# הנחיות לפרויקט ה- Rasterization pipeline

# חלק 1 – הכרות עם הפרויקט ורסטריזציה של קוים

## מבנה הפרויקט Rasterization

כמו הפרויקט RayTracing גם הפרויקט Rasterization מחולק לשתי חבילות. **בחבילה app\_interface** תמצאו את קבצי ממשק המשתמש של הפרויקט ומחלקות נוספות שכבר כתובות עבורכם ונועדו על מנת שתוכלו להתמקד בכתיבת הקוד האלגוריתמי שנלמד בקורס. פיתחו את הקובץ **main** שנמצא בחבילה זו **על מנת להריץ** את הפרויקט.

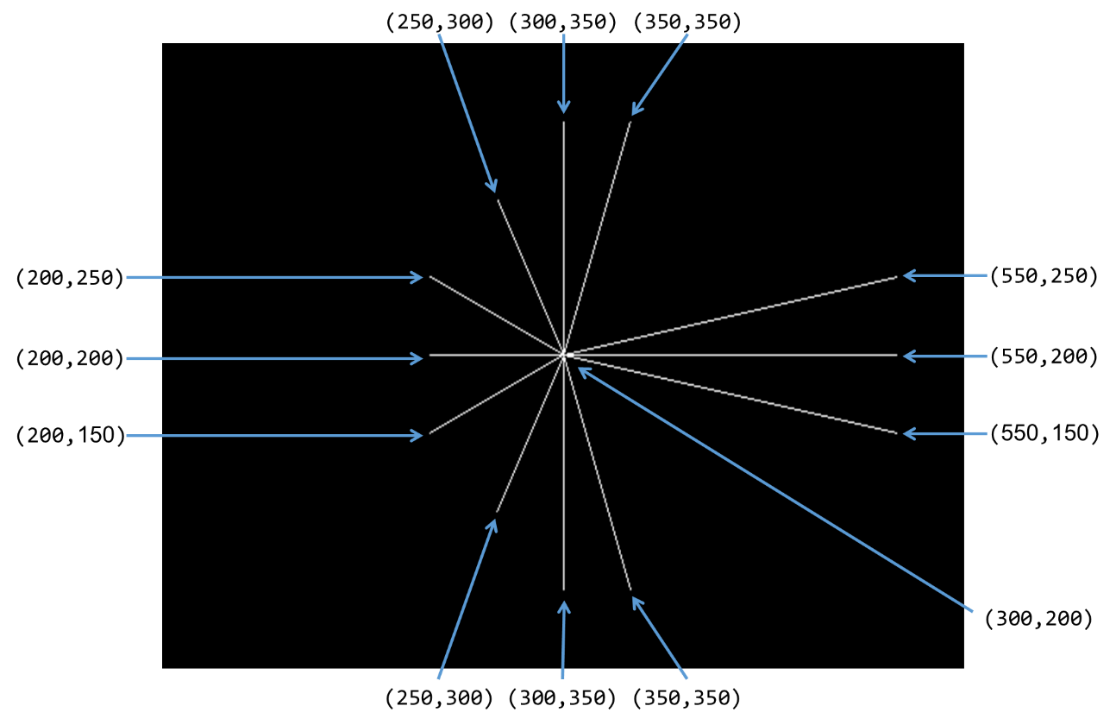
**בחבילה your\_code** נמצאות המחלקות שאותן תעדכנו. במהלך הפרויקט תעדכנו בעיקר את הקבצים ObjectModel.java ו-WorldModel.java. המחלקה WorldModel אחראית על טעינה ו-rendering של על העולם שממול למצלמה. היא עושה זאת על ידי שימוש במחלקה ObjectModel שאחראית על טעינת אובייקט אחד מהעולם ועל rendering שלו. בתרגיל זה נתמקד במחלקה ObjectModel. המתודה load של המחלקה ObjectModel אחראית לטעינת האובייקט. היא טוענת את הנתונים על כל הקודקודים באובייקט לרשימה verticesData, ואת כל הפאות במודל (איזה קודקודים הם חלק מכל פאה) לרשימה faces.

התבוננו במתודה render של המחלקה ObjectModel. מתודה זו אחראית על rendering של המודל, כלומר ציור שלו על גבי התמונה. לשם כך היא קוראת למתודה vertexProcessing עבור כל קודקוד של האובייקט (ברשימה verticesData), ואחר כך קוראת למתודה rasterization לכל פאה של האובייקט (ברשימה faces). המתודה rasterization צריכה לקרוא למתודה fragmentProcessing על כל פיקסל בפאה.  
המתודות vertexProcessing, rasterization, ו-fragmentProcessing מממשות את השלבים המתאימים ב-Rasterization graphics pipeline. המתודה vertexProcessing מקבלת אובייקט של המחלקה VertexData שמכילה נתונים על הקודקוד שנטענו מקובץ האובייקט ונשמרו ברשימה verticesData ומחשבת עבור כל קודקוד את המיקום שלו על גבי החלון, וחישובים נוספים שמעודכנים באובייקט של המחלקה VetexData. חישובים אלו מסייעים לחשב את הצבע של כל פיקסל. המתודה rasterization נקראת על כל פאה של האובייקט שברשימה faces. למתודה rasterization מועברים נתוני הקודקודים של שלושת הפאות שמרכיבות אותו (VertexData). המתודה rasterization צריכה לקרוא עבור כל הפיקסלים שבפאה למתודה fragmentProcessing ולעביר לה נתונים דרושים לכל פיקסל בעזרת אובייקט של המחלקה FragmentData.

תוכנת הדוגמה

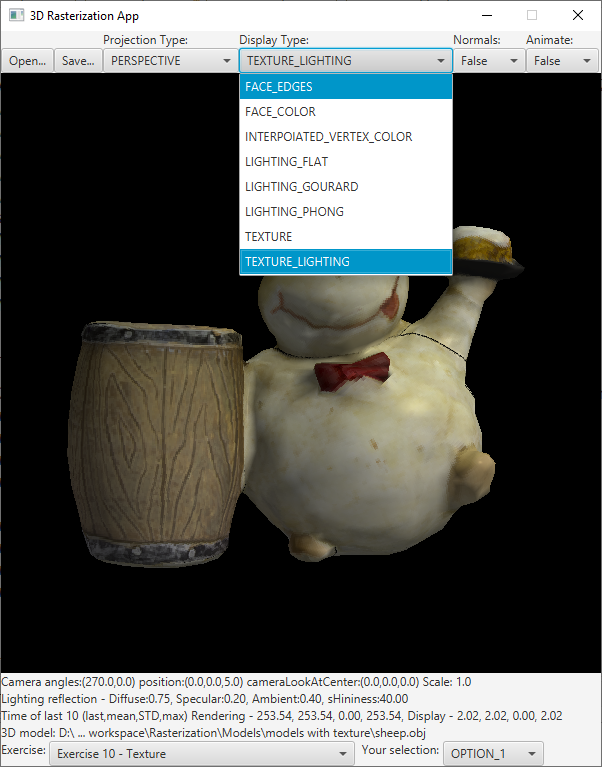
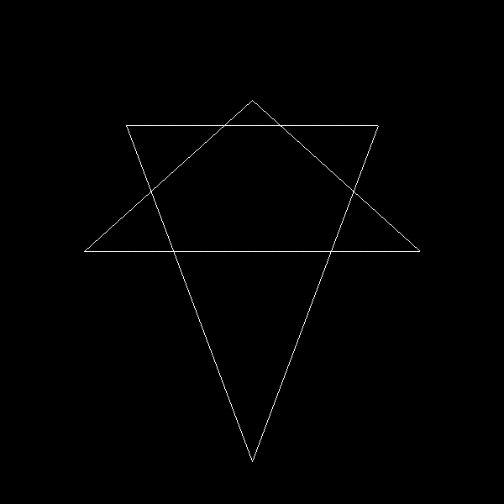
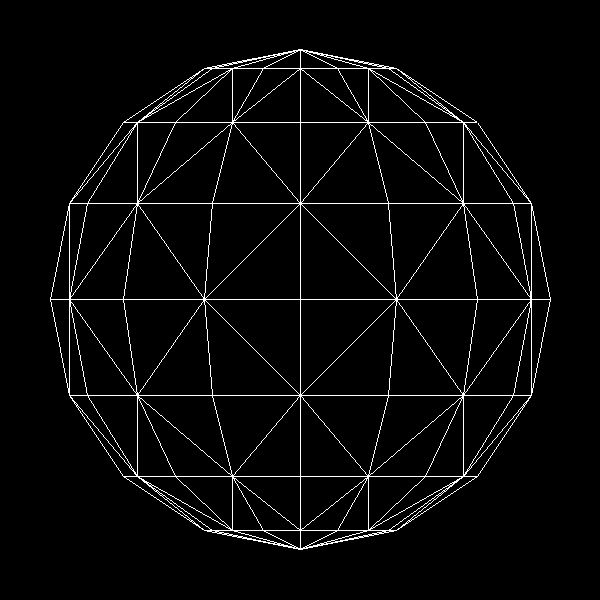
כמו בפרויקט ה-RayTracing גם בתיקיה הראשית של הפרויקט Rasterization תמצאו את הקובץ executable.jar שהוא דוגמה של הפרויקט הסופי שעליו אנחנו עובדים. נסו להריץ את הקובץ ע"י לחיצה כפולה עליו. אם זה לא עובד נסו פתיחה עם Open with ונסו את ה-JVM שמותקנים במחשבכם וקיבעו את זה שעובד כברירת המחדל. אם גם זה לא עובד, תוכלו להפעיל את הקובץ run\_executable.bat על ידי לחיצה כפולה עליו.   
שימו לב שבתחתית תוכנת הדוגמה אתם צריכים לבחור את הפרויקט שעליו אתם עובדים. התוכנה מתפקדת אחרת ונוספות לה יכולות ככל שמתקדמים בפרויקטים.

תרגיל 1 – רסטריזציה של קוים (38)  
בתרגיל זה נשלים רסטריזציה של קוים.

**א. השלימו את המתודה drawLineDDA במחלקה ObjectModel (35)**  
המתודה מקבלת את נקודת הההתחלה והסיום של הקו כנקודות p1,p2 ואת צבע הקו כ-r,g,b. התעלמו מרכיב ה-z של הנקודות.  
המתודה מקבלת גם אובייקט של המחלקה IntBufferWrapper שהוא הבאפר של התמונה. כתיבה לפיקסל במיקום x,y בבאפר התמונה (0,0 – פינה שמאלית תחתונה), של הצבע r,g,b, מתבצעת כך:  
intBufferWrapper.setPixel(x, y, r, g, b).   
כדי לפשט חברו בקו את הקאורדינטות המעוגלות של הקודקודים.  
שימו לב שחישובים ב-int מחזירים תוצאה ב-int ולכן אינם יכולים למשל להתאים לחישוב של שיפוע שיכול לצאת קטן מאחד.   
בידקו את המתודה שכתבתם על ידי שימוש במתודה testDrawLineDDA שבמחלקה UnitTest. המתודה מציירת 12 קוים ושומרת כתמונה. התמונה נשמרת כקובץ בשם "line drawing test.bmp" בתיקיה הראשית של הפרויקט, ואמורה להראות כמו בתמונה הבאה. אם אתם נתקלים בבעיות, שימו בהערות את כל הקוים והשאירו קו אחד בעייתי ודבגו אותו.  


**ב. קריאה למתודה לציור קוים (3)**  
במתודה rasterization מצוירים בתחילת המתודה רק הקודקודים של הפאה. מיחקו קוד זה ותחת התנאי:

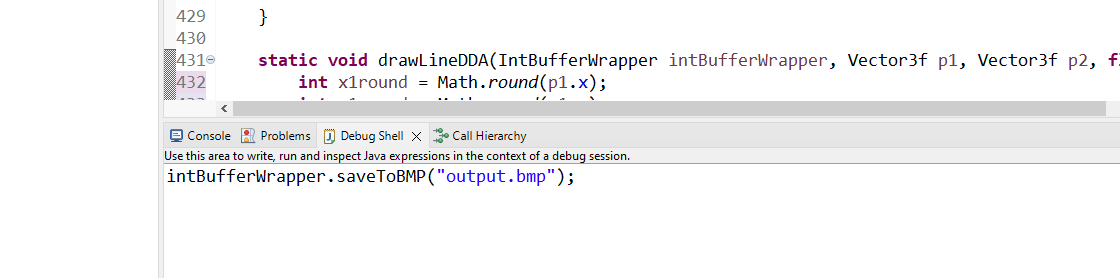
**if** (worldModel.displayType == DisplayTypeEnum.***FACE\_EDGES***) {

הוסיפו ציור של הצלעות שמחברות את הקודקודים בפאה בצבע לבן, בעזרת קריאה למתודה drawLineDDA.  
כדי לבדוק את עבודתכם שנו בתוכנה את ה-Display Type ל-FACE\_EDGES כמו בתמונה הבאה.   
טענו את אחד המודלים: ex\_01\_\_\_FirstExample\_0to600.obj, ex\_01\_\_\_sphere\_0to600.obj, שבתיקייה: Models/models\_0to600. טעינת הקובץ ע"י לחיצה על הכפתור Open… בתוכנה.   
כיוון שעדיין לא טיפלנו בטרנספורמציה הדרושה מנקודות בתלת מימד למיקום על גבי החלון, הקבצים האלו מותאמים לכך במיוחד וקבצים אחרים לא יתאימו בינתיים.  
התוצאה צריכה להראות כך:  
 

טיפים לדיבוג

* באפר התמונה שלנו מנוהל על ידי המחלקה intBufferWrapper. למחלקה זו יש את המתודה saveToBMP ששומרת את באפר התמונה כתמונה. לדוגמה שורת הקוד הבאה שומרת את הבאפר השמור באוביקט intBufferWrapper כתמונה בשם output.bmp בתיקיה הראשית של הפרויקט:

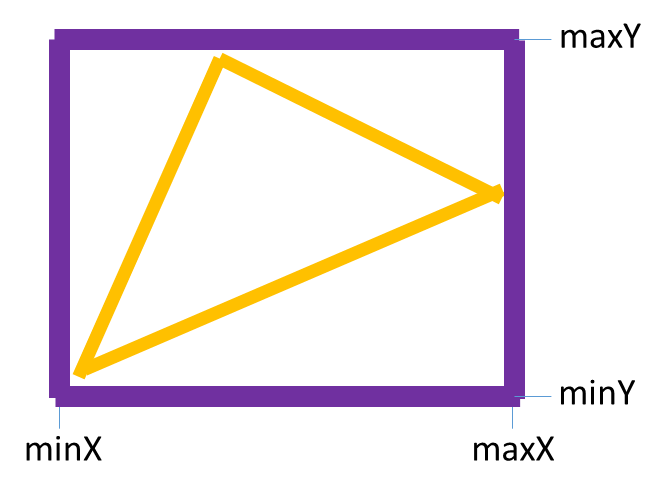
intBufferWrapper.saveToBMP("output.bmp");

* שימוש ב-Debug Shell מאפשר להריץ שורות קוד ב-Java בזמן דיבוג של קוד. למשל אפשר לבצע את השורה מהטיפ הקודם כשעוצרים ב-break-point בעזרת ה-Debug Shell. ה-Debug Shell נמצא בדרך כלל באחד הטאבים לצד הטאב של חלון ה-console, כמו בתמונה הבאה:  
    
  כדי להריץ את השורה. מסמנים אותה ובתפריט שקובץ בלחיצה על הלחצן הימני של העכבר בוחרים ב-Execute.

# חלק 2 – רסטריזציה של משולשים (27)

## תרגיל 2

**א. המתודה calcBoundingBox (9)**  
השלימו את המתודה הריקה calcBoundingBox שבהינתן שלוש נקודות של פאה, מחזירה את המלבן החוסם של הפאה. המתודה מחזירה את המלבן החוסם כוקטור Vector4i. הרכיב הראשון בוקטור הוא minx, השני maxX, השלישי minY, והרביעי maxY בהתאם לתרשים הבא. בידקו את המתודה בעזרת מתודת בדיקות היחידה שלה במחלקה UnitTest.



**ב. תירגול ידני של קאורדינטות בריסנטריות**ציירו לכם על דף את המשולש עם הנקודות הבאות: (5,1), (10,3), (7,4), ואת הנקודה (7,2) והנקודה (7,1). נסו להעריך את הקאורדינטות הבריסנטריות של הנקודות האלו ביחס למשולש.

הקואורדינטות הבריסנטריות הן Alpha:0.09, Beta:0.36, Gamma:0.55.

A graph of a triangle with a line and points

AI-generated content may be incorrect.ולכן הנקודה בתוך המשולש   
  
**ג. הבנה של המחלקה BarycentricCoordinates ודרך השימוש בה (10)**  
אחרי שעשיתם את סעיף ב, נבצע את אותם חישובים בעזרת המחלקה BarycentricCoordinates. התחילו מלקרוא ולהבין את המחלקה BarycentricCoordinates שבחבילה your\_code. התמקדו בשדות של המחלקה, במתודות: הבנאי, calcCoordinatesForPoint, isPointInside, ו-toString ובמתודות העזר הפרטיות: lineFrom2Points, semiDistanceOfPointFromLine (במתודות שעוסקות באינטרפולציה נתמקד בהמשך).  
במתודה main של המחלקה הזו הוסיפו קוד שמחשב את הקאורדינטות הבריסנטריות של הנקודות ביחס למשולש והאם הנקודות בתוך המשולש או מחוצה לו. שימו לב שהבנאי מקבל נקודות תלת מימדיות, אך לצורך חישוב הקאורדינטות הבריסנטריות הוא מתעלם מרכיב z, אז התעלמו ממנו.

**ד. לולאה על כל הפיקסלים של המלבן החוסם ותנאי על הפיקסלים שבתוך הפאה (7 מלבד הקוד שצריך להעתיק מלמטה)**  
בתרגיל הקודם מימשנו רסטריזציה של קוים תחת התנאי של תצוגה מסוג ***FACE\_EDGES*** במתודה rasterization***.*** כל סוגי התצוגה שנממש בהמשך משתמשים ברסטריזציה של פאות משולשות ולכן נממש את הרסטריזציה של פאות משולשות בחלק של ה-else לתנאי של תצוגה מסוג ***FACE\_EDGES***.

עליכם לממש לולאה שעוברת על כל הפיקסלים של המלבן החוסם של הפאה שהקודקודים שלה התקבלו כפרמטרים של המתודה rasterization. לפני הלולאה צרו אובייקט של המחלקה BarycentricCoordinates. בתוך הלולאה בידקו האם הפיקסל הוא בתוך הפאה והוסיפו את הקוד הבא בתוך התנאי. בתוך התנאים האלו נממש את שאר סוגי התצוגה בפרויקט:

FragmentData fragmentData = **new** FragmentData();  
**if** (worldModel.displayType == DisplayTypeEnum.***FACE\_COLOR***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***INTERPOlATED\_VERTEX\_COLOR***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***LIGHTING\_FLAT***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***LIGHTING\_GOURARD***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***LIGHTING\_PHONG***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***TEXTURE***) {  
} else **if** (worldModel.displayType == DisplayTypeEnum.***TEXTURE\_LIGHTING***) {  
}  
Vector3f pixelColor = fragmentProcessing(fragmentData);  
intBufferWrapper.setPixel((**int**) x, (**int**) y, pixelColor);

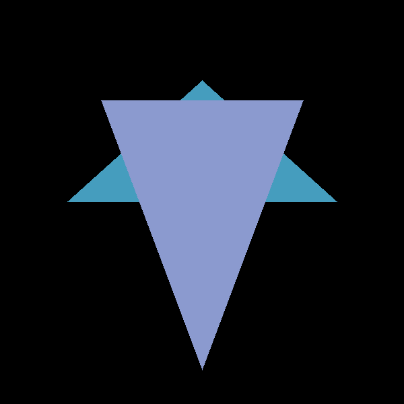
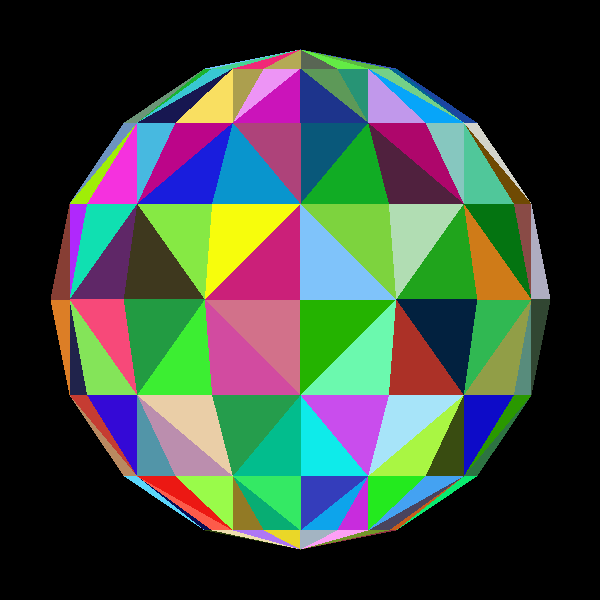
תחת התנאי הראשון (תצוגה מסוג ***FACE\_COLOR)*** עדכנו את fragmentData.pixelColor לצבע שבמשתנה faceColor שמתקבל כפרמטר למתודה.

שימו לב שבסוף התנאים קיימת קריאה למתודה fragmentProcessing ועידכון של הבאפר intBufferWrapper לצבע pixelColor שמחזירה מתודה זו.

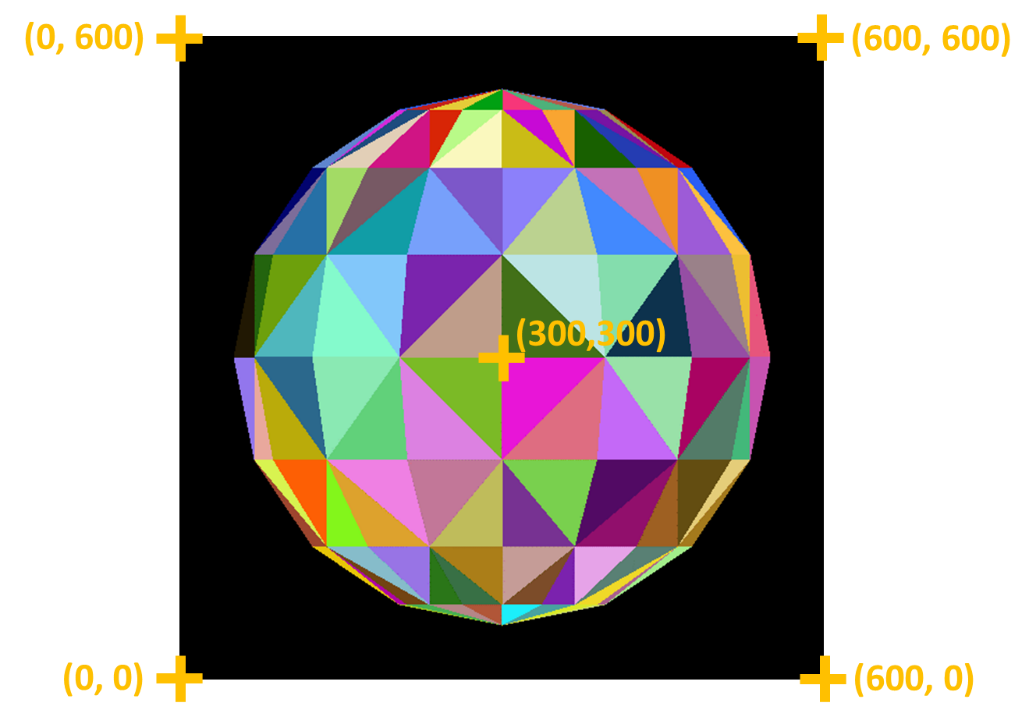
**ה. עידכון המתודה fragmentProcessing (1)**במתודה fragmentProcessing לא קורה הרבה במקרה זה. במתודה fragmentProcessing תחת התנאי:

**if** (worldModel.displayType == DisplayTypeEnum.***FACE\_COLOR***) {

החזירו את fragmentData.pixelColor.

**ו. בדיקה**כדי לבדוק את עבודתכם שנו בתוכנה את סוג התצוגה ל-FACE\_COLOR, וטענו את המודלים שאיתם בדקנו בתרגיל הקודם.   
התוצאה צריכה להראות כך, רק שצבע הפאות יהיה שונה כיוון שהוא נבחר רנדומלי:  
 

# חלק 3 – טרנספורמציות מידול (31)

כיוון שעדיין לא טיפלנו בטרנספורמציה הדרושה מנקודות בתלת מימד למיקום על גבי החלון, בתרגיל זה הציגו בינתיים רק את המודלים: ex\_01\_\_\_FirstExample\_0to600.obj, ex\_01\_\_\_sphere\_0to600.obj, שבתיקייה: Models/models\_0to600.   
המודלים שנשתמש בהם בינתיים הם כאלו שהקאורדינטות בהם מתאימות לקאורדינטות הפיקסלים על גבי החלון ולכן לא נדרשת בנתיים טרנספורמציה מתלת מימד למיקום על גבי החלון. לדוגמה מרכז הכדור במודל שנשתמש בו בנתיים הוא בקאורדינטות x=300 ו-y=300 וכך הכדור יצויר ממורכז במרכז החלון כי קאורדינטות אלו הן המרכז של החלון.  
בתרשים הבא אפשר לראות את הקאורדינטות של פיקסלים שונים בחלון.  


## תרגיל 3.1 (5)

המחלקה WorldModel אחראית על טעינה ו-rendering של כל העולם שממול למצלמה. המתודה load טוענת את האובייקטים והמתודה render של מחלקה זו מבצעת את ה-rendering. כרגע המתודה load טוענת רק אובייקט אחד בשם object1 והמתודה render עושה rendering של אובייקט זה. בתחילת המתודה rendering מתבצע קידום של counter שהוא מונה של מספר הפריימים, איתחול של באפר התמונה, ואיתחול של מטריצות הטרנספורמציה שעובר object1. בסיום המתודה מתבצע rendering של object1.

בתרגיל זה תוסיפו טרנספורמציה שתזיז את האובייקט object1 תזוזה קטנה בכל מחזור לכיוון רנדומלי ביחס למיקום במחזור הקודם. את הקוד הוסיפו תחת התנאי הבא במתודה rendering במחלקה WorldModel:

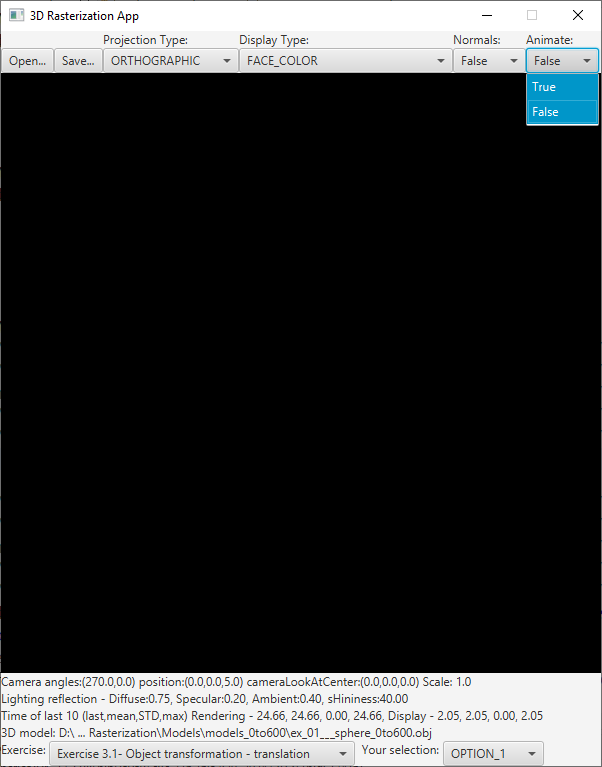
**if** (exercise.ordinal() == ExerciseEnum.***EX\_3\_1\_Object\_transformation\_\_\_translation***.ordinal()) {

הוסיפו את הקוד הבא:

**א. יצירת מטריצת טרנספורמציה (4)**  
צרו מטריצת טרנספורמציה שתזיז את האובייקט לכיוון אקראי ביחס למיקום הקודם. קיבעו את כיוון התנועה להיות אקראי וגודלו כמה פיקסלים לשני הצירים.

**ב. עידכון מטריצת המידול של object1 (1)**  
עדכנו את מטריצת הטרנספורמציה שהאובייקט object1 עובר בעזרת הקריאה למתודה setModelM של האובייקט object1 שמקבלת מטריצה.

הריצו את התוכנית, שנו את התרגיל לתרגיל 3.1 בחלק התחתון של החלון, ושנו את מצב האנימציה ל-True כמו בתמונה הבאה. במצב כזה מתבצע rendering ברצף ולא רק rendering של תמונה בודדת. האנימציה נוצרת הודות לשינוי ב-rendering כפונקציה של המשתנה counter.



## תרגיל 3.2 (3)

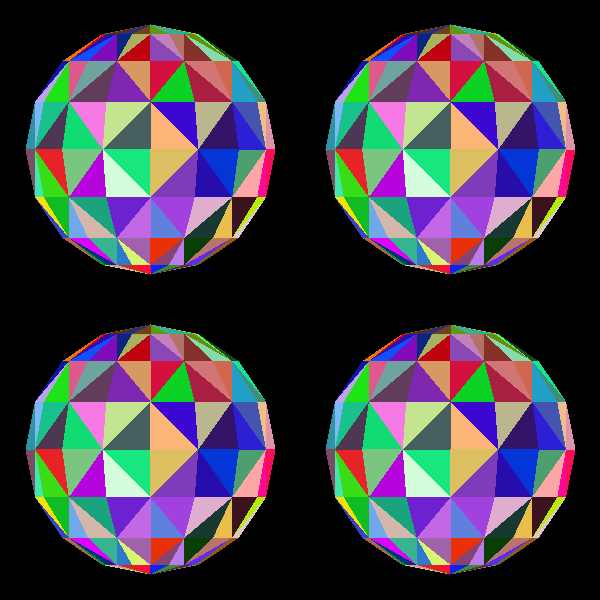
בדומה לתרגיל הקודם את תרגיל זה ממשו תחת התנאי:  
 if (exercise.ordinal() == ExerciseEnum.EX\_3\_2\_Object\_transformation\_\_\_scale.ordinal())

ממשו הגדלה והקטנה של גודל האובייקט באנימציה מגודל של 90% לגודל של 110% מגודל האובייקט המקורי וחוזר חלילה. השתמשו לשם כך במשתנה counter שהוא מונה של מספר הפריימים.

## תרגיל 3.3 (23)

בדומה לתרגיל הקודם את תרגיל זה ממשו תחת התנאי:  
if (exercise.ordinal() == ExerciseEnum. EX\_3\_3\_Object\_transformation\_\_\_4\_objects.ordinal())

כרגע יש רק אובייקט אחד במודל אך אפשר לעשות עליו טרנספורמציות ולעשות לו rendering כמה פעמים. בתרגיל זה נצייר את האובייקט מוקטן פי שניים בארבעה מיקומים שונים, כמו בצילום הבא:



לשם כך עליכם לעדכן בכל פעם את הטרנספורמציה שעובר object1 ע"י קריאה למתודה שלו setModelM ואחר כך לצייר את המודל ע"י קריאה למתודה render שכרגע נקראת רק פעם אחת. עבור הטרנספורמציה האחרונה לא תצטרכו להוסיף קריאה למתודה render כי היא כבר קיימת בסוף המתודה.

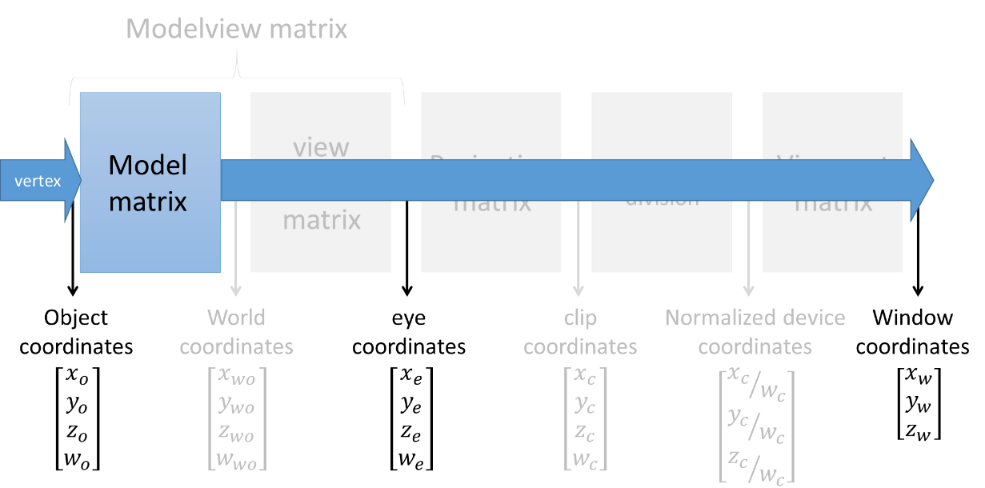
# חלק 4 – הטלה אורתוגרפית וטרנספורמציית viewport (17)

בתרגיל זה תממשו את הטרנספורמציות של קודקודים באובייקט למיקום שלהן בחלון. לשם כך תעדכנו את המתודה vertexProcessing של המחלקה ObjectModel על מנת שתבצע את הטרנספורמציות האלו על קודקודי האובייקט. כמו כן תעדכנו במתודה render של המחלקה WorldModel את מטריצות הטרנספורמציה של המחלקה ObjectModel.

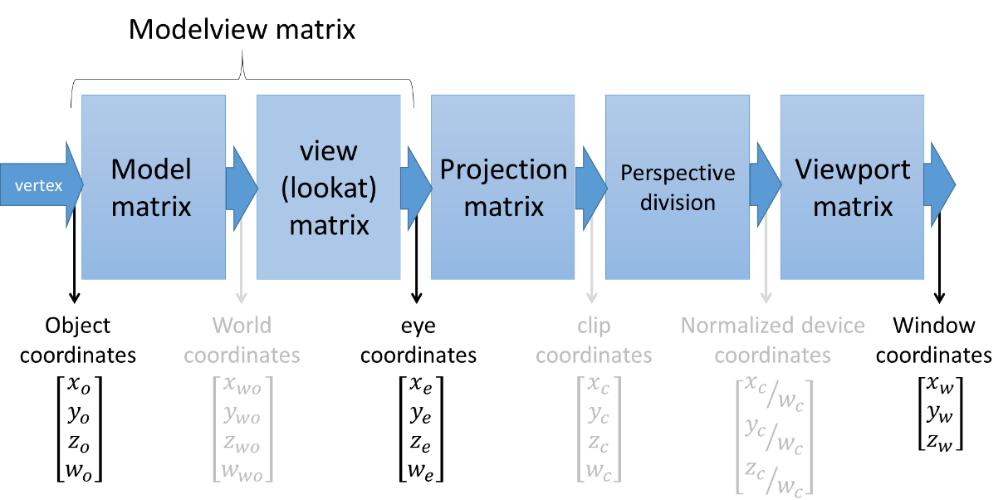
## תרגיל 4

**א. עידכון המתודה vertexProcessing של המחלקה ObjectModel (12)**

המתודה vertexProcessing מחשבת את הטרנספורמציות שעובר כל קודקוד באובייקט. כדי שהתרגיל הקודם על טרנספורמציות יעבוד לנו במתודה vetexProcessing כבר יש חישוב של מיקום הנקודה בקאורדינטות החלון בעזרת ביצוע טרנספורמצית מידול בלבד ועידכון של הנקודה בקאורדינטות העיין לאותו הערך. ראו בשירטוט הבא את מהלך הטרנספורמציות שמתבצע עכשיו במתודה Vector processing.



בתרגיל זה נחשב ונעדכן את קאורדינטות העין ואת קאורדינטות החלון בהתאם לחישוב שבשירטוט הבא.



המתודה vertexProcessing מקבלת כפרמטר אובייקט של המחלקה VertexData. מחלקה זו היא המחלקה שמחזיקה את הנתונים על קודקודים מהאובייקט. השדה pointObjectCoordinates של המחלקה VertexData מציין את המיקום של הקודקוד בקאורדינטות האובייקט (כפי שהוא כתוב בקובץ). בתרגיל זה נחשב במתודה vertexProcessing מהשדה pointObjectCoordinates את השדות pointEyeCoordinates ו-pointWindowCoordinates שמציינים את המיקום של הקודקוד בקאורדינטות העין, ואת המיקום של הקודקוד בקאורדינטות החלון. שדות אלו יהיו בשימוש במתודות rasterization ו-fragmentProcessing.   
חשבו ועדכנו את vertex.pointEyeCoordinates ואת vertex.pointWindowCoordinates כשנתון לכם vertex.pointObjectCoordinates, ומטריצות הטרנספורמציה modelM, lookatM, projectionM, ו- viewportM שהן שדות של המחלקה ObjectModel.

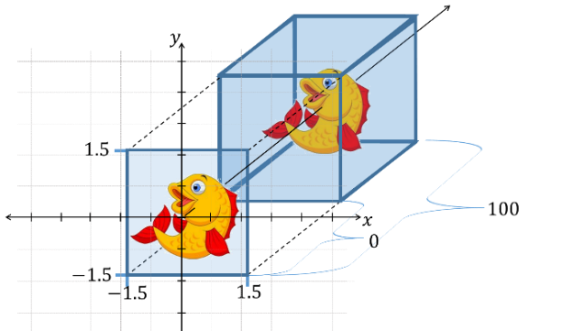
שימו לב שהוקטורים במחלקה VertexData נשמרים כלא הומוגנים ולכן עליכם להעביר להומוגני לצורך הטרנספורמציות ולהחזיר לצורך שמירה. טרנספורמציות ה-modelview לא משנות את רכיב ה-w, ואילו טרנספורמציית ההטלה משנה את רכיב ה-w ולכן אחריה צריך לחלק ב-w כפי שמצוין בתרשים (כשמחלקים ב-w אל תשכחו לחלק גם את הרכיב הרביעי שהוא w בעצמו).

**ב. עידכון מטריצת ההטלה האורתוגרפית במתודה render של המחלקה WorldModel (3)**

במתודה render תחת התנאי:

**if**(projectionType==ProjectionTypeEnum.***ORTHOGRAPHIC***) {

עדכנו את מטריצת ההטלה בעזרת המתודה object1.setProjectionM. היעזרו במתודה ortho של JOML לצורך יצירת המטריצה.  
השתמשו במידול של תיבת ההטלה הבאה:



**ג. עידכון מטריצת ה-viewport במתודה render של המחלקה WorldModel (2)**

במתודה render **אחרי** התנאי:

**if**(projectionType==ProjectionTypeEnum.***ORTHOGRAPHIC***) {  
 … (not here)  
{  
here

עדכנו את מטריצת ה-viewport בעזרת המתודה object1.setViewportM. היעזרו במתודה createViewportMatrix של המחלקה YoursUtilities לצורך יצירת המטריצה.  
חשבו את מטריצת ה-viewport כך שהתמונה שתיווצר תוצג על גבי החלון כולו. רוחב וגובה החלון בפיקסלים נתונים בשדות: imageWidth, imageHeight.

**ד. טענו מודלים לבדיקה.**

עכשיו אחרי שמימשתם טרנספורמצית הטלה ו-viewport תוכלו לראות את כל הקבצים שבתיקיה models\_with\_texture ובתקיה models\_without\_texture, אך לא תוכלו לראות מודלים בתיקיה models\_0to600, כי ה-x וה-y של הנקודות שבהם הוא מ-0 עד 600 אך עכשיו אנחנו מתבוננים רק על תיבה ברוחב 3 ובגובה 3 סביב ראשית הצירים.

# חלק 5 – עדכון נקודת המבט (2)

ממשק המשתמש בנוי כך שניתן יהיה לשנות את נקודת המבט בשתי דרכים. האחת, ע"י לחיצה על העכבר וגרירה לכיוון כלשהוא כשלחצן העכבר לחוץ. בשיטה זו גוררים את המצלמה על גבי כדור ברדיוס 5, שמרכזו בראשית הצירים, כשהמצלמה מתבוננת לכיוון ראשית הצירים. שימו לב שאתם **גוררים את המצלמה ולא את האוביקט**, גרירה ימינה גוררת את המצלמה ימינה כשהאובייקט נשאר במקומו, לכן נראה שהאובייקט מסתובב לכיוון שמאל.  
הדרך השנייה היא לשנות את נקודת המבט בעזרת החצים על גבי המישור z,x. החצים למעלה ולמטה מקדמים את המצלמה קדימה ואחורה, ואילו החצים ימינה ושמאלה מסובבים את כיוון המצלמה ימינה או שמאלה.  
בשלב זה כל עוד לא מימשנו הטלה פרספקטיבית, לא נבחין בתנועה קדימה ואחורה אלה רק בשינוי כיוון נקודת המבט (חישבו למה).

## תרגיל 5

**א. עידכון מטריצת ה-Lookat במתודה render של המחלקה WorldModel (2)**

עדכנו את מטריצת ה-Lookat במתודה render של המחלקה WorldModel אחרי הקוד שמעדכן את מטריצת ה-viewport. ממשק המשתמש כבר מעדכן עבורכם את מיקום המצלמה, הכיוון שאליה היא מתבוננת ואת כיוון הלמעלה: cameraPos, cameraLookAtCenter, cameraUp. עליכם לעדכן את מטריצת ה-LookAt של object1 ע"י המתודה object1.setLookatM. העזרו במתודה lookAt של JOML כדי לבנות את מטריצת ה-lookAt.   
טענו כל אחד מהמודלים בתיקיה models\_without\_textures. שנו את נקודת המבט על ידי גרירה עם העכבר או ע"י תנועה עם החצים.

**ג. טרנספורמציה של נורמלים - בדיקה שחישוב הנורמלים בקואורדינטות העין וציור הנורמלים מתבצע באופן תקין**חישוב הנורמלים של הקודקודים בקאורדינטות העיין ישמש אותנו באחד התרגילים הבאים שבו נבצע חישובי תאורה. טרנספורמציה של נורמל היא מעט שונה מטרנספורמציה של קודקודים, וכיוון שלא למדנו את הנושא הזה, וכדי להקל עליכם בתרגיל, הקוד שמבצע את חישוב הנורמל וגם ההצגה שלו נתונים לכם במתודה transformNormalFromObjectCoordToEyeCoordAndDrawIt שקריאה שלה כבר קיימת בקוד. עליכם רק לבדוק שבמהלך העבודה לא פגעתם בקוד זה. בידקו שהקריאה למתודה נמצאת בסוף המתודה של render אחרי ביצוע הטרנספורמציות. ביחרו בתוכנה בהצגת הנורמלים ובידקו שהם מוצגים באופן תקין.  
למי שמתעניין להלן פירוט של ההסבר של הקוד שנכתב עבורכם:  
יש לחשב את הנורמל בקואורדינטות העין ולעדכן את השדה normalEyeCoordinates של הקודקוד וזאת כדי לאפשר חישובי תאורה. בנוסף יש להוסיף ציור של נורמלים לכל קודקוד אם המשתמש בוחר בזאת.  
כדי לחשב את הנורמל בקואורדינטות העין נדרש להעביר אותו טרנספורמציות. אך צריך לשים לב שהנורמל אינו עובר טרנספורמציות באותה הדרך כמו קודקוד. לדוגמה נורמל של נקודה באובייקט לא משתנה בטרנספורמציית הזזה.   
עבור טרנספורמציה איזומטרית (שאינה מעוותת את צורת האובייקט) חישוב הטרנספורמציה של נורמל הוא פשוט. כדי לחשב את הטרנספורמציה של נורמל בטרנס' איזומטרית יש להכפיל את 3 איברי הנורמל במטריצה 3 על 3 השמאלית עליונה של מטריצת הטרנספורמציה ההומוגנית (4 על 4). (עבור טרנספורמציה לא איזומטרית יש צורך להכפיל ב-inverse transpose אך אין בזה צורך כאן).   
כדי לצייר נורמל עבור קודקוד צריך לחשב נקודה על כיוון הנורמל מהקודקוד ולהעביר את הנקודה הזאת טרנספורמציה של הטלה ו-viewport, ואז ניתן לצייר את הקו של הנורמל בקואורדינטות המסך.

# חלק 6 – הטלה פרספקטיבית (3)

בתרגיל זה תוסיפו אפשרות להטלה פרספקטיבית.

## תרגיל 6

**עידכון מטריצת ההתלה הפרספקטיבית (3)**  
במתודה render של המחלקה WorldModel תחת התנאי:

**if**(projectionType==ProjectionTypeEnum.***PERSPECTIVE***) {

עדכנו את מטריצת ההטלה של object1 ע"י המתודה object1.setProjectionM להטלה פרספקטיבית. היעזרו במתודה perspective של JOML כדי לבנות את מטריצת ההטלה הפרספקטיבית. ביחרו ב-FOV של 30 מעלות וב-zNear ו-zFar של 1 ו-100 בהתאמה. שימו לב שהמתודה מצפה לקבל את fovy ברדיאנים. ה-aspect הוא היחס בין הרוחב לגובה של החלון ובמקרה שלנו הוא יהיה 1 כיוון שהרוחב והגובה של החלון זהים.

# חלק 7 – אינטרפולציה (13)

לצורך תרגיל זה ניתנה לכל קודקוד באובייקט תכונה של צבע. בתרגיל זה נבצע אינטרפולציה בין הצבעים של הקודקודים וכל פיקסל במשולש יקבל את הצבע שהוא אינטרפולציה של שלושת הצבעים של קודקודי המשולש.

## תרגיל 7

**א. תירגול ידני של אינטרפולציה בעזרת קאורדינטות בריסנטריות**ציירו לכם על דף את המשולש עם הנקודות הבאות: (0,0), (10,0), (5,10), ואת הנקודה (5,0) והנקודה (5,5). נסו להעריך את הקאורדינטות הבריסנטריות של הנקודות האלו ביחס למשולש.   
אם נצמיד את המספרים הבאים 10,30,60 לקודקודים לפי הסדר, נסו להעריך מה תהיה האינטרפולציה של ערכים אלו בשתי הנקודות שצוינו.  
  
**ב. תירגול ביצוע אינטרפולציה בעזרת המחלקה BarycentricCoordinates (10)**אחרי שעשיתם את סעיף ב, נבצע את אותם חישובים בעזרת המחלקה BarycentricCoordinates. התחילו מלקרוא ולהבין את שלושת המתודות interpolate שבסוף המחלקה BarycentricCoordinates שבחבילה your\_code.   
במתודה main של המחלקה הזו הוסיפו קוד שמחשב את האינטרפולציה שחישבתם בסעיף הקודם.

**ג. עידכון המתודה rasterization (2)**  
במתודה rasterization תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.INTERPOlATED\_VERTEX\_COLOR) {

בצעו אינטרפולציה של הצבעים של קודקודי המשולש שבשדה color של VertexData. עדכנו את השדה fragmentData.pixelColor בצבע האינטרפולציה שהתקבל.

**ד. עידכון המתודה fragmentProcessing (1)**במתודה fragmentProcessing לא קורה הרבה במקרה זה כי את עבודת האינטרפולציה עשתה הרסטריזציה. במתודה fragmentProcessing תחת התנאי:

**if** (worldModel.displayType == DisplayTypeEnum. ***INTERPOlATED\_VERTEX\_COLOR***) {

החזירו את fragmentData.pixelColor.

**ה. בדיקה**  
פיתחו את המודל models\_without\_textures\triangle1.obj, הקודקוד השמאלי התחתון אדום, הימני ירוק, והעליון כחול, כשהפיקסלים בפאה הם אינטרפולציה בין הקודקודים כמו בצילום הבא:



תוכלו לפתוח גם מודלים אחרים, אך המודל הזה מראה בצורה היפה ביותר את האינטרפולציה בין הצבעים של הקודקודים.

# חלק 8 – Z-Buffer (7)

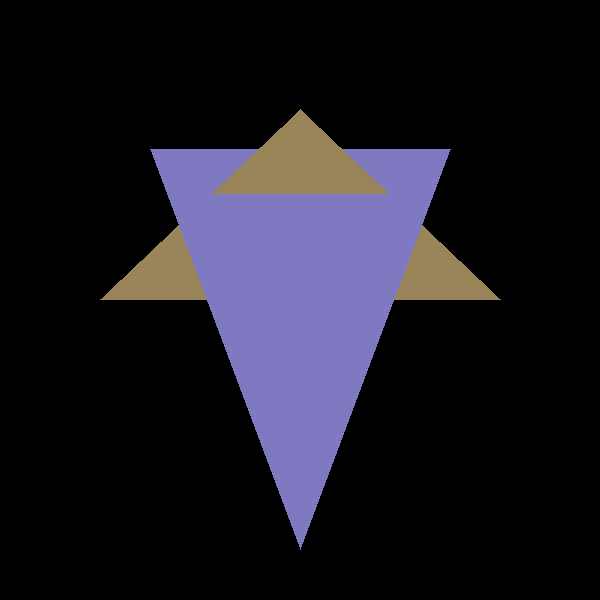
בתוכנה שיצרנו פאות מסתירות אחת את השניה לפי הסדר שבו הן צוירו ולא לפי העומק שלהן. כדי שחלקים קרובים של האובייקטים יסתירו חלקים רחוקים, נשתמש באלגוריתם ה-Z-Buffer.   
בסוף המתודה rasterization שבמחלקה ObjectModel, אנחנו קוראים למתודה fragmentProcessing ומעדכנים את באפר התמונה בצבע של הפיקסל שחושב. נוסיף תנאי שיבצע זאת רק במקרה שבו הפיקסל שאנחנו רוצים לצייר קרוב מהפיקסל שכבר מצויר.

## תרגיל 8

**א. בדיקה לפני העידכון**  
פיתחו את הקבצים: FirstExample.obj ואת המודל sphere.obj מהתיקיה models\_without\_textures.   
שנו את נקודת המבט מסביב למודל וצפו במה שקורה.

**ב. אינטרפולציה של רכיב העומק של הנקודות בקאורדינטות החלון (2)**את הקוד החדש נוסיף בסוף המתודה rasterization שבמחלקה ObjectModel לפני הקריאה ל-fagramentProcessing ועידכון באפר התצוגה עם המתודה intBufferWrapper.setPixel,   
התחילו מלחשב את רכיב ה-z של הפיקסל שאנחנו רוצים לצייר אותו, על סמך רכיב ה-z של מיקום הקודקודים בקאורדינטות המסך.

**ג. בדיקה אם לחשב ולצייר את הפיקסל (5)**  
בידקו אם רכיב ה-z של הפיקסל שאנחנו רוצים לצייר (שחישבתם ב-ב) הוא קטן יותר מרכיב ה-z של הפיקסל שכרגע מצויר ע"י בדיקה ב-Z-Buffer במיקום המתאים (worldModel.zBuffer). אם רכיב ה-z של הפיקסל שרוצים לצייר קרוב מהרכיב של הפיקסל שכבר מצויר בבאפר התמונה, עדכנו את באפר התמונה ואת ה-Z-Buffer. אחרת הפיקסל שמצויר כבר מסתיר את הפיקסל שאנחנו רוצים לצייר ולכן לא נחשב את ערכו ולא נצייר אותו.  
שימו לב שב-Java מקובל שבמטריצות דו-ממדיות האינדקס הראשון הוא של מספר השורה והשני של מספר העמודה. כך שכדי לקרוא את ערך ה-z של הפיקסל במיקום x,y צריך לגשת למטריצה במיקום worldModel.zBuffer[y][x].

**ד. בדיקה**  
שימו לב שיש לאפס גם את באפר התמונה וגם את ה-Z-Buffer בתחילת ה-rendering. זה נעשה בתחילת המתודה render של WorldModel.  
טענו את המודל: FirstExample.obj מהתיקיה models\_without\_textures. הוא אמור להראות כך בתצוגה מסוג FACE\_COLOR. בידקו גם עבור מודלים אחרים בתיקיה שעכשיו הפיקסלים הקרובים מסתירים את הרחוקים.  


# חלק 9 – תאורה (27)

בתרגיל זה נוסיף צללים של תאורה לאובייקט. נעשה זאת בשלוש שיטות shading: Flat, Gouraud, Phong. את התרגיל תוכלו לבדוק על כל המודלים בתיקיה models\_without\_textures.

## תרגיל 9.1 – חישוב תאורה Flat Shading (10)

**א. השלמת המתודה lightingEquation**השלימו את המתודה lightingEquation במחלקה ObjectModel. תוכלו פשוט להעתיק את המתודה שכתבתם בפרויקט RayTracing במחלקה WorldModel. כדי להקל בפרויקט זה חישוב התאורה הוא ברמות אפור ולא בצבע. כדי לחסוך בכתיבת קוד ישנה מתודה שמבצעת העמסה למתודה lightingEquation שמבצעת את חישובי התאורה ברמות אפור ע"י קריאה למתודה שכבר כתבתם.

**ב. חישוב של מיקום התאורה בקאורדינטות העיין (3)**חישוב התאורה מתבצע בקאורדינטות העיין. כשמשנים את נקודת המבט, מיקום האור בקאורדינטות העיין משתנה, ולכן נדרש לעשות טרנספורמציה על מיקום מקור האור. ניתן לעשות טרנספורמציה זו פעם אחת עבור כל האובייקט ולא עבור כל קודקוד פאה או פיקסל ולכן נוסיף אותה בתחילת המתודה render של ObjectModel.

בצעו טרנספורמציה של מיקום מקור האור - worldModel.lightPositionWorldCoordinates לקאורדינטות העין ועדכנו את השדה lightPositionEyeCoordinates. השתמשו במטריצה שינוי נקודת המבט לשם כך (lookatM). שימו לב שתצטרכו וקטור הומוגני לשם הטרנספורמציה, ואחר כך תצטרכו לעבור חזרה לוקטור לא הומוגני.

**ג. חישוב התאורה במתודה rasterization (6)**חישוב תאורה בשיטת ה-flat shading היה מבוצע בתקופה שלפני שה-graphics pipeline שמקובל היום ולכן הוא אינו מתאים כל כך למבנה שלו. ב- flat shadingחישוב התאורה מתבצע פעם אחת לפאה. המתודה היחידה שנקראת פעם אחת לפאה היא rasterization ולכן נעשה את החישובים בה למרות שזה לא התפקיד המקובל של rasterization.  
לפני הלולאה על הפיקסלים במתודה rasterization קיראו למתודה lightingEquation כדי לחשב את התאורה בפאה ועדכנו משתנה מקומי.  
במתודה rasterization תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.**LIGHTING\_FLAT**) {

עדכנו את המשתנה fragmentData.pixelIntensity0to1.   
כמה פרטים:  
הנקודה עליה מחשבים את התאורה - כיוון שמחשבים את התאורה של פאה שלמה אפשר להתיחס לאחד הקודקודים כנקודה עליה מחשבים את התאורה.   
הנורמל של הפאה - נתון במשתנה faceNormal שחושב כמכפלה וקטורית של צלעות הפאה.  
מקדמי חישוב התאורה – נתונים בשדות: worldModel.lighting\_Diffuse, worldModel.lighting\_Specular, worldModel.lighting\_Ambient, worldModel.lighting\_sHininess

**ד. עידכון המתודה fragmentProcessing (1)**גם כאן לא קורה המון במתודה fragmentProcessing. המתודה מקבלת את ערך רמות האפור של התאורה שחושב עבור הפיקסל וצריכה להחזיר וקטור תלת מימדי עם רמת אפור בצבע זה.   
במתודה fragmentProcessing תחת התנאי:

**if** (worldModel.displayType == DisplayTypeEnum.**LIGHTING\_FLAT**) {

החזירו Vector3f שכל ערכיו הם fragmentData.pixelIntensity0to1.

**ד. בדיקה**  
שימו לב שאתם יכולים לשנות את מקדמי ההחזרה של החומר על ידי המקשים: a ו-z למקדמי התאורה האמביינטית, s ו-x למקדמי התאורה הספקולרית, d ו-c למקדמי התאורה הדיפיוסיבית.

## תרגיל 9.2 – חישוב תאורה Gourard Shading (7)

א. **חישוב התאורה של כל קודקוד (3)**המתודה rasterization נקראת עבור כל פאה. כיוון שכל קודקוד הוא חלק מכמה פאות, אם נעשה את החישוב של התאורה על כל קודקוד במתודה זו נבצע את החישוב עבור כל קודקוד בכל הפאות שבהם הוא נמצא. כדי לחסוך נבצע את חישוב התאורה על כל קודקוד פעם אחת במתודה vertexProcessing.  
בסוף המתודה vertexProcessing (אחרי הטרנספורמציה של הנורמל) חשבו את התאורה עבור הקודקוד ועדכנו את השדה vertex.lightingIntensity0to1. שימו לב שמיקום הקודקוד והנורמל בקאורדינטות העיין נתונים כשדות של vertex.

ב. **חישוב התאורה** **במתודה rasterization (3)**במתודה rasterization תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.**LIGHTING\_GOURARD**) {

בצעו אינטרפולציה של התאורה שחושבה עבור כל קודקוד לפיקסל העכשוי ועדכנו את fragmentData.pixelIntensity0to1.

**ג. עידכון המתודה fragmentProcessing (1)**גם כאן לא קורה המון במתודה fragmentProcessing. המתודה מקבלת את ערך רמות האפור של התאורה שחושב עבור הפיקסל וצריכה להחזיר וקטור תלת מימדי עם רמת אפור בצבע זה.   
במתודה fragmentProcessing תחת התנאי:

**if** (worldModel.displayType == DisplayTypeEnum.**LIGHTING\_GOURARD**) {

החזירו Vector3f שכל ערכיו הם fragmentData.pixelIntensity0to1.

## תרגיל 9.3 – חישוב תאורה Phong Shading (10)

**א. אינטרפולציה של מיקום הנקודה והנורמל לכל פיקסל במתודה rasterization (4)**במתודה rasterization תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.LIGHTING\_PHONG) {

בצעו אינטרפולציה של מיקום הנקודה והנורמל בקאורדינטות העיין של שלושת הקודקודים והציבו בשדות: fragmentData.pointEyeCoordinates, fragmentData.normalEyeCoordinates. שימו לב שלמחלקה BarycentricCoordinates יש כמה מתודות לאינטרפולציה, ביניהן כאלו שעושות אינטרפולציה על וקטורים.

**ב. חישוב התאורה במתודה fragmentProcessing (6)**במתודה fragmentProcessing תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.LIGHTING\_PHONG) {

השתמשו במיקום הנקודה והנורמל בקאורדינטות העיין שחישבתם בסעיף הקודם שמועברים בשדות: fragmentData.pointEyeCoordinates, fragmentData.normalEyeCoordinates.  
כדי לחשב את התאורה של הפיקסל והחזירו וקטור תלת מימדי שכל איבריו עם רמת האפור שהתקבלה.

# חלק 10 – טקסטורה

## תרגיל 10.1 – טקסטורה ללא תאורה (6)

א. **אינטרפולציה של קאורדינטות הטקסטורה לכל פיקסל במתודה** **rasterization (2)**במתודה rasterization תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.**TEXTURE**) {

בצעו אינטרפולציה של קאורדינטות הטקסטורה של שלושת הקודקודים ועדכנו את השדה: fragmentData.textureCoordinates.

ב. **חישוב צבע הטקסטורה של הפיקסל במתודה fragmentProcessing (4)**במתודה fragmentProcessing תחת התנאי:

if (worldModel.displayType == DisplayTypeEnum.**TEXTURE**) {

מיצאו את הצבע של הטקסטורה בקאורדינטה fragmentData.textureCoordinates והחזירו צבע זה.  
שימו לב שהקאורדינטות של טקסטורה הן בין 0 ל-1. כדי להשתמש בהן כדי לגשת לפיקסל בתמונת הטקסטורה תצטרכו להכפיל את הקאורדינטות ברוחב וגובה תמונת הטקסטורה פחות אחד (כי האינדקס מתחיל באפס).  
תמונת הטקסטורה מאוחסנת בשדה textureImageIntBufferWrapper. את רוחב הטקסטורה וגובה הטקסטורה אפשר לקבל ע"י השדות: getImageWidth(), ו-getImageHeight(). פיקסל בתמונת הטקסטורה אפשר לקבל ע"י המתודה getPixel.  
שימו לב שהאינטרפולציה של הקאורדינטות מחזירה מיקום מדוייק ב-float. הדרך הקלה ביותר היא לעגל ולגשת לפיקסל הקרוב, וכך תעשו, אך ניתן גם לעשות אינטרפולציה ולקבל פיקסל יותר מדויק.

## תרגיל 10.2 – טקסטורה עם תאורה Phong Shading (15)

חשבו את הצבע של הטקסטורה בפיקסל כפי שעשיתם בתרגיל הקודם (10.1).   
חשבו את התאורה בפיקסל לפי שיטת Phong Shadingכפי שעשיתם בתרגיל 9.3.  
למען הפשטות החזירו מהמתודה fragmentProcessing את המכפלה בין צבע הטקסטורה שחישבתם לרמת האפור של התאורה שחישבתם.