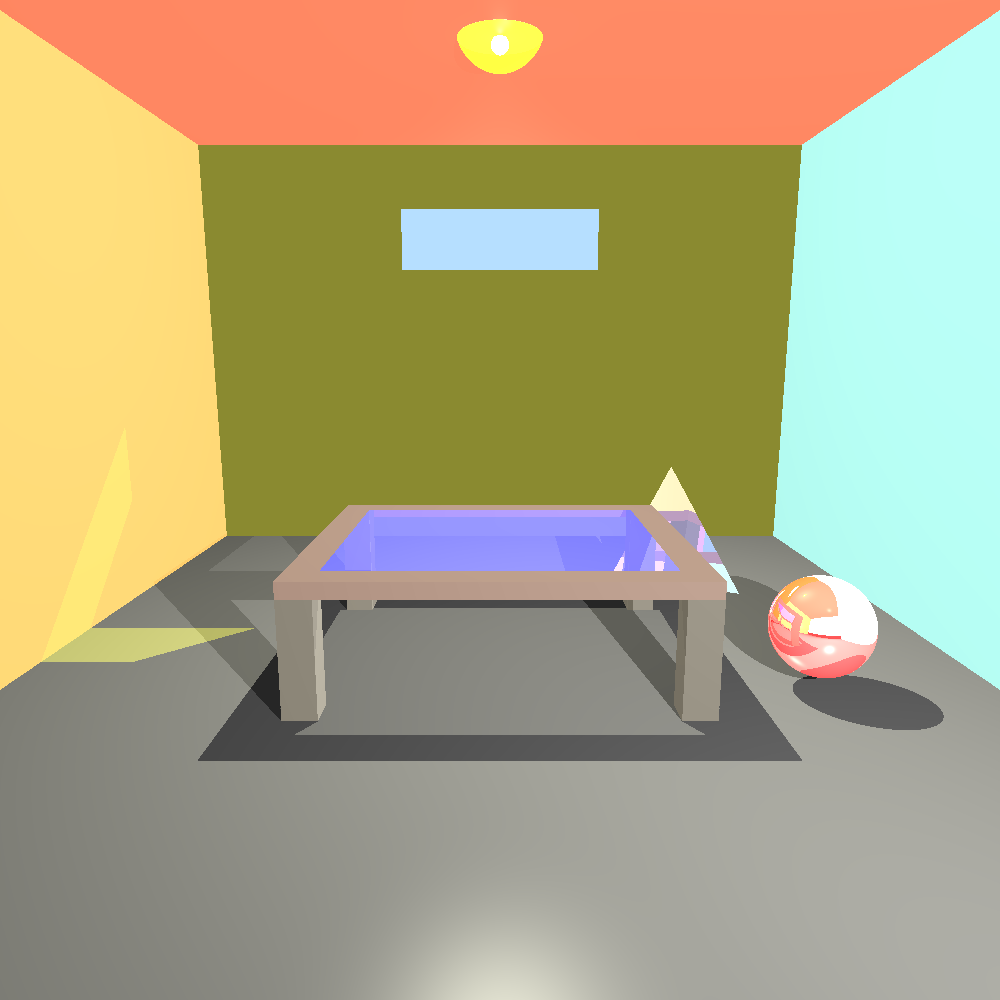
**דוח מיניפרוייקט 1 – שיפור תמונה** מגישים: נתנאל גלאובר וירון גרסטל

התמונה אותה ברצוננו לשפר היא זאת:

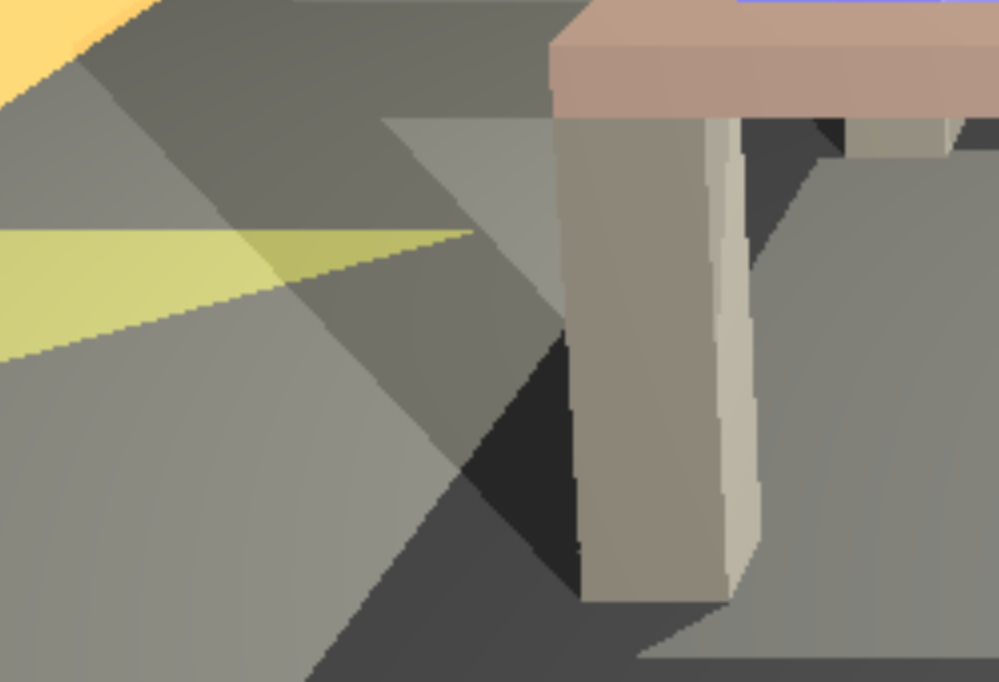
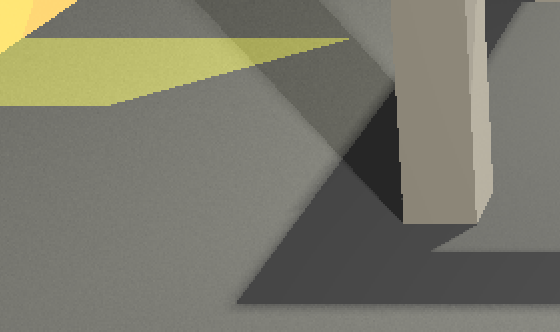


נשים לב לצל החד שקיים , צל זה אינו אמין שכן המעבר בין מקום מוצל למקום שאינו מוצל הינו מעבר מדורג ואצלנו אין זה מתקיים. לכן שיפרנו ע"י הוספת צל רך:

התמונה לאחר השיפור:

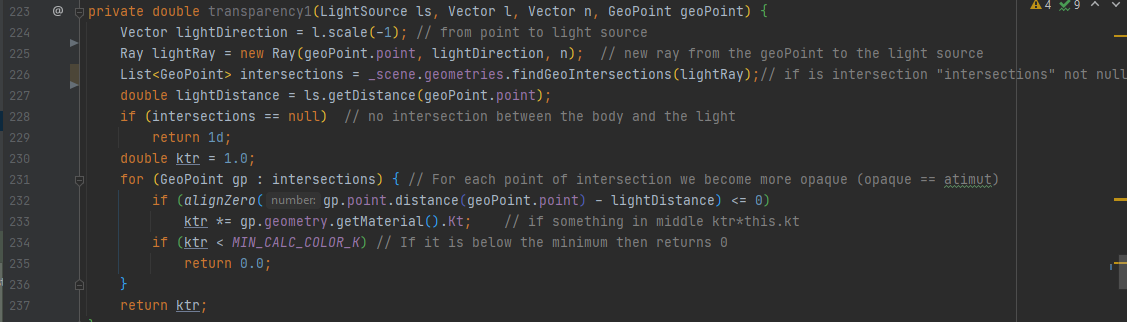


נדגיש את ההבדלים:

לפני: אחרי:

ניתן לראות את ההבדל שנעשה, ואת השוני בין השינוי החד בצל לבין הרך.

כעט נעבור לקוד ע"מ להסביר איך מימשנו את עקרון הצל הרך:

הקוד לפני השינויים:

מטרת הפונקציה הזו, עבור כל geoPint שנשלחת להחזיר לפונקציית חישוב הצבע בנקודה(calcLocalEffects), את כמות האור המגיעה לנק ממקור האור הנתון כרגע.(הכפלה של כל הKT שבדרך ממקור האור אל הנקודה)

הסבר לפונקציה:

שורות 224-227:

ניצור וקטור מהנקודה אל מקור האור. ע"י וקטור זה ניצור קרן מהנקודה על המשטח שקיבלנו ווקטור זה, כלומר יצרנו קרן מהנקודה שמגיעה למקור האור.

כעט ניצור ליסט המכיל את כל הנקודות אותם הקרן שלנו חוצה בדרך למקור האור.

וכן נחשב את המרחק מהנקודה עד למקור האור.

שורה 228:

ווידוא שאכן הקרן חוצה איזשהו גאומטריה, אחרת חוזר הערך KTR=1 כלומר שקיפות מלאה.(אין צל)

שורות 230-236:

כעט, ניצור ערך התחלתי מקסימלי עבור KTR(1), ולאט לאט בהמשך החישובים ערך זה יקטן

לולאה: עבור כל נקודה מבין נקודות החיתוך של הקרן שיצרנו עם הגופים:

ראשית נוודא שאכן נקודת חיתוך זו אכן בין תחילת הקרן(הנקודה שלנו) למקור האור, כלומר בין הנקודה שלנו לנקודת החיתוך קטן מהמרחק שחישבנו בין הנק' שלנו למקור האור.

* אם אכן כך: נכפיל את KTR שלנו בKT של אותה נקודת חיתוך, כך KTR יקטן ביחס לשקיפות שיש לאותה נקודה.

שנית נבדוק אם KTR שלנו קטן ממספר מינימלי שהגדרנו (0.001) עבורו כל חישוב נוסף יהיה מיותר..

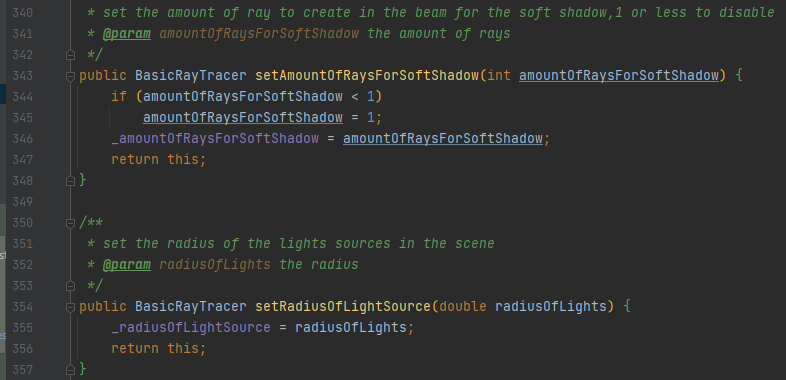
* אם אכן כך יחזיר 0=KTR חזרה לפונקציית חישוב הצבע המקומי בנקודה.

במידה מKTR לא קטן מאותו מספר מינימלי , ובנוסף סיימנו לעבור את כל נק' החיתוך – יוחזר KTR.

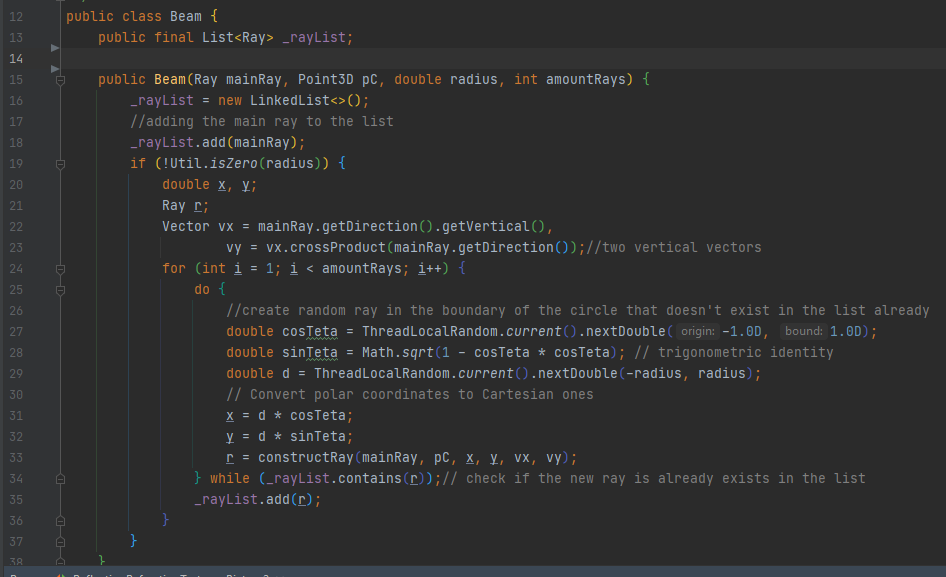
השיפור:

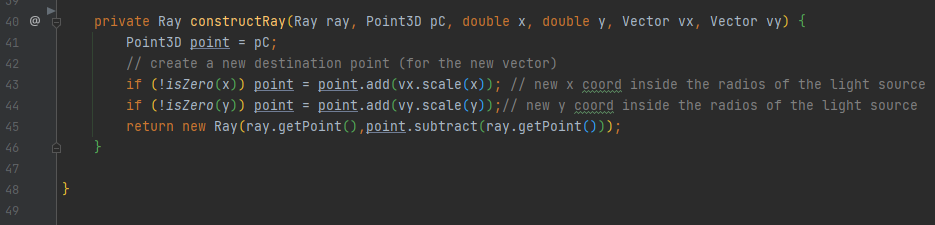
הרעיון העומד מאחורי השיפור הוא להרחיב את מקור האור למעגל בעל רדיוס מסוים, ולא נק' יחידה, ולשלוח כמות קרניים מכל רחבי המעגל אל ה geopoint . כך, אם עבור קרן מסוימת ייווצר צל ועבור קרן שנמצאת בקרבה אליה לא ייווצר צל, נחשב לבסוף את הצבע הממוצע עבור כל הקרניים ונקבל צבע ביניים בין צל לאור באותה נקודה. כך ייווצר האפקט של הצל הרך.

לצורך כך יצרנו כמה פונקציות:

מכיוון שכל הפרוייקט בנוי על builder design ניצור 2 פונקציות SETERS שמקבלות את רדיוס האור ואת כמות הקרניים שרצויה:

בנוסף בכדי לאפשר את חישוב הקרניים הצמודות יצרנו מחלקה BEAM ומיקמנו אותה בחבילת primitives:





* כמובן שכתבנו JAVADOC אך מחקנו עבור הדוח.

למחלקה זו ישנה תכונה המכילה רשימה של קרניים.

הבנאי ממלא את הרשימה הנ"ל בקרניים עם זווית מעט שונה בכל פעם , אך כולן מגיעות לאותה geopoint, ממקומות שונים על רדיוס האור.

שורה 18:

ראשית נכניס את הקרן הראשית, זו המגיעה מGEOPOINT שלנו אל עבר אמצע מקור האור.(בקריאה לפונקציה, זו אנו שולחים קרן מהGEOPOINT אל מרכז מקור האור)

לאחר ווידא שאכן מקור האור הינו בעל רדיוס שגדול מ0, נגדיר 2 וקטורים המאונכים לקרן הראשית וגם מאונכים בניהם.

שורה 24:

אז ניכנס ללולאה שרצה כמספר הקרניים שקיבלנו בפרמטרים:

ובכל איטרציה מגרילה מספר בין -1 ל 1 (COSTETA) ומתוך זה נחשב את SINTETA ע"י

הזהות: לאחר מכן נגריל מספר בין -R לR .

הזהות מאפשרת לנו לקבל נקודה רנדומלית על **היקף** מעגל היחידה, ואילו המספר ש בין -R לR בהכפלה עם הנקודה על היקף מעגל היחידה ייתן לנו נקודה רנדומלית **בתוך** מעגל האור.

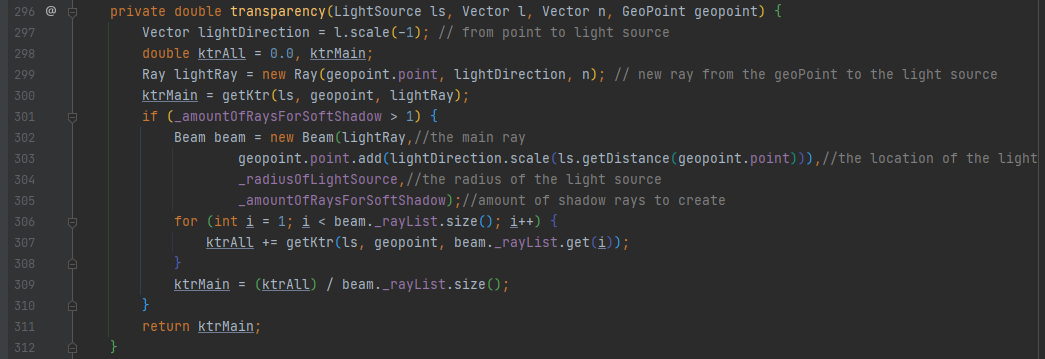
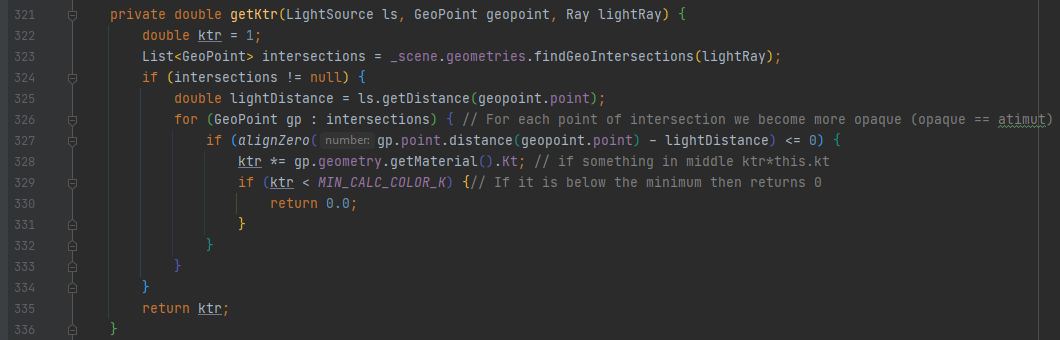
לאחר שקיבלנו ערכי X ו Y של נקודה נשלח לפונקציה constructRay את כל המרכיבים כדי לקבל נקודה רנדומלית בתוך המעגל של **מקור האור**: אנו שולחים את X ו Y , ואת 2 הוקטורים המאונכים לקרן הראשית, וכן את נקודת המרכז של מקור האור.

ע"י הכפלה של X בווקטור VX נקבל ערך של X על מעגל התאורה

ע"י הכפלה של Y בווקטור VY נקבל ערך של Y על מעגל התאורה,

כך נקבל נקודה רנדומלית שאליה תחושב קרן חדשה : מהGEOPOINT אל נקודה רנדומלית זו.

לאחר מכן בשורה 34 נוודא שלא קיבלנו קרן שכבר קיימת ונכניס את הקרן שחישבנו לליסט הקרניים.

כעט, לאחר שעברנו על פונקציות העזר, נראה את השינוי שעשינו בפונקציה transparency

\*כמובן שכתבנו JAVADOC גם עבור פונקציות אלה, מפאת חיסכון במקום לא העתקנו אותם.

ראשית:

ניצור קרן מהGEOPOINT אל מרכז מקור האור.

כמו כן, ניצור KTRALL עבור סכימת כל הצבעים בנקודה, שיאותחל ל0.0.

וכן KTRMAIN שערכו יהיה הKTR של הקרן הראשית(זו שמה-geopoint למרכז מקור האור)

חישוב הKTR משתמש בפונקציית עזר: GETKTR.

לאחר מכן, בהנחה שקיבלנו ערך של כמות קרניים הגדולה מ1, ניצור רשימה של קרניים,ע"י מחלקת beam:

נשלח את הקרן מהgeopoint אל מקור האור,

נחשב את מרכז האור, ע"י חישוב המרחק מהאור לנקודה, נכפיל בכיוון הוקטור וכל זה נוסיף לנקודה כך נקבל את מרכז מקור האור.

נשלח את רדיוס מקור האור וכמות הקרניים.

כעט נעבור על כל קרן וקרן מתוך הליסט שקיבלנו ממחלקת הBEAM ונסכום את כל הKTR של כל ה קרניים.(שורה 306-307)

לבסוף בשורה 309:

נחשב את ממוצע הצבע KTR בנקודה, ע"י חילוק KTRALL בכמות הקרניים.

ואם לא קיבלנו כמות קרניים גדולה מ-1 מחזיר KTRMAIN שחושב בשורה309.

וכך נקבל מקדם KR עבור כל GEOPOINT שנשלחת לפונקציית transparency , שבמידה ונשלחה כמות קרניים גדולה מאחד, מתחשב בכמות קרניים המופקות מרדיוס מסוים ולא מנק' אחת, ובמידה ולא , מחשבת בצורה רגילה עבור קרן אחת.

דוח מיניפרוייקט 2 – שיפור BVH + שיפור ע"י multithreads

הרעיון מאחורי ה multithreads לפצל את סך הפיקסלים לכמות תהליכונים שהמשתמש בוחר, וככה נוצרת פעולה מקבילית והזמן אמור להתקצר משמעותית.

בנוסף, הרעיון שעומד מאחורי BVH - **bounding volume hierarchy** הוא להכניס כל גוף לקופסא, וקבוצת גופים לקופסה ג"כ, ואז במקום לחשב עבור כל קרן מחדש את הצבע ע"י חישוב החיתוך של הקרן עם הגוף, אם הקרן חוצה את הקופסא נחשב עבורה את הצבע שמתקבל, ואם הקרן אינה חוצה את הקופסא נדלג על החישוב הצבע בחיתוך עם אותו גוף. כך נחסוך הרבה מאוד חישובים ברנדור.

התמונה:

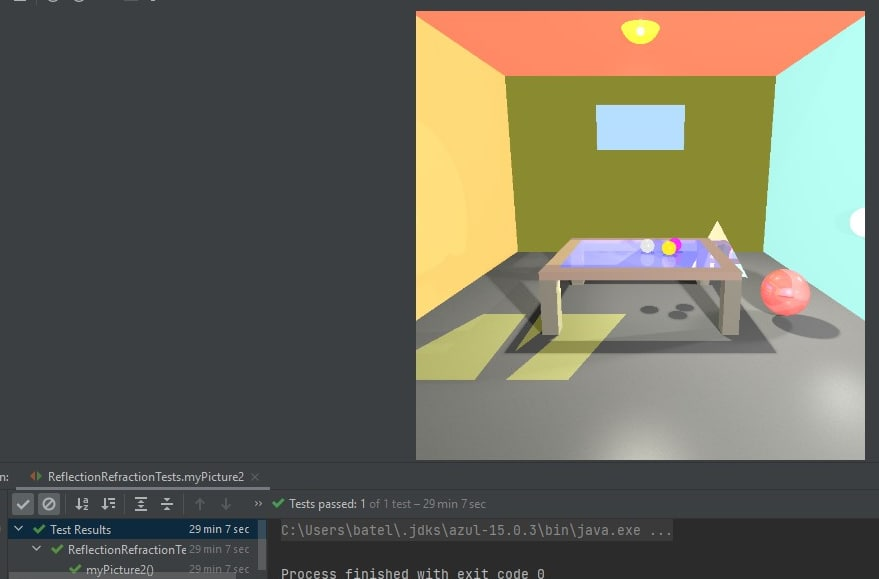


זמני רנדור ללא BVH וללא טרדים:

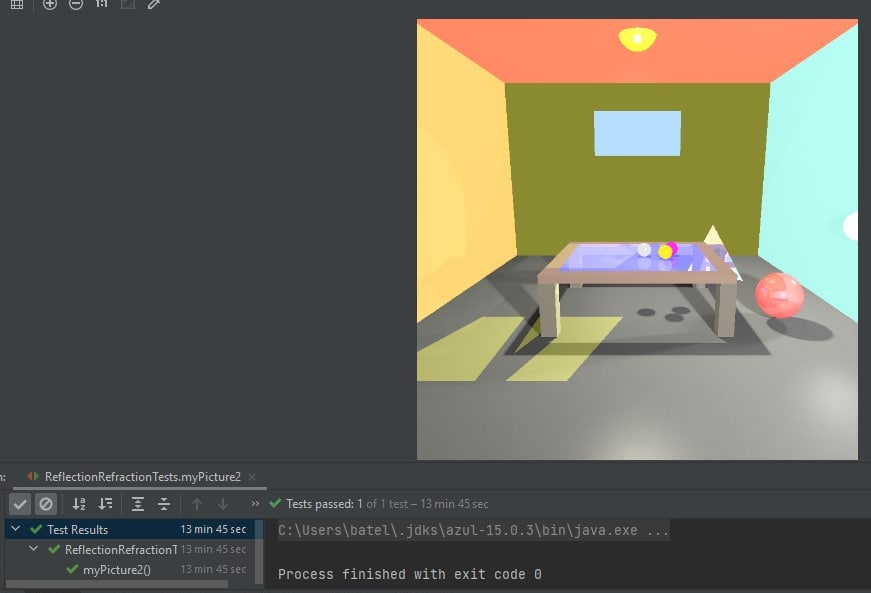
תמונה שמכילה טקסט, צילום מסך, אלקטרוניקה, מחשב

התיאור נוצר באופן אוטומטי

\*תמונה עם BVH



\*תמונה עם טרדים וBVH



הקוד:

לגבי ה multithread זהו קוד שקיבלנו מדן נסביר מעט את הקוד:

public Render setMultithreading*(*int threads*) {* if *(*threads < 0*)* throw new IllegalArgumentException*(*"Multithreading parameter must be 0 or higher"*)*;  
 if *(*threads != 0*)* this.threadsCount = threads;  
 else *{* int cores = Runtime.*getRuntime()*.availableProcessors*()* - *SPARE\_THREADS*;  
 this.threadsCount = cores <= 2 ? 1 : cores;  
 *}* return this;  
*}*

כאן מופיע פונקציה שמקבלת מהמשתמש את כמות התהליכונים שהוא רוצה להפעיל עבור פעולת הרנדור. הערך ברירת המחדל הוא 1. ואם המשתמש מכניס 0 , אז מוקצה כמות טרדים פחות 2.

public void renderImage*() {* if *(*\_imageWriter == null*)* throw new MissingResourceException*(RESOURCE\_ERROR*, *RENDER\_CLASS*, *IMAGE\_WRITER\_COMPONENT)*;  
 if *(*\_camera == null*)* throw new MissingResourceException*(RESOURCE\_ERROR*, *RENDER\_CLASS*, *CAMERA\_COMPONENT)*;  
 if *(*\_rayTracerBase == null*)* throw new MissingResourceException*(RESOURCE\_ERROR*, *RENDER\_CLASS*, *RAY\_TRACER\_COMPONENT)*;  
  
 final int nX = \_imageWriter.getNx*()*;  
 final int nY = \_imageWriter.getNy*()*;  
 if *(*threadsCount == 0*)* for *(*int i = 0; i < nY; ++i*)* for *(*int j = 0; j < nX; ++j*)* castRay*(*nX, nY, j, i*)*;  
 else  
 renderImageThreaded*()*;  
*}*

בפונקציה זו לאחר כל הבדיקות במידה וכמות הטרדים הינה 0 המעבר על הפיקסלים הוא מעבר לינארי, אחד אחד. במידה ויש יותר מ 0 , מריץ פונקציה יעודית:

nX מקבל את סך השורות

nY מקבל את סך העמודות

נוצר משתנה מסוג פיקסל המאותחל עם ערכי המקסימום של השורות והעמודות בתמונה.

ואז אנו מקצים לכל טרד את כלל הפיקסלים, ולאחר מכן מפעילים את כולם, כך שהטרד הפנוי יעשה את הפיקסל הבא.

private void renderImageThreaded*() {* final int nX = \_imageWriter.getNx*()*;  
 final int nY = \_imageWriter.getNy*()*;  
 final Pixel thePixel = new Pixel*(*nY, nX*)*;  
 // Generate threads  
 Thread*[]* threads = new Thread*[*threadsCount*]*;  
 for *(*int i = threadsCount - 1; i >= 0; --i*) {* threads*[*i*]* = new Thread*(()* -> *{* Pixel pixel = new Pixel*()*;  
 while *(*thePixel.nextPixel*(*pixel*))* castRay*(*nX, nY, pixel.col, pixel.row*)*;  
 *})*;  
 *}* // Start threads  
 for *(*Thread thread : threads*)* thread.start*()*;  
  
 // Print percents on the console  
 thePixel.print*()*;  
  
 // Ensure all threads have finished  
 for *(*Thread thread : threads*)* try *{* thread.join*()*;  
 *}* catch *(*Exception e*) {  
 }* if *(*print*)* System.*out*.print*(*"\r100%"*)*;  
*}*

BVH:

יצרנו מחלקה עבור BOX ומיקמנו אותה בתוך אינטרפייס intersectable, שתהווה קופסה שתכיל את הגאומטריות

public class Box *{* Point3D \_min;  
 Point3D \_max;  
 Point3D mid;  
  
 */\*\*  
 \* constractor that generate a box with two points  
 \*  
 \** ***@param*** *min point  
 \** ***@param*** *max point  
 \*/*public Box*(*Point3D min, Point3D max*) {* \_max = max;  
 \_min = min;  
 mid = new Point3D*((*max.getX*()* + min.getX*())* / 2d, *(*max.getY*()* + min.getY*())* / 2d, *(*max.getZ*()* + min.getZ*())* / 2d*)*;  
*}*

public Point3D get\_min*() {* return \_min;  
 *}* public Point3D get\_max*() {* return \_max;  
 *}  
  
 /\*\*  
 \** ***@param*** *ray  
 \** ***@return*** *true if the ray intersect withe the box and false if not  
 \*/* public boolean intersectBox*(*Ray ray*) {* if *(*ray.getDirection*()*.get\_head*()*.getX*()*==0*){*return false;*}* double txmin = *(*\_min.getX*()* - ray.getPoint*()*.getX*())* / ray.getDirection*()*.get\_head*()*.getX*()*;  
 double txmax = *(*\_max.getX*()* - ray.getPoint*()*.getX*())* / ray.getDirection*()*.get\_head*()*.getX*()*;  
 if *(*txmin > txmax*) {* double temp = txmax;  
 txmax = txmin;  
 txmin = temp;  
 *}* if *(*ray.getDirection*()*.get\_head*()*.getY*()*==0*){*return false;*}* double tymin = *(*\_min.getY*()* - ray.getPoint*()*.getY*())* / ray.getDirection*()*.get\_head*()*.getY*()*;  
 double tymax = *(*\_max.getY*()* - ray.getPoint*()*.getY*())* / ray.getDirection*()*.get\_head*()*.getY*()*;  
 if *(*tymin > tymax*) {* double temp = tymax;  
 tymax = tymin;  
 tymin = temp;  
 *}* if *(*txmin > tymax || tymin > txmax*)* return false;  
 if *(*tymin > txmin*)* txmin = tymin;  
  
 if *(*tymax < txmax*)* txmax = tymax;  
 if *(*ray.getDirection*()*.get\_head*()*.getZ*()*==0*){*return false;*}* double tzmin = *(*\_min.getZ*()* - ray.getPoint*()*.getZ*())* / ray.getDirection*()*.get\_head*()*.getZ*()*;  
 double tzmax = *(*\_max.getZ*()* - ray.getPoint*()*.getZ*())* / ray.getDirection*()*.get\_head*()*.getZ*()*;  
 if *(*tzmin > tzmax*) {* double temp = tzmax;  
 tzmax = tzmin;  
 tzmin = temp;  
 *}* if *((*txmin > tzmax*)* || *(*tzmin > txmax*))* return false;  
 return true;  
 *}  
  
  
  
  
}*

המחלקה מכילה שדה מסוג POINT3D עבור נקודת המקסימום בקופסא

המחלקה מכילה שדה מסוג POINT3D עבור נקודת המינימום בקופסא

המחלקה מכילה שדה מסוג POINT3D עבור נקודת המרכז בקופסא

בנוסף קיימת פונקציה בוליאנית שבודקת האם קרן נתונה חותכת את הקופסא.

כעט, לאחר שיצרנו מחלקת BOX אנו נשתמש בה:

עבור כל צורה אנו יוצרים BOX בבנאי שלה, ע"י פונקציה createBox המקיפה צורה זו בקופסא המקבילה לצירים:

עבור ספירה:

private void createBox*(){* double xmin, xmax, ymin, ymax, zmin, zmax;  
 xmin = this.\_center.getX*()* - this.\_radius;  
 xmax = this.\_center.getX*()* + this.\_radius;  
 ymin = this.\_center.getY*()* - this.\_radius;  
 ymax = this.\_center.getY*()* + this.\_radius;  
 zmin = this.\_center.getZ*()* - this.\_radius;  
 zmax = this.\_center.getZ*()* + this.\_radius;  
 this.\_box = new Box*(*new Point3D*(*xmin, ymin, zmin*)*,new Point3D*(*xmax, ymax ,zmax*))*;  
*}*

מכיוון שמדובר בכדור עם מרכז נתון ורדיוס נתון, עבור כל קודקוד בBOX נוסיף או נחסיר את הרדיוס בהתאמה ממרכז הכדור.

עבור PLAN:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

מכיוון שהמשטח הוא אינסופי הגבולות של הקופסא גם כן באינסוף.

עבור פוליגון:

תמונה שמכילה טקסט, צילום מסך, מסך, סגור

התיאור נוצר באופן אוטומטי

נאתחל את הערך המקסימלי באינסוף שלילי וכן ערך מינימלי לאינסוף חיובי,

לאחר מכן נעבור על כל קודקוד בפוליגון ונשנה את המינימום לנקודת המינימום ואת המקסימום לנקודת המקסימום. כך שהפוליגון יהיה מוקף בקופסא.

כיוון שמשולש יורש מפוליגון נבצעה בניית קופסא של משולש בעזרת הפונקציה הנ"ל.

לאחר מכן ניצור קופסא המכילה את כל הצורות : במחלקת GEOMETRIES

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט, מסך, צילום מסך, כסף

התיאור נוצר באופן אוטומטי

בכל פעם שנשלח ע"י פונקציית ADD עוד צורות או צורה, הקופסה שמכילה את כל הצורות תתעדכן להכיל את כלל הצורות בסצנה.

על מנת לשמור על עיקרון של builder design יצרנו במחלקת הBaceRayTrace פונקציה זו:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

אשר מהווה הפעלה או ניתוק לשיפור הBVH.

כאשר ברירת המחדל עבור ערך הBVH הוא FALSE

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

כעט נראה את המימוש של ה"מתג" הזה בפונקציות:

בפונקציית findClosestIntersection שקוראת לפונקציות מציאת החיתוך עם הגאומטריות findGeoIntersection הוספנו פרמטר BVH בוליאני, שערכו הדיפולטיבי כאמור הינו F.

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

לכן נצטרך כעט לעדכן את כל פונקציות findGeoIntersection עבור כל הגופים:

עבור geometries:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

אנו רואים כי אם הBVH הוא TRUE אז חישוב הקרן האם חוצה את הקופסא פועל ואם לא אז לא.

בנוסף פונקציה זו שולחת לפונקציות ייעודיות עבור כל צורה ולכן את אותו ערך בוליאני BVH נשלח ג"כ אל כל פונקציה ייעודית:

ספירה:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

מישור:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

פוליגון:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

משולש:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי