (יוסבר בהמשך).
3. apertureRadius – רדיוס הצמצם שמייצג בעצם את רדיוס בסיס אלומת הקרניים אשר תשלח מאזור המצלמה על מנת ליצור את האפקט (יוסבר בהמשך), ככל שנגדיל אותו כך תהיה ירידה דרסטית יותר בחדות התמונה עם ההתרחקות מהכדור שמייצג את אוסף נקודות הפוקוס.
4. depthOfField – שדה לייצוג עומק השדה כערך מספרי, בפשטות זהו הרדיוס של כדור נקודות הפוקוס שהמצלמה ניצבת במרכזו, ככל שנגדיל אותו כך הפוקוס שלנו יהיה עמוק יותר בתוך הסצינה.

האלגוריתם לאפקט עומק השדה מתבצע במספר שלבים שמבוצעים עבור כל פיקסל בנפרד:

1. נמצא את נקודת הפוקוס עבור הפיקסל הנוכחי עי כך שנתקדם depthOfField יחידות ממיקום המצלמה על הוקטור שמכוון ממיקום המצלמה אל מרכז הפיקסל.
2. נשתמש ב Jittered sampling כדי ליצור אוסף נקודות דגימה שמרכזו יהיה במיקום המצלמה.
3. נירה קרניים מכל הנקודות שבאוסף נקודות הדגימה לכיוון נקודת הפוקוס, התוצאה: אלומת קרניים בצורת שעון חול שמרכז בסיסו הוא מיקום המצלמה, צוואר הבקבוק הוא בנקודת הפוקוס והקרן המקורית לפני השיפור הייתה חותכת את שעון החול הזה לאורך ועוברת דרך המרכזים של שני בסיסיו.
4. נבצע מעקב קרניים עבור כל אחת מהקרניים שיצרנו בשלב 3 כדי למצוא את הצבע שבמופק ממנה.
5. נמצא את הממוצע של הצבעים אשר קיבלנו בשלב 4.
6. צבענו את הפיקסל בצבע שקיבלנו בשלב 5.

התהליך הזה תואם להיגיון שהצגנו – ככל שהגופים שבהם פוגעות הקרניים שיצרנו בשלב 3 רחוקים יותר מנקודת הפוקוס מבחינת העומק, כך נקודות הפגיעה יתפזרו על שטח נרחב יותר (בגלל צורת שעון החול של הקרניים) וכך הצבע הסופי של הפיקסל יורכב מממוצע צבעים של שטח נרחב יותר בסצנה.

התוצאה בתמונה הסופית היא אפקט של מעבר הדרגתי בצבעים שיתפרש על שטח רחב יותר ככל שהעומק של אזור הפגיעה הוא רחוק יותר מהכדור שמייצג את אוסף נקודות הפוקוס.

נשים לב שאם נקודת הפוקוס עבור פיקסל מסוים מונחת בדיוק על אובייקט בסצנה אזי כל פגיעות של הקרניים שניצור עבור הפיקסל הזה יפגעו באותה נקודה ואז נעשה ממוצע של צבעים זהים שהתוצאה שלו תהיה גם כן אותו הצבע וכך נקבל את אותו הצבע בדיוק שהיינו מקבלים במצב הרגיל ללא שום כלומר לא יהיה מעבר הדרגתי כלל שזה בדיוק תואם למראה שאנחנו צריכים לראות עבור קטע הסצנה שמונח בדיוק על הכדור שמייצג את אוסף נקודות הפוקוס.

### **מימוש בקוד**

*/\*\*  
 \* Calculate the averaged color for the sampled area using non-adaptive super sampling for Depth of Field (DOF).  
 \* Based on a given direction vector.  
 \*  
 \* @param direction the direction vector for the DOF rays  
 \* @return the averaged color for the sampled area using non-adaptive super sampling for Depth of Field (DOF).  
 \*/*private Color nonAdaptiveDofColor(Vector direction) {  
 ArrayList<Ray> DOFrays = constructDOFRays(direction);  
 return averageRays(DOFrays);  
}

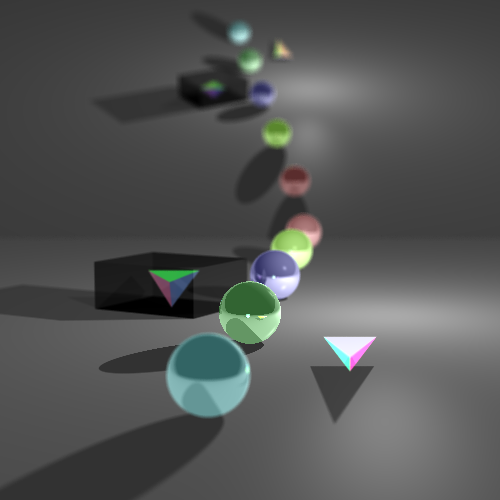
*/\*\*  
 \* Constructs Depth of Field (DOF) rays in a specific direction.  
 \* This method is used when Depth of Field (DOF) effect is enabled.  
 \*  
 \* @param direction the direction vector for the DOF rays  
 \* @return a list of DOF rays  
 \*/*private ArrayList<Ray> constructDOFRays(Vector direction) {  
 ArrayList<Ray> DOFRays = new ArrayList<>();  
  
 Point focalPoint = location.add(direction.scale(depthOfField));  
  
 List<Point> aperturePoints = BlackBoard.*generateJitteredSamples*(  
 location, vRight, vUp, apertureRadius, amountOfRays\_DOF, boardShape);  
  
 for (Point aperturePoint : aperturePoints) {  
 Vector dir = focalPoint.subtract(aperturePoint).normalize();  
 DOFRays.add(new Ray(aperturePoint, dir));  
 }  
  
 return DOFRays;  
}

### **השוואת תמונה ללא האפקט לתמונה עם האפקט**

תמונה שמכילה צילום מסך, אומנות, צבעוני, גרפיקה

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.ללא DOF:

עם DOF:



## **החלקת עקומות (Anti-Aliasing / AA)**

אפקט פשוט יחסית, מתבסס ישירות על ריבוי דגימות בלי הרבה מעבר לכך, והוא מאפשר יצירת מצב שבו בהרבה מהאזורים בתמונה שבהם צבע מתחלף בצבע אחר ההחלפה הזו תתבצע באופן הדרגתי. כך נרמה את העין לחשוב שהתמונה היא מושלמת וכל נקודה צבועה בדיוק בצבע המתאים לה (בניגוד למציאות שבה כל פיקסל צבוע בצבע אחיד ואין לנו יכולת לצבוע אזורים שונים בפיקסל בצעים שונים) מה שניתן לראות גם כהחלקת העקומות המשוננות שנוצרות בשיטה הרגילה.

### **כיצד ממומש בפועל?**

ראשית נוסיף מספר שדות הכרחיים:

1. useAA – שדה שיקבע האם יהיה שימוש ב aa בתמונה הנוכחית.
2. amountOfRays\_AA

האלגוריתם לאפקט החלקת עקומות מתבצע במספר שלבים שמבוצעים עבור כל פיקסל בנפרד:

1. נשתמש ב Jittered sampling כדי ליצור אוסף נקודות דגימה שמרכזו יהיה במרכז הפיקסל.
2. נירה קרניים ממיקום המצלמה דרך כל אחד מנקודות הדגימה וכך ניצור מעין קונוס של קרניים שיעבור דרך הפיקסל.
3. נבצע מעקב קרניים עבור כל אחת מהקרניים שיצרנו בשלב 2 כדי לקבל את הצבע שיופק ממנה.
4. נמצא את הממוצע של הצבעים שקיבלנו בשלב 3.
5. נצבע את הפיקסל בצבע שקבלנו בשלב 4.

התהליך תואם להגיון ירי של מספר קרניים במקום קרן יחידה יאפשר מעבר הדרגתי יותר בין הצבעים בתמונה.

למשל אם מקודם היו 4 פיקסלים כלשהם הימני ביותר בצבע שחור של הרקע והשלושה משמאלו בצבע אדום של ספירה עכשיו 4 הפיקסלים ייתכן שייראו ככה: הימני ביותר שחור מוחלט משמאלו פיקסל שצבעו מורכב מ 70% שחור 30% אדום משמאלו פיקסל שבצעו מורכב מ 70% אדום 30% שחור ומשמאלו פיקסל אדום מוחלט.

כך נקבל מעבר הדרגתי יותר של הצבעים שייתן יותר תחושה של תמונה שבה כל נקודה צבועה בצבע המתאים לה וכמובן העקומות בתמונה נראות יותר חלקות.

### **מימוש בקוד**

*/\*\*  
 \* Non-adaptive super sampling for Anti-Aliasing (AA) effect.  
 \* This method samples the pixel area uniformly.  
 \*  
 \* @param j the horizontal pixel index  
 \* @param i the vertical pixel index  
 \* @return the averaged color for the sampled area  
 \*/*private Color nonAdaptiveAaColor (int j, int i) {  
  
 Color color;  
  
 if (useDOF) {  
 Color totalColor = Color.*BLACK*;  
 List<Vector> AAVectors = getAAVectors(j, i);  
  
 for (Vector AAVector : AAVectors) {  
 Color dofColorForAaVector = dofColorFromDirection(AAVector);  
 totalColor = totalColor.add(dofColorForAaVector);  
 }  
 color = totalColor.reduce(AAVectors.size());  
  
 } else {  
 List<Ray> rays = constructAARays(j, i);  
 color = averageRays(rays);  
 }  
  
 return color;  
}

*/\*\*  
 \* Constructs Anti-Aliasing (AA) rays for a specific pixel.  
 \*  
 \* @param j the horizontal pixel index  
 \* @param i the vertical pixel index  
 \* @return a list of AA rays  
 \*/*private ArrayList<Ray> constructAARays(int j, int i) {  
 ArrayList<Ray> AARays = new ArrayList<>();  
 ArrayList<Vector> AAVectors = getAAVectors(j, i);  
  
 for (Vector AAVector : AAVectors) {  
 AARays.add(new Ray(location, AAVector));  
 }  
  
 return AARays;  
}

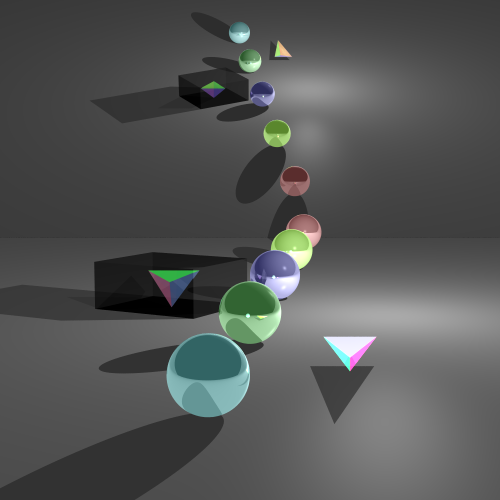
*/\*\*  
 \* Generates Anti-Aliasing (AA) vectors for a specific pixel.  
 \* These vectors are used to create rays for anti-aliasing effects.  
 \*  
 \* @param j the horizontal pixel index  
 \* @param i the vertical pixel index  
 \* @return a list of AA vectors  
 \*/*private ArrayList<Vector> getAAVectors(int j, int i) {  
 ArrayList<Vector> AAVectors = new ArrayList<>();  
  
 Point pij = calculatePixelCenter(j, i);  
  
 double pixelRadius = (viewPlaneWidth / nX) / 2.0;  
  
 List<Point> pixelPoints = BlackBoard.*generateJitteredSamples*(  
 pij, vRight, vUp, pixelRadius, amountOfRays\_AA, boardShape);  
  
 for (Point pixelPoint : pixelPoints) {  
 Vector dir = pixelPoint.subtract(location).normalize();  
 AAVectors.add(dir);  
 }  
  
 return AAVectors;  
}

### **השוואת תמונה ללא האפקט לתמונה עם האפקט**

ללא AA:

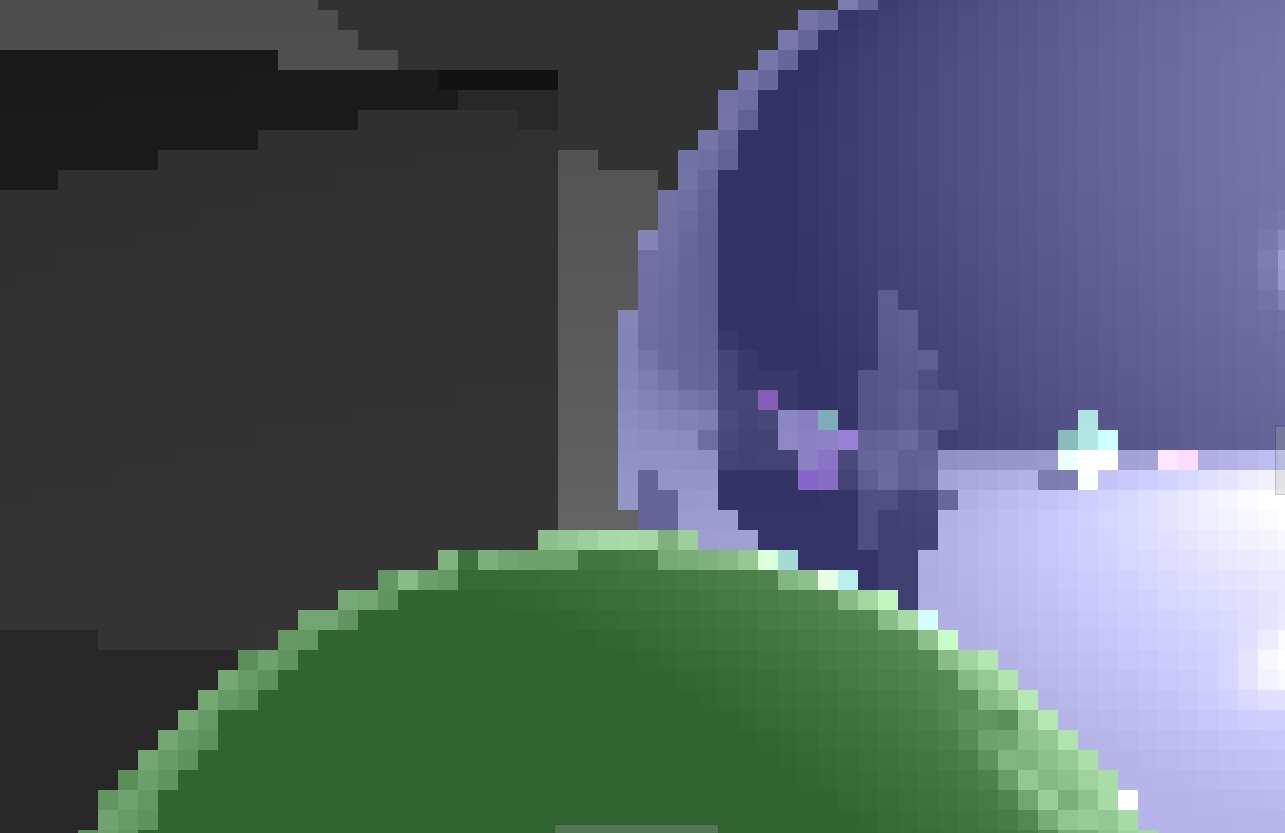
תמונה שמכילה צילום מסך, אומנות, צבעוני, גרפיקה

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

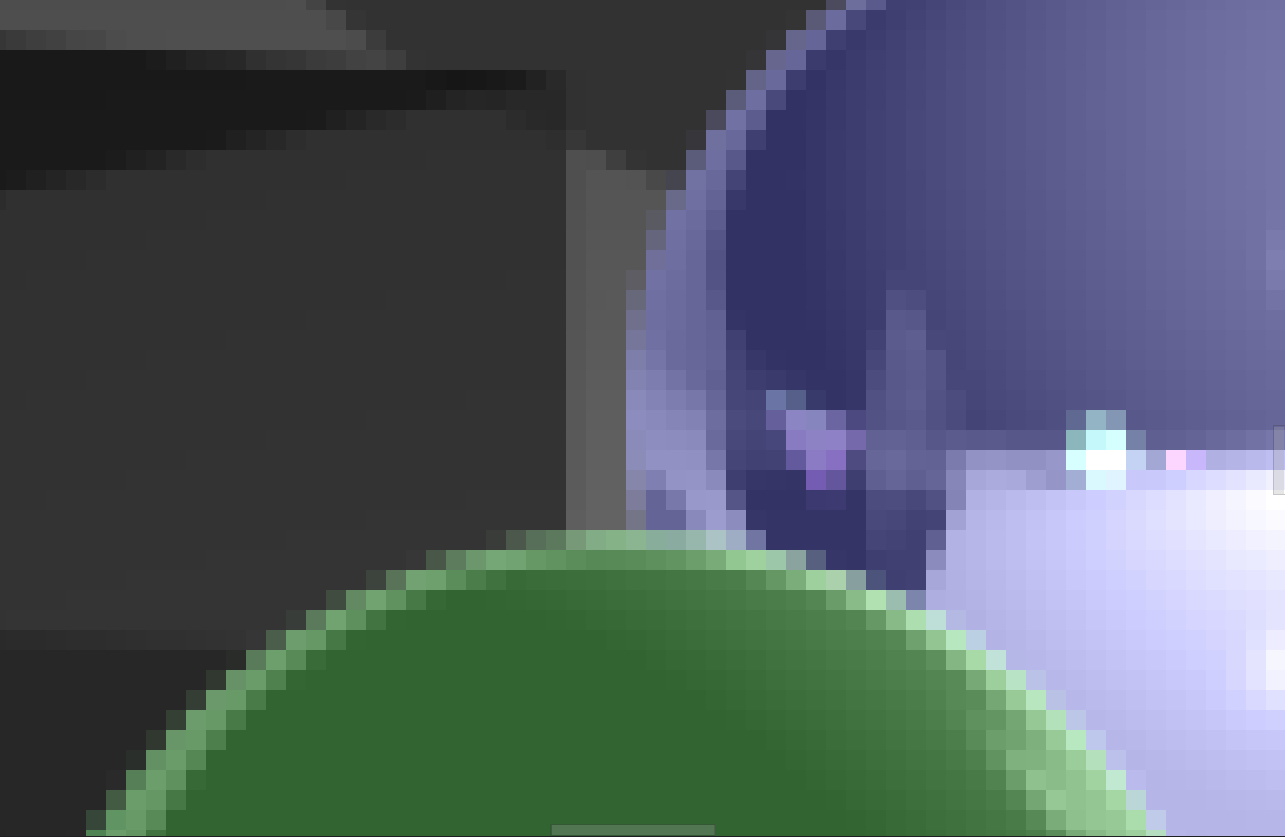
עם AA:

### **השוואת תמונה ללא האפקט לתמונה עם האפקט, ממבט קרוב**

ללא AA:



עם AA:



**שילוב AA יחד עם DOF**השילוב של AA יחד עם DOF הוא מעט מסובך, ניתן לחשוב על מספר דרכים לבצע אותו שאולי יתנו תוצאות דומות יחסית, אנחנו החלטנו לבצע אותו בדרך שנציג להלן כי היא נראתה לנו כמדויקת ביותר מבחינה פיזיקלית (וכך גם המליץ לנו ה gpt).

באופן כללי המימוש הוא ע"י ביצוע האלגוריתם לחישוב הצבע של אפקט ה DOF עבור כל וקטור כיוון של קרן AA, ואז קבלת הצבע הסופי עבור הפיקסל ע"י ממוצע של הצבעים הנ"ל.

### **כיצד ממומש בפועל?**

להלן נציג את שלבי המימוש בצורה יותר מפורטת (השלבים מבוצעים עבור כל פיקסל בנפרד):

1. יצרנו מתודה getAAVectors שתחזיר לנו את הכיוונים בהם היו אמורות להשלח קרני ה AA.

עבור כל אחד מוקטורי הכיוון שקבלנו בצענו את השלבים הבאים:

1. מצאנו את נקודת הפוקוס ע"י התקדמות של depthOfField יחידות על גבי וקטור הכיוון מהנקודה בה ממוקמת המצלמה.
2. מצאנו את הצבע שמתקבל מאפקט ה DOF ע"פ שלבים 2-5 בהסבר לגבי מימוש DOF.
3. מצאנו את הצבע שמתקבל מממוצע כל הצבעים שהתקבלו בשלב 1.ב עבור וקטורי הכיוון של קרני AA (בעצם זהו ממוצע של ממוצעים כי כל צבע שהתקבל בשלב 1.ב התקבל בעצמו מממוצע של צבעים).
4. צבענו את הפיקסל בצבע הממוצע שהתקבל בשלב 2.

### **מימוש בקוד**

החלק הרלוונטי מודגש בצהוב.

*/\*\*  
 \* Non-adaptive super sampling for Anti-Aliasing (AA) effect.  
 \* This method samples the pixel area uniformly.  
 \*  
 \* @param j the horizontal pixel index  
 \* @param i the vertical pixel index  
 \* @return the averaged color for the sampled area  
 \*/*private Color nonAdaptiveAaColor (int j, int i) {  
  
 Color color;  
  
 if (useDOF) {  
 Color totalColor = Color.*BLACK*;  
 List<Vector> AAVectors = getAAVectors(j, i);  
  
 for (Vector AAVector : AAVectors) {  
 Color dofColorForAaVector = dofColorFromDirection(AAVector);  
 totalColor = totalColor.add(dofColorForAaVector);  
 }  
 color = totalColor.reduce(AAVectors.size());  
  
 } else {  
 List<Ray> rays = constructAARays(j, i);  
 color = averageRays(rays);  
 }  
  
 return color;  
}

### **הדגמה של השילוב בין האפקטים AA ו DOFתמונה שמכילה צילום מסך, צבעוני, אומנות, עיצוב תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.**

# **מיני פרויקט 2**

## **ריבוי דגימות מסתגל (Adaptive Super Sampling)**

ביצענו את השיפור הזה גם עבור DOF וגם עבור AA אבל העקרון מאוד דומה לכן נסביר באופן כללי את השיפור ולא בנפרד עבור AA ועבור DOF.

כל מטרתנו כאן היא לבצע האצה של האלגוריתמים שיצרנו עבור מיני פרויקט 1, כלומר ברצוננו להגיע לתמונה שתהיה 'מספיק' קרובה לתמונה שקבלנו במיני פרויקט 1 תוך כדי חסכון גדול בזמן. מזה 'מספיק' קרובה? נראה בהמשך וגם נוכל לשלוט עד כמה היא תהיה קרובה כמובן.

בעצם המוטיבציה מאחורי ההאצה הזו היא שבמהלך השיפורים במיני פרויקט 1 בצענו ריבוי דגימות כדי לשפר את איכות התמונה או כדי לאפשר אפקטים חדשים, מה הבעיה? הבעיה היא שעבור כל אזור דגימה יצרנו כמות דגימות קבועה כשפועל כמות הדגימות שלה אנו זקוקים בפועל כדי לשפר את איכות התמונה משתנה בין אזור דגימה למשנהו.

דוגמא פשוטה וקיצונית:

נניח שבתמונה שלנו יש פיקסל שהאזור בסצינה שאותו הוא מכסה מכיל חלקים של 20 אובייקטים שונים. נקרא לו פיקסל 1.

בנוסף נניח שישנו פיקסל אחר בתמונה שבאזור בסצינה שאותו הוא מכסה אין אובייקטים כלל אלא רק רקע. נקרא לו פיקסל 2.

היינו רוצים לירות כמות גדולה של קרני AA כדי לקבל תמונה מציאותית עבור פיקסל 1 למשל 100 קרניים, אבל כרגע במצב לפני ההאצה המשמעות היא שנאלץ לירות גם 100 קרני AA עבור פיקסל 2 שזה בזבוז זמן מוחלט כיוון שכל הקרניים האלו פשוט יחזירו צבע זהה (צבע הרקע) ונקבל אותה תוצאה שהיינו מקבלים עבור ירי של קרן יחידה.

אנחנו רוצים "להחכים" את האלגוריתם שלנו כך שידע לזהות מתי הוא צריך לירות יותר קרניים ומתי פחות וכך לחסוך זמן.

### **כיצד ממומש בפועל?**

ראשית נוסיף כמה שדות הכרחיים:

עבור DOF:

1. useAdaptiveSuperSamplingForDOF – שדה שייקבע האם יהיה שימוש בריבוי דגימות מסתגל עבור DOF.
2. numOfSubAreaSamplesAdaptiveDOF – שדה שייקבע את מספר הדגימות שיילקחו בכל תת אזור דגימה במהלך האלגוריתם (עבור השימוש לטובת DOF)
3. maxSamplesAdaptiveDOF – שדה שיציין את מספר הדגימות המקסימלי שיכול להילקח עבור אזור הדגימה כולו במהלך האלגוריתם (עבור השימוש לטובת DOF), דאגנו שלשדה הזה יהיה אפשר להזין רק ערכים מהצורה numOfSubAreaSamplesAdaptiveDOF \* 4^k כאשר k הוא מספר שלם שגדול או שווה ל 0, מכיוון שאלו הדילוגים שמבצע האלגוריתם כשהוא מחלק את אזור הדגימה (יוסבר בהמשך).
4. colorThresholdAdaptiveDOF – שדה שמציין את המרחק המקסימלי בין צבעים על מנת שהם ייחשבו 'דומים' מבחינת האלגוריתם (עבור השימוש לטובת DOF).

עבור AA:

1. useAdaptiveSuperSamplingForAA - שדה שייקבע האם יהיה שימוש בריבוי דגימות מסתגל עבור AA.
2. numOfSubAreaSamplesAdaptiveAA - שדה שייקבע את מספר הדגימות שיילקחו בכל תת אזור דגימה במהלך האלגוריתם (עבור השימוש לטובת AA).
3. maxSamplesAdaptiveAA - שדה שיציין את מספר הדגימות המקסימלי שיכול להילקח עבור אזור הדגימה כולו במהלך האלגוריתם (עבור השימוש לטובת AA), דאגנו שלשדה הזה יהיה אפשר להזין רק ערכים מהצורה numOfSubAreaSamplesAdaptiveAA \* 4^k כאשר k הוא מספר שלם שגדול או שווה ל 0, מכיוון שאלו הדילוגים שמבצע האלגוריתם כשהוא מחלק את אזור הדגימה (יוסבר בהמשך).
4. colorThresholdAdaptiveAA - שדה שמציין את המרחק המקסימלי בין צבעים על מנת שהם ייחשבו 'דומים' מבחינת האלגוריתם (עבור השימוש לטובת AA).

האלגוריתם יבצע את הדגימה בצורה רקורסיבית.

תנאי העצירה של הרקורסיה הוא התקיימות של לפחות אחד מהתנאים הבאים:

* מספר הדגימות בעומק הנוכחי של הרקורסיה הוא גדול או שווה למספר הדגימות המקסימלי שהוחלט ע"י המשתמש (maxSamplesAdaptiveAA או maxSamplesAdaptiveDOF תלוי לטובת איזה אפקט משתמשים בריבוי הדגימות המסתגל).
* הצבעים שהתקבלו מהדגימות היו דומים זה לזה כלומר המרחק בין הצבע של כל דגימה לבין הצבע של שאר הדגימות האחרות לא עולה על הסף לדמיון בין צבעים שהוחלט ע"י המשתמש (colorThresholdAdaptiveAA או colorThresholdAdaptiveDOF תלוי לטובת איזה אפקט משתמשים בריבוי הדגימות המסתגל).

ברקורסיה נתחיל מדגימה של numOfSubAreaSamplesAdaptiveAA דגימות באזור הדגימה המקורי, כעת נבדוק האם תנאי העצירה הנ"ל מתקיים.

אם תנאי העצירה מתקיים נחזיר את ממוצע הדגימות שנאספו בקריאה הנוכחית של הפונקציה הרקורסיבית.

אם תנאי העצירה לא מתקיים נחלק את אזור הדגימה של הקריאה הנוכחית לפונקציה הרקורסיבית לארבע תתי אזורים בצורת ריבועים זהים, ונפעיל את הפונקציה הרקורסיבית עבור כל אחד מתתי האזורים האלו.

מכל אחת מארבעת הקריאות הרקורסיביות של הפונקציה לעצמה (עבור כל אחד מהרבעים של תת אזור הדגימה שעבורו היא נקראה) היא מקבלת צבע, את הצבעים היא מוסיפה לרשימה. לאחר סיום הקריאות יש לפונקציה רשימה עם ארבעה צבעים והיא מחזירה את הממוצע של ארבעת הצבעים האלה.

**הערה:** דאגנו שהשילוב של AA יחד עם DOF יפעל בצורה תקינה ע"פ המודל שהצגנו לעיל (במיני פרויקט 1) גם כאשר לא נעשה שימוש בריבוי דגימות מסתגל עבור אף אחד מהאפקטים, גם כאשר נעשה שימוש בריבוי דגימות מסתגל רק עבור אחד מהאפקטים וגם כאשר נעשה שימוש בריבוי דגימות מסתגל עבור שני האפקטים.

### **מימוש בקוד**

#### **מימוש ריבוי דגימות מסתגל עבור אפקט החלקת עקומות**

הפונקציה שמזמנת את הפונקציה הרקורסיבית (הזימון מודגש בצהוב):

**הערה:** זוהי הפונקציה הכללית לחישוב צבע הפיקסל כאשר יש שימוש באפקט החלקת עקומות, גם עבור ריבוי דגימות מסתגל וגם עבור ריבוי דגימות שאינו מסתגל.

*/\*\*  
 \* Adaptive super sampling for Anti-Aliasing (AA) effect.  
 \* This method samples the pixel area adaptively based on color variance.  
 \*  
 \* @param j the horizontal pixel index  
 \* @param i the vertical pixel index  
 \* @return the averaged color for the sampled area  
 \*/*private Color aaColor(int j, int i) {  
 Color color;  
  
 if (useAdaptiveSuperSamplingForAA) {  
 Point pc = calculatePixelCenter(j, i);  
 double pixelRadius = (viewPlaneWidth / nX) / 2.0;  
 color = adaptiveAaColorForSubArea(pc, pixelRadius, 0);  
  
 } else {  
 color = nonAdaptiveAaColor(j, i);  
 }  
  
 return color;  
}

הפונקציה הרקורסיבית:

**הערה:** שימו לב שהקוד תומך בצורה תקינה בשילוב AA יחד עם DOF, הדגשנו בצהוב את החלק שבו זה מתרחש.

*/\*\*  
 \* Helper method for adaptive sampling on arbitrary point and radius.  
 \*  
 \* @param center center point of the sub-pixel area  
 \* @param radius if sampling area (boardShape) is circular, this is the radius of that circle, if square, this is half the side length  
 \* @param depth current recursion depth  
 \* @return averaged color for the sampled area  
 \*/*private Color adaptiveAaColorForSubArea(Point center, double radius, int depth) {  
 List<Point> samplePoints = BlackBoard.*generateJitteredSamples*(  
 center, vRight, vUp, radius, numOfSubAreaSamplesAdaptiveAA, boardShape);  
  
 List<Color> sampleColors = new ArrayList<>();  
  
 for (Point samplePoint : samplePoints) {  
 Vector dir = samplePoint.subtract(location).normalize();  
 Ray ray = new Ray(location, dir);  
 Color sampleColor;  
  
 if (useDOF) {  
 sampleColor = dofColorFromDirection(dir);  
 }  
 else {  
 sampleColor = rayTracer.traceRay(ray);  
 }  
  
 sampleColors.add(sampleColor);  
 }  
  
 // If the number of samples exceeds the maximum allowed or colors are similar, return the average color.  
  
 // Math.pow(4, depth) represents the number of subareas the pixel area is divided into at the current recursion depth.  
 // Multiplying it by numOfSubAreaSamplesAdaptiveAA gives the total number of rays (samples) traced at this depth.  
  
 // for the maximum check we could use == because the number of samples is always numOfSubAreaSamplesAdaptiveAA \* k^4,  
 // but we use >= just to be safe  
 if (numOfSubAreaSamplesAdaptiveAA \* Math.*pow*(4, depth) >= maxSamplesAdaptiveAA || colorsAreSimilar(sampleColors, colorThresholdAdaptiveAA)) {  
 return averageColors(sampleColors);  
 }  
  
 double halfRadius = radius / 2.0;  
 Color totalColor = Color.*BLACK*;  
 double[] offsetsX = {-halfRadius, halfRadius};  
 double[] offsetsY = {-halfRadius, halfRadius};  
  
 for (double offsetX : offsetsX) {  
 for (double offsetY : offsetsY) {  
 Point subCenter = center.add(vRight.scale(offsetX)).add(vUp.scale(offsetY));  
 totalColor = totalColor.add(adaptiveAaColorForSubArea(subCenter, halfRadius, depth + 1));  
 }  
 }  
  
 // We reduce by 4 to get the average, because total color is the sum of the colors of 4 sub-areas.  
 return totalColor.reduce(4);  
}

#### **מימוש ריבוי דגימות מסתגל עבור אפקט עומק השדה**

הפונקציה שמזמנת את הפונקציה הרקורסיבית (הזימון מודגש בצהוב):

**הערה:** זוהי הפונקציה הכללית לחישוב צבע (של פיקסל במקרה של DOF בלבד או של איבר בממוצע הצבעים כאשר יש שימוש ב DOF יחד עם AA) כאשר יש שימוש באפקט עומק השדה, גם עבור ריבוי דגימות מסתגל וגם עבור ריבוי דגימות שאינו מסתגל.

*/\*\*  
 \* Calculate the averaged color for the sampled area using adaptive super sampling for Depth of Field (DOF).  
 \* Based on a given direction vector.  
 \*  
 \* @param direction the direction vector for the DOF rays  
 \* @return the averaged color for the sampled area using adaptive super sampling for Depth of Field (DOF).  
 \*/*private Color dofColorFromDirection(Vector direction) {  
  
 Color color;  
  
 if (useAdaptiveSuperSamplingForDOF) {  
 // The focal point is where all rays should converge  
 Point focalPoint = location.add(direction.scale(depthOfField));  
  
 color = adaptiveDofColorForSubArea(location, apertureRadius, focalPoint, 0);  
 } else {  
 color = nonAdaptiveDofColor(direction);  
 }  
  
 return color;  
}

הפונקציה הרקורסיבית:

*/\*\*  
 \* Helper method for adaptive sampling on arbitrary point and radius.  
 \* This method samples the pixel area adaptively based on color variance.  
 \*  
 \* @param center center point of the sub-area  
 \* @param radius if sampling area (boardShape) is circular, this is the radius of that circle, if square, this is half the side length  
 \* @param focalPoint the focal point for DOF  
 \* @param depth current recursion depth  
 \* @return averaged color for the sampled area  
 \*/*private Color adaptiveDofColorForSubArea(Point center, double radius, Point focalPoint, int depth) {  
 List<Point> apertureSamples = BlackBoard.*generateJitteredSamples*(  
 center, vRight, vUp, radius, numOfSubAreaSamplesAdaptiveDOF, boardShape);  
  
 List<Color> sampleColors = new ArrayList<>();  
  
 for (Point apertureSample : apertureSamples) {  
 Vector dir = focalPoint.subtract(apertureSample).normalize();  
 Ray ray = new Ray(apertureSample, dir);  
 Color color = rayTracer.traceRay(ray);  
 sampleColors.add(color);  
 }  
  
 // If the number of samples exceeds the maximum allowed or colors are similar, return the average color.  
  
 // Math.pow(4, depth) represents the number of subareas the area is divided into at the current recursion depth.  
 // Multiplying it by numOfSubAreaSamplesAdaptiveDOF gives the total number of rays (samples) traced at this depth.  
  
 // for the maximum check we could use == because the number of samples is always numOfSubAreaSamplesAdaptiveDOF \* k^4,  
 // but we use >= just to be safe  
 if (numOfSubAreaSamplesAdaptiveDOF \* Math.*pow*(4, depth) >= maxSamplesAdaptiveDOF || colorsAreSimilar(sampleColors, colorThresholdAdaptiveDOF)) {  
 return averageColors(sampleColors);  
 }  
  
 double halfRadius = radius / 2.0;  
 Color totalColor = Color.*BLACK*;  
 double[] offsetsX = {-halfRadius, halfRadius};  
 double[] offsetsY = {-halfRadius, halfRadius};  
  
 for (double offsetX : offsetsX) {  
 for (double offsetY : offsetsY) {  
 Point subCenter = center.add(vRight.scale(offsetX)).add(vUp.scale(offsetY));  
 totalColor = totalColor.add(adaptiveDofColorForSubArea(subCenter, halfRadius, focalPoint, depth + 1));  
 }  
 }  
  
 // We reduce by 4 to get the average, because total color is the sum of the colors of 4 sub-areas.  
 return totalColor.reduce(4);  
}

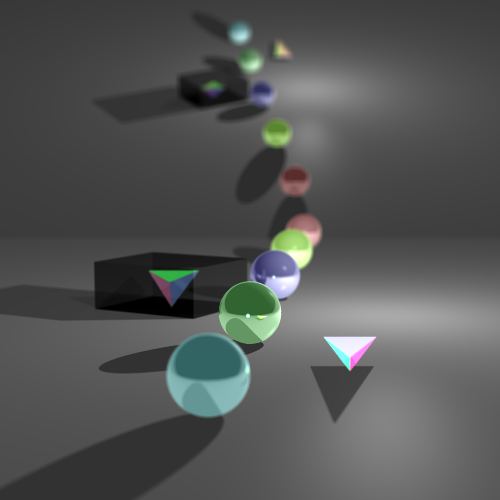
### **השוואת תמונה ללא ההאצה לתמונה עם ההאצה**

DOF עם ריבוי דגימות לא מסתגל ו AA עם ריבוי דגימות לא מסתגל:

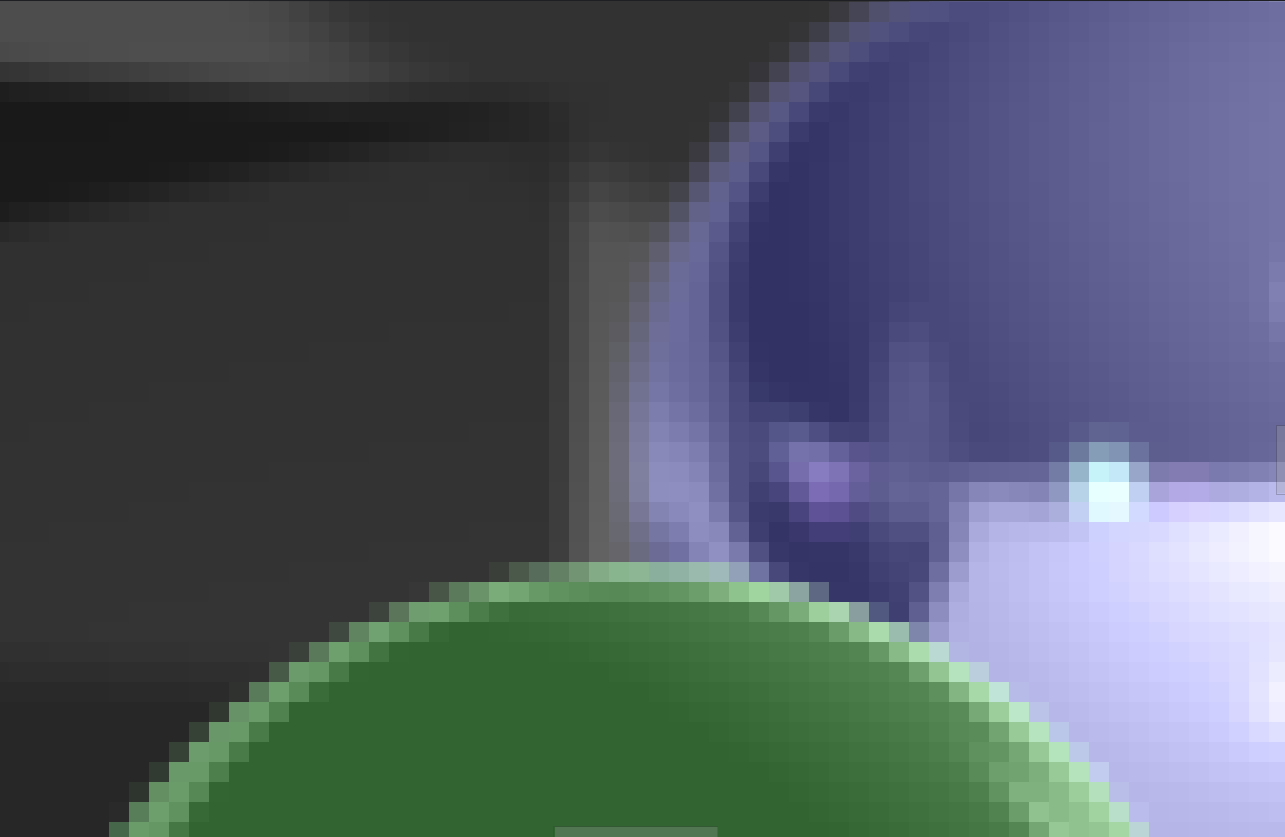
תמונה שמכילה צילום מסך, צבעוני, אומנות, עיצוב

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

DOF עם ריבוי דגימות מסתגל ו AA עם ריבוי דגימות מסתגל:



### **השוואת תמונה ללא ההאצה לתמונה עם ההאצה, ממבט קרוב**

DOF עם ריבוי דגימות לא מסתגל ו AA עם ריבוי דגימות לא מסתגל:

DOF עם ריבוי דגימות מסתגל ו AA עם ריבוי דגימות מסתגל:



ניתן לראות שהתמונה עם ריבוי דגימות מסתגל זהה כמעט לחלוטין לתמונה עם ריבוי דגימות לא מסתגל, שזה בדיוק מה שרצינו להשיג בשימוש בריבוי דגימות מסתגל – להגיע לתמונה זהה בזמן קצר הרבה יותר (את השיפור בזמן נראה בהמשך).

### **הדגמת השיפור בזמני הריצה בעקבות ההאצה**

להלן זמני הריצה של שבעה בדיקות שונות בהן מתבצע רינדור תמונה אשר מציגה את אותה הסצנה (הסצנה של התמונה הסופית שלנו) עם הוגנות מלאה מבחינת ערכי המצלמה.

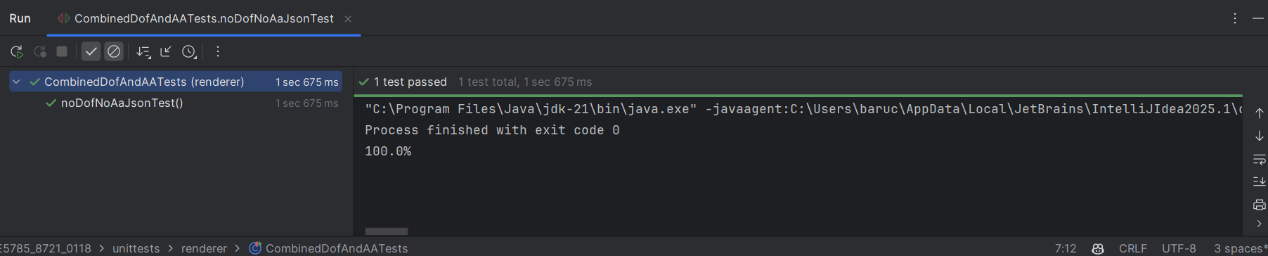
ההוגנות בערכי המצלמה מתבטאת בכך ש:

* כל ערכי המצלמה שמשותפים לכמה מהבדיקות הם זהים בכל אותן הבדיקות (לדוגמא: הרזולוציה זהה בכל הבדיקות ((500x500, דוגמא נוספת: רדיוס הצמצם בבדיקת DOF עם ריבוי דגימות לא מסתגל זהה לרדיוס הצמצם בבדיקת DOF עם ריבוי דגימות מסתגל (הרדיוס הוא 1) ).
* כאשר יש שימוש ב AA עם ריבוי דגימות לא מסתגל ישנן 64 קרני AA.
* כאשר יש שימוש ב DOF עם ריבוי דגימות לא מסתגל ישנן 64 קרני DOF .
* כאשר יש שימוש ב AA עם ריבוי דגימות מסתגל מספר הדגימות המקסימלי הוא 64.
* כאשר יש שימוש ב DOF עם ריבוי דגימות מסתגל מספר הדגימות המקסימלי הוא 64.

**ואלו הזמנים, בצירוף תמונות מתוך ה intellij שמציגות אותם:**

**הערה:** בצילומי המסך של זמני הריצה השם של התיקייה baruc"" שמופיע בנתיב של קובץ ההרצה זה השם של אבא של אחד מהמגישים.

ללא AA + ללא DOF:

1.675 שניות

AA עם ריבוי דגימות לא מסתגל + ללא DOF:

26.470 שניות

תמונה שמכילה טקסט, תכונות מולטימדיה, תוכנה, תוכנה גרפית

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

ללא AA + DOF עם ריבוי דגימות לא מסתגל:

25.613 שניות

תמונה שמכילה טקסט, תכונות מולטימדיה, תוכנה, תוכנה גרפית

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

AA עם ריבוי דגימות לא מסתגל + DOF עם ריבוי דגימות לא מסתגל:

27 דקות 2 שניות

תמונה שמכילה טקסט, צילום מסך, תוכנה, תכונות מולטימדיה

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

AA עם ריבוי דגימות מסתגל + DOF עם ריבוי דגימות לא מסתגל:

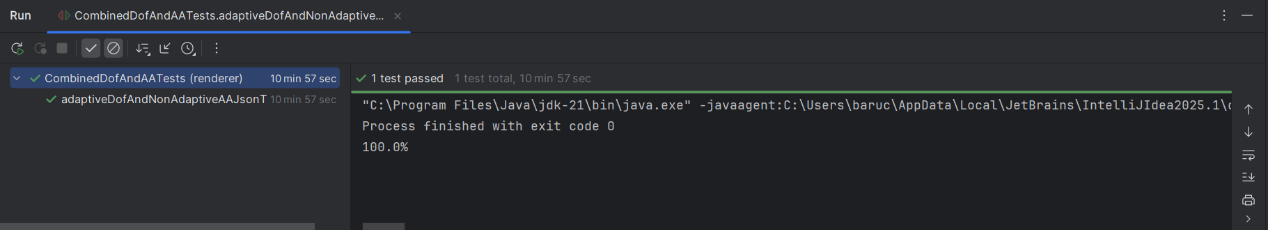
6 דקות 33 שניות

תמונה שמכילה טקסט, תכונות מולטימדיה, תוכנה, תוכנה גרפית

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

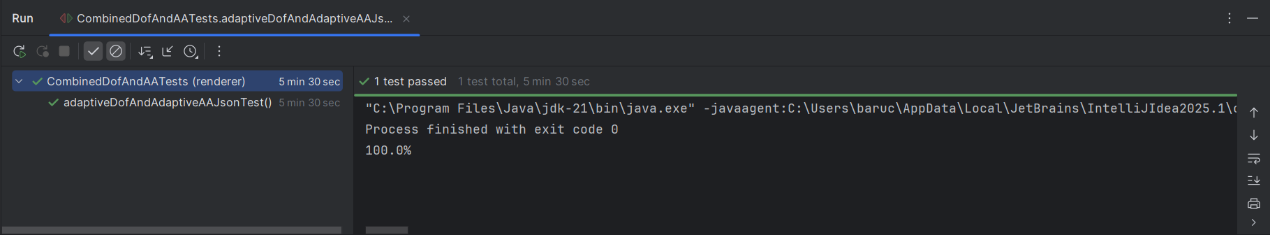
AA עם ריבוי דגימות לא מסתגל + DOF עם ריבוי דגימות מסתגל:

10 דקות 57 שניות



AA עם ריבוי דגימות מסתגל + DOF עם ריבוי דגימות מסתגל:

5 דקות 30 שניות



הדגש העיקרי הוא על כך שזמן הריצה של הבדיקה

AA עם ריבוי דגימות מסתגל + DOF עם ריבוי דגימות מסתגל

לקח בערך פי 5 פחות זמן מזמן הריצה של הבדיקה

AA עם ריבוי דגימות לא מסתגל + DOF עם ריבוי דגימות לא מסתגל

# **רשימת בונוסים שעשינו**

1. שלב 2 - נורמל לגליל סופי
2. שלב 3 - חיתוך עם מצולע
3. שלב 3 - חיתוך עם משולש בשיטה שמבוססת על קואורדינטות בריצנטריות
4. שלב 5 או יותר מאוחר - שימוש בקבצי json להגדרת סצנה
5. שלב 6 - ספוט ממוקד ("פנס")
6. שלב 6 - סיום מלא תוך שבוע
7. שלב 7 - תמונה עם 10+ עצמים
8. שלב 7 - פתרון בעיית מרחק בהצללה בדרך 2
9. מיני פרויקט 1 - jittered sampling
10. מיני פרויקט 1 - ביצוע שיפור החלקת עקומות בנוסף לשיפור שהוקצה לנו (עומק השדה)
11. מיני פרויקט 2 – ביצוע האצה של ריבוי דגימות מסתגל עבור עומק השדה בנוסף להאצה שהוקצתה לנו (ריבוי דגימות מסתגל עבור החלקת עקומות)