ד"וח פרויקט

* כל הפונקציות מחזירות פרמטרים חדשים תמיד – כולל הפונקציות של חיבור/חיסור, או הכפלה וקטורית (למעט נירמול שמתבצע על האיבר עצמו).
* יצירת setters רק כשהיה חייב (כמובן שלכל משתנה (חוץ מכשלא רצינו אופציה לראות את המשתנה) יצרנו getter)
* השתדלנו להשתמש בשמות משתנים כמו במבנה המצגת של ד"ר אלישי.
* Java doc המסביר כל מחלקה וכל פונ' בצורה פורמאלית.
* שיפורים:
* Depth of field
* Threads
* Full implemented Bounding Volume Hierarchy
* מה שדן זילברשטיין כתב, יש עליו קרדיט לו

הסבר על שיפורי התמונה

בעיית depth of field:

הבעיה הייתה שעצמים מרוחקים נראו באותה חדות כמו עצמים מקורבים, וגם לא היה אפשרות להתפקס על עצם מסוים. (הכל בסצנה היה בפוקוס)

התיקון:

במציאות, כשאנחנו רוצים לראות אובייקטים שנמצאים במרחק בצורה יותר ברורה, אנחנו מצמצמים את האישונים. כשהאישונים אינם מצומצמים, אובייקטים מרוחקים יותר, יהיו מטושטשים יותר. על מנת להשיג עומק שדה, הגדרנו את פעולת הצמצום וההרחבה של האישונים על המצלמה. הגדרנו משטח חדש בשם משטח מיקוד (focal plane) שמוצב מול משטח הצפייה. הגדרנו אותו במחלקת מצלמה, והוא מוגדר ע"י המרחק שלו ממשטח הצפייה. הנקודה על משטח המיקוד המתקבלת על ידי הקרן שנשלחת מהנקודה על משטח הצפייה נקראת נקודת מיקוד. הגדרנו את צמצם המצלמה - ריבוע על משטח הצפייה שמתאר את "כמות" החלון שדרכו עובר האור. הגדרנו את הצמצם במחלקת מצלמה, והוא מוגדר ע"י אורך הריבוע על המשטח.

מהצמצם, שלחנו קרניים (כמספר הקרניים שקיבלנו בטסט (שמור ברנדר)) מפוזרות באופן אחיד (רנדומלית) דרך הfocal point. אם כל הקרניים ששלחנו מהצמצם דרך נקודת המיקוד מחזירות אותו צבע, סימן שהאובייקט נמצא בפוקוס ואת הצבע שחזר נשים בפיקסל. אחרת, נערבב את הצבעים שחזרו (מה שיוצר את האשליה שהתמונה לא בפוקוס) ואת צבע זה נשים בפיקסל.

נתקלנו בבעיה שלפעמים יש צורות בין המצלמה למשטח הצפייה, וככה המצלמה לא תזהה אותם. לכן אומנם שלחנו קרניים כביכול מהצמצם, אך בפועל כל קרן ששלחנו מהצמצם, חישבנו מה נקודת החיתוך שלה עם המישור שמקביל למשטח הצפייה שהמצלמה נמצאת בו, ושלחנו את הקרן משם, כדי לא לפספס את מה שבאמצע.

הוספנו גם בrender משתנה שמפעיל/מכבה את הפיצ'ר. (אם הוא כבוי, נחשב את הצבע כמו לפני השיפור).

הסבר איפה כל דבר שמור ולמה:

הפרמטרים שיהיו לנו הם:

גודל הצמצם – aperture\_ שהוא אורך כל צלע, בריבוע שמגדיר את הצמצם – הריבוע של view plane's source area (מסוג double)

המרחק של הfocus plane - \_focusDistance שהוא המרחק של הfocus plane מהview plane (מסוג double)

בנוסף נוסיף עוד שני פרמטרים לrender בשביל הפיצ'ר:

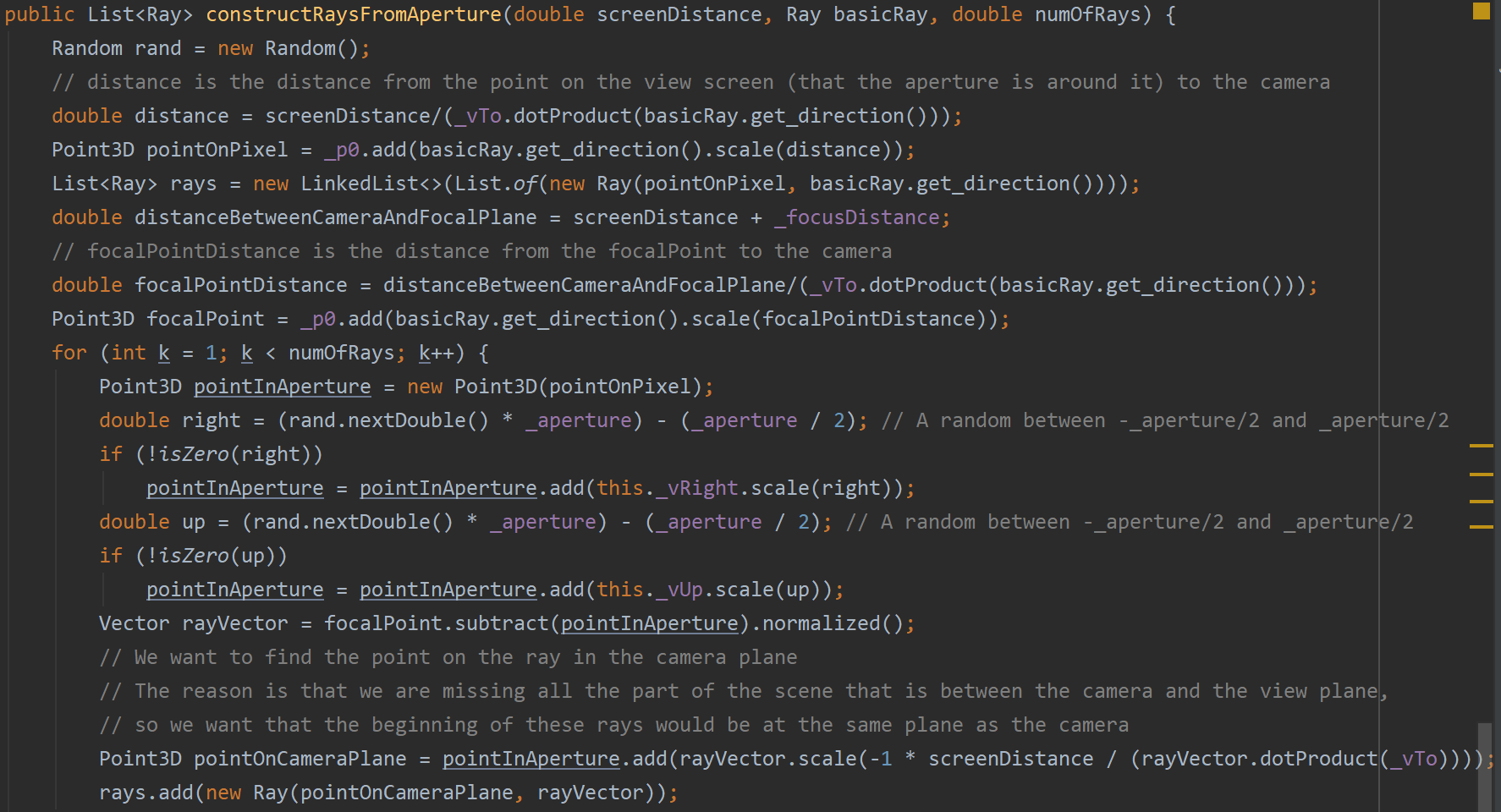
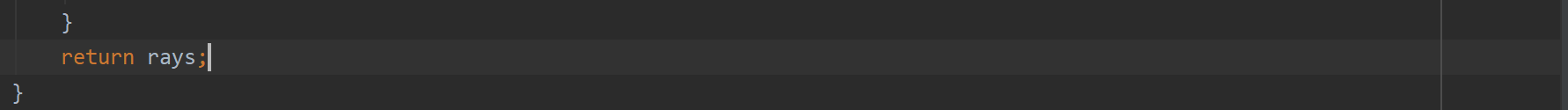
מספר הקרניים שצריך להוציא מה view plane's source areaל focus planeכשמחשבים צבע של פיקסל – numOfRays\_ שהוא מספר הקרניים שנוציא כשנחשב את הצבע, כולל קרן הבסיס. המספר חייב להיות גדול או שווה ל50, כדי שהתמונה תהיה יחסית מדויקת. (מסוג int)

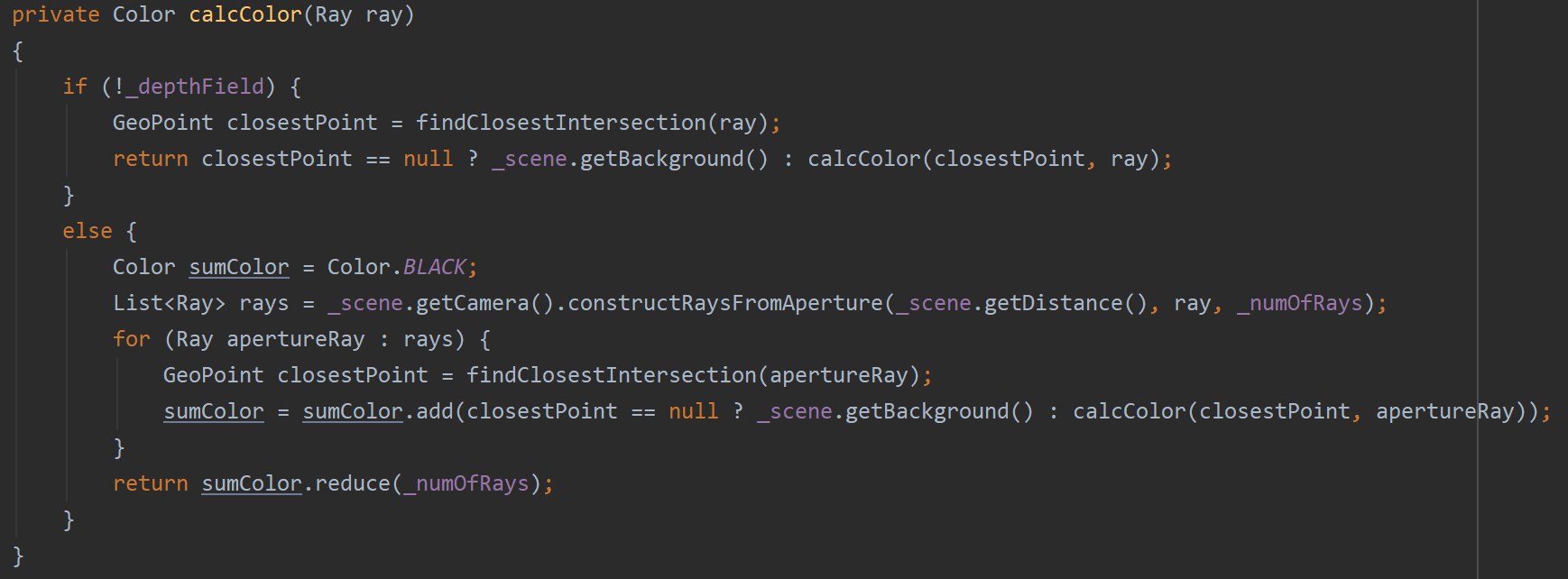
האם להשתמש בפיצ'ר או לא – depthField\_ שהוא האם הפיצ'ר דולק או לא: true אם כן, false אם לא (מסוג boolean)

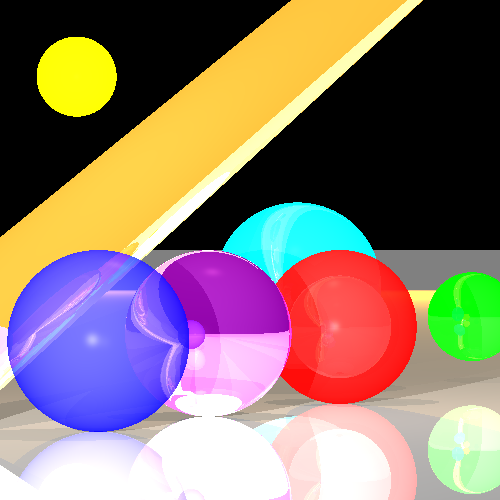
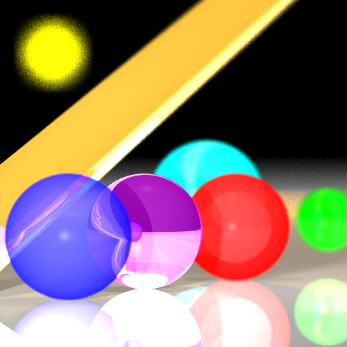
גודל הצמצם והמרחק מהfocus plane ישמרו במצלמה, בגלל שאלו פרמטרים שמוגדרים בעולם האמיתי בתוך המצלמה, והמצלמה עצמה היא מי שאחראית עליהם, והם גם מגדירים מה סוג המצלמה וכו.

את כמות הקרניים ואת האם הפיצ'ר עובד או לא, נגדיר בrender מכיוון שזה דברים המשפיעים על צורת התמונה הסופית (איכות, שינוי צבע וכו') ולכן הם קשורים יותר לרינדור, להגדרות שמגדירים בזמן הצילום, ולא מגדירים את המצלמה או קשורים אליה, אלא רק את אופן עשיית התמונה.

ככה הנראית הפונקציה לחישוב הקרניים מהצמצם: (אותה ביצענו בcamera כי היא אחראית ליצירת קרניים)



וככה נראית הפונקציה לחישוב הצבע: (שנמצאת בrender. בimage writer, שלחנו ישירות אליה במקום של כתיבת הצבע)

תמונה לפני השיפור: תמונה אחרי השיפור:

זמן ריצה: 10 שניות זמן ריצה: 5 דקות ו44 שניות

בעיית זמני הריצה:

כפי שניתן לראות, בגלל שעבור כל פיקסל, שולחים יותר קרניים (לפחות 50) זמן הרצה עולה, ונהיה מאוד ארוך, אפילו עבור תמונה פשוטה בלי הרבה צורות.

שיפרנו את זמן הריצה ב2 דרכים:

1. Threads
2. BVH - Bounding Volume Hierarchy

Threads

לגרום למחשב להריץ במקביל, בכמה תהליכים שונים, ככה שהוא יחשב את הצבעים במקביל.

הקוד נלקח מדן זילברשטיין.

אחרי ההוספה של זה, זמן הרצה השתפר מ5 דקות ו44 שניות ל3 דקות ו30 שניות.

BVH - Bounding Volume Hierarchy

כל אובייקט תחום בקופסא. בנוסף, נתחום את כל הסצנה בקופסא אחת (בgeometries הגדרנו קופסה (כהקופסא הכי קטנה שמכילה את כל הקופסאות – כל האובייקטים)), ואז נחלק את הסצנה ל2 (לפי החלוקה המוצלחת ביותר). כל חלק נתחום בקופסא (כל חלק נגדיר כgeometries), ונחלק ל2, וכו'... ככל שיש יותר "נפח" (יותר אובייקטים) בקופסא מסוימת, אז יהיו יותר קופסאות בפנים.

כשנחפש נקודות חיתוך נבדוק האם הקרן פוגעת בקופסא החיצונית (וזה זמן ריצה מהיר) ואז נתקדם לקופסאות פנימיות יותר, עד שנגיע לקופסא הפנימית ביותר. אם הקרן פוגעת בקופסא הפנימית ביותר אז עם הפרימיטיביות בתוך הקופסא הזו נמצא נקודות חיתוך.

הדרך שבה חילקנו את הgeometries ל2: חילקנו לפי כל אחד משלושת הצירים ל2 (חלוקה באמצע), ובדקנו באיזה חלוקה אנחנו מקבלות נפח קופסאות הכי קטן (כדי שלקרניים יהיה כמה שפחות חיתוך עם הקופסאות).

הפונקציות שבהן עשינו את זה: (רשום בgeometries, בה מוגדרות כל הצורות של הסצנה, ולכן היא אחראית לחלק אותם):

public void separateToGeometries() {  
 if (this.\_finalElements.size() <= 2)  
 return;  
 // We get a separation (cut in the middle) according to x axis, to y axis, and to z axis.  
 // The separation is a list of two "boxes" (2 Intersectable)  
 List<Intersectable> xSeparation = separateToGeometries('x');  
 List<Intersectable> ySeparation = separateToGeometries('y');  
 List<Intersectable> zSeparation = separateToGeometries('z');  
 // finds to each separation, what is the sum of the boxes volume (each separation divides the box, the geometries, into 2 boxes)  
 double xVolume = xSeparation.get(0).getBox().volume() + xSeparation.get(1).getBox().volume();  
 double yVolume = ySeparation.get(0).getBox().volume() + ySeparation.get(1).getBox().volume();  
 double zVolume = zSeparation.get(0).getBox().volume() + zSeparation.get(1).getBox().volume();  
 // sets the separation to be the one that the sum of the boxes volume is the smallest.  
 // the way it sets it is setting the \_finalElements to be the separation (the list of the 2 boxes)  
 if (xVolume < yVolume)  
 if (xVolume < zVolume)  
 this.\_finalElements = xSeparation;  
 else  
 this.\_finalElements = zSeparation;  
 else  
 if (yVolume < zVolume)  
 this.\_finalElements = ySeparation;  
 else  
 this.\_finalElements = zSeparation;  
 // if the first box is Geometries type - has a lot of geometries in it  
 if (\_finalElements.get(0) instanceof Geometries)  
 // set improvementBVH to be true, but as a Geometries (so it will activate separateToGeometries on the box)  
 ((Geometries) \_finalElements.get(0)).set\_improvementBVH(true);  
 else  
 // any way activate improvementBVH to be true  
 \_finalElements.get(0).set\_improvementBVH(true);  
 // if the second box is Geometries type - has a lot of geometries in it  
 if (\_finalElements.get(1) instanceof Geometries)  
 // set improvementBVH to be true, but as a Geometries (so it will activate separateToGeometries on the box)  
 ((Geometries) \_finalElements.get(1)).set\_improvementBVH(true);  
 else  
 // any way activate improvementBVH to be true  
 \_finalElements.get(1).set\_improvementBVH(true);  
}

private List<Intersectable> separateToGeometries(char axis) {  
 if (axis != 'x' && axis != 'y' & axis != 'z')  
 throw new IllegalArgumentException("Axes must be x, y, z");  
 // get the value of the middle point according to the axis (x, y or z axis)  
 double middle = ((axis == 'x' ? getBox().min.get\_x() : axis == 'y'? getBox().min.get\_y() : getBox().min.get\_z()).get() +  
 (axis == 'x'? getBox().max.get\_x() : axis == 'y'? getBox().max.get\_y() : getBox().max.get\_z()).get()) / 2;  
 List<Intersectable> small = new LinkedList<Intersectable>(); // all the geometries that their the middle value of their box is smaller than middle value of the big box (Geometry box)  
 List<Intersectable> big = new LinkedList<Intersectable>(); // all the geometries that their the middle value of their box is bigger than middle value of the big box (Geometry box)  
 for (Intersectable element : this.\_finalElements) {  
 // find the geometry (element) middle value according to axis  
 double elementMid = ((axis == 'x' ? element.getBox().max.get\_x() : axis == 'y' ?  
 element.getBox().max.get\_y() : element.getBox().max.get\_z()).get() +  
 (axis == 'x' ? element.getBox().min.get\_x() : axis == 'y' ?  
 element.getBox().min.get\_y() : element.getBox().min.get\_z()).get()) / 2d;  
 // associate the geometry to the right list  
 if (elementMid < middle)  
 small.add(element);  
 else  
 if (elementMid > middle)  
 big.add(element);  
 else  
 // if elementMid == middle  
 // then chose randomly to which list to associate it to  
 if (Math.*random*() < 0.5)  
 small.add(element);  
 else  
 big.add(element);  
 }  
 if (small.size() == 0 || big.size() == 0) // if none of the geometries went to small or to big, then the random did not divide it good  
 return separateToGeometries(axis); // so do it again  
 if (small.size() == 1) // needs only to make big list to Geometries  
 return List.*of*(small.get(0), new Geometries(big.toArray(new Intersectable[big.size()])));  
 if (big.size() == 1) // needs only to make small list to Geometries  
 return List.*of*(big.get(0), new Geometries(small.toArray(new Intersectable[small.size()])));  
 // make both lists to Geometries  
 return List.*of*(new Geometries(small.toArray(new Intersectable[small.size()])),  
 new Geometries(big.toArray(new Intersectable[big.size()])));  
}

הגדרנו קופסא בקוד ככה: הוספנו מחלקת קופסא: BVHBox, שמייצגת קופסא שמקבילה לצירים. הגדרנו אותה על ידי שני קודקודים: קודקוד אחד (min) שבו יש את הערך הקטן של כל ציר, וקודקוד אחד (max) שבו יש את הערך הגדול של כל ציר. והגדרנו constructor שיקבל את 2 הנקודות, min וmax. הגדרנו להם גם getters. אחר כך הגדרנו פונקציה: public boolean anyIntersections(Ray ray), שמקבלת קרן, ובודקת האם הקרן חותכת את הקופסא. את הקוד לקחנו מהאינטרנט: <https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection>.

הוספנו, לצורך החלוקה לקבוצות, גם פונקציה public double volume() שמחזירה את נפח הקופסא (גובה כפול רוחב כפול אורך).

אחרי שהגדרנו מה זה אומר קופסא (מחלקת קופסא), התלבטנו איפה להגדיר שלכל צורה יהיה קופסא. לבסוף החלטנו ככה: הוספנו לintersectable, שבו יש את כל ההגדרות לגבי כל צורה, ואת החיתוך עם צורה, משתנה: private BVHBox box; שהוא הקופסא שיש על אותה צורה (כדי להוסיף משתנה, שינינו את intersectable ממשק למחלקה אבסטרקטית). הגדרנו גם לintersectable בנאי, שבו הוא יגדיר את הקופסא להיות null. אנחנו רוצים לוודא שהמחשב יחשב פעם אחת לכל צורה את הקופסא (כדי לחסוך בזמן) (בגלל זה גם הגדרנו את הקופסא כמשתנה). לכן הגדרנו 2 פונקציות:

1. public final BVHBox getBox() שהיא דואגת למופע אחד של הקופסא בצורה הבאה:

public final BVHBox getBox() {  
 if (box == null)  
 box = calcBox();  
 return box;  
}

1. ופונקציה protected abstract BVHBox calcBox(); שאותה כל צורה מממשת לעצמה, שבה מוגדר החישוב של הקופסא לאותה צורה.

אופן חישוב הקופסא לכל צורה:

עבור Poligon (ובפרט משולש) חישבנו ככה: מצאנו מה ערך הx הכי קטן שקיים בקודקודים, מה ערך הy הכי קטן שקיים בקודקודים, ומה ערך הz הכי קטן שקיים בקודקודים, ובחרנו את הנקודה שמורכבת משלושתם להיות הנקודת מינימום של הקופסא. ככה עשינו גם עבור הנקודת מקסימום של הקופסא (רק עם ערך מקסימלי של x,y וz במקום מינימלי).

עבור כדור: לקחנו את נקודת המרכז, עבור נקודת המינימום: הגדרנו את x להיות הx של נקודת המרכז פחות הרדיוס, את y להיות הy של נקודת המרכז פחות הרדיוס ואת z להיות הz של נקודת המרכז פחות הרדיוס. עבור נקודת המקסימום עשינו אותו הדבר, רק במקום להוריד את הרדיוס, הוספנו אותו.

עבור plane וtube: שניהם אין סופיים, ועבור צורות אין סופיות, לא מוגדרת קופסא, לכן עשינו שיחזירו null. (נרחיב עוד בהמשך על צורות אין סופיות).

עבור Geometries: הגדרנו את הקופסא להיות קופסא שמקיפה את כל הצורות (כל הקופסאות), באופן הבא: מצאנו מה ערך הx הכי קטן שקיים במינימומים של כל הקופסאות, מה ערך הy הכי קטן שקיים במינימומים של כל הקופסאות, ומה ערך הz הכי קטן שקיים במינימומים של כל הקופסאות, ובחרנו את הנקודה שמורכבת משלושתם להיות הנקודת מינימום של הקופסא. ככה עשינו גם עבור הנקודת מקסימום של הקופסא (רק עם ערך מקסימלי של x,y וz במקום מינימלי).

בעיית הצורות האין סופיות: את הצורות האינסופיות לא ניתן להכניס לקופסא ולכן נאלץ עבור כל קרן לבדוק האם יש לה חיתוך עם הצורה כמו לפני השיפור. הנחנו שזה לא יהיה מאוד משמעותי כי בד"כ אין הרבה צורות אינסופיות בסצנה ולכן השיפור ימשיך להיות משמעותי.

מה שעשינו עם הצורות האין סופיות, הוא להגדיר בgeometries שתי רשימות במקום elements. רשימה אחת של הצורות הסופיות: finalElements ורשימה אחת של האין סופיות,infiniteElements . כל פעם כשקיבלנו את הצורות, חילקנו אותם לרשימות, כל צורה לרשימה המתאימה. החלוקה נעשתה כך, אם הקופסא של הצורה היא null (element.getBox() == null), הצורה אין סופית. אם לא, הצורה סופית.

כאשר נחשב חיתוך עם Geometries, נבדוק בנפרד (גם אם הקרן לא פוגעת בקופסא של Geometries) גם אם הקרן פוגעת בצורות האין סופיות (ואותם גם לא נחלק לתת קופסאות), כי הצורות האין סופיות, לא חלק מהקופסא הגדולה.

רצינו לוודא שכאשר מחשבים חיתוך עם צורה, בודקים קודם כל את החיתוך עם הקופסא. לכן שינינו את findIntersections שבIntersectable לפעול ככה:

public List<GeoPoint> findIntersections(Ray ray) {  
 if (getBox().anyIntersections(ray) || getBox() == null)  
 return findIntersectionsTemp(ray);  
 return null;  
}

כלומר גרמנו לה להיות פונקציית מעטפת, ועשינו refactor והגדרנו את findIntersectionsTemp(Ray) להיות הפונקציה שבה בודקים את החיתוך עם הצורה הגיאומטרית.

והגדרנו שרק במידה ואין קופסא סביב הגאומטריה, או שיש חיתוך עם הקופסא, הוא יחשב חיתוך עם הצורה.

חיתוך קרן עם Geometries:

בGeometries, אנחנו רוצים שהוא יחשב חיתוך עם הצורות האין סופיות, גם אם אין חיתוך עם הקופסא הגדולה. לכן דרסנו את findIntersections עבור Geometries, ועשינו ככה: אם קיימות לך צורות אין סופיות (!\_infinityElements.isEmpty()), תחשב חיתוך אם כל אחת מהצורות האין סופיות, ותשים את כולם ברשימה. ואז (גם אם לא קיימות), במידה וקיימת לקרן חיתוך עם הקופסא, תקרא ל findIntersectionsTemp(Ray), שבה הוא יחשב את החיתוך של הקרן עם כל הצורות הסופיות (יחשב אם כל אחת (ע"י findIntersections(Ray ray), ואז על כל צורה זה יבדוק אם זה חותך את הקופסא וכו') ויחזיר איחוד של כל החיתוכים). תעשה איחוד של הרשימות (אם אין צורות אין סופיות, או אין חיתוך איתם יאחר עם רשימה ריקה), ותחזיר את הרשימה המאוחדת. במידה ואין חיתוך עם הקופסא הגדולה, תחזיר רק את החיתוך אם האין סופיות (במידה ואין חיתוך, או אין אין סופיות, מחזיר null). ככה:

// find intersection with the infinite elements (infinite elements don't have a box)  
 for (Intersectable element : \_infinityElements) {  
 List<GeoPoint> temp = element.findIntersections(ray); // list of intersection with the infinite element  
 if (temp != null) // if there are intersections  
 intersections.addAll(temp); // add them to all the intersections with Geometries  
 }  
 if (getBox().anyIntersections(ray)) {  
 List<GeoPoint> finalIntersections = findIntersectionsTemp(ray); // list of intersections with final elements  
 if (finalIntersections != null) // if there are intersections  
 intersections.addAll(finalIntersections); // add them to all the intersections with the infinite elements  
 }  
 if (intersections.isEmpty()) // if there are intersections  
 return null;  
 return intersections;

והפונקציה findIntersectionsTemp:

List<GeoPoint> intersections = new LinkedList<GeoPoint>();  
for (Intersectable element : \_finalElements) {  
 // temp is list of intersection with the finite element  
 List<GeoPoint> temp = element.findIntersections(ray); // in separateToGeometries we had set for all the elements the \_improvementBVH to be true,  
 // so it will calculate first (at Intersectable class) if the ray intersects with the box.  
 if (temp != null) // if there are intersections  
 intersections.addAll(temp); // add them to all the intersections with Geometries  
}  
if (intersections.isEmpty()) // if there are intersections  
 return null;  
return intersections;

הוספת אופציית כיבוי והפעלה לשיפור:

לצורך כך, הוספנו בIntersectable משתנה protected boolean \_improvementBVH;, (הוספנו שם, כי שם מוגדר כל החיתוכים עם הקופסאות, ולכן שם נכון להגדיר משתנה שאומר לנו כיצד לחשב את החיתוך), שמגדיר האם יש להשתמש בשיפור, או לא. וככה, בפונקציה findIntersections, במידה והשיפור לא מופעל, נחשב את החיתוך עם הצורה בכל מקרה (גם אם אין חיתוך עם הקופסא). ככה שזה נראה כך:

public List<GeoPoint> findIntersections(Ray ray) {  
 if (!\_improvementBVH || getBox().anyIntersections(ray) || getBox() == null)  
 return findIntersectionsTemp(ray);  
 return null;  
}

בנוסף, בבנאי הגדרנו כברירת מחדל את השיפור כבוי, אך הגדרנו למשתנה set וget ככה שניתן לשנות.

עבור geometries, דרסנו הרי את findIntersections, וצריך להגדיר איך החיתוך מתבצע במידה והשיפור לא מופעל. לכן, הגדרנו רשימה חדשה elements, בה יהיו את כל ההאלמנטים, לא מחולקים לקופסאות, כדי שבמידה והשיפור כבוי, נוכל להשתמש בה (כי יש מצב שfinalElements כבר יהיה מחולק לקופסאות). וכל פעם שמוסיפים צורות לgeometry, הגדרנו שגם נוסיף את הצורות לelements מלבד החלוקה לרשימות לפי צורה סופית או לא.

בfindIntersections הגדרנו שבמידה והפיצ'ר כבוי, תחשב חיתוך עם כל אחד מהצורות בelements (ולכל אחד מהם תגדיר שהפיצ'ר כבוי, כדי שלא יחשב חיתוך עם הקופסא) ותוסיף לרשימה.

ככה שזה יראה ככה:

@Override  
public List<GeoPoint> findIntersections(Ray ray) {  
 List<GeoPoint> intersections = new LinkedList<GeoPoint>();  
 if (!\_improvementBVH) {  
 for (Intersectable element : \_elements) { // check for intersections with all the elements (not in boxes)  
 element.set\_improvementBVH(false); // set for each geometry the \_improvementBVH to be false  
 List<GeoPoint> temp = element.findIntersections(ray); // list of intersection with the element  
 if (temp != null) // if there are intersections  
 intersections.addAll(temp); // add them to all the intersections with Geometries  
 }  
 }  
 else {  
 // find intersection with the infinite elements (infinite elements don't have a box)  
 for (Intersectable element : \_infinityElements) {  
 List<GeoPoint> temp = element.findIntersections(ray); // list of intersection with the infinite element  
 if (temp != null) // if there are intersections  
 intersections.addAll(temp); // add them to all the intersections with Geometries  
 }  
 if (getBox().anyIntersections(ray)) {  
 List<GeoPoint> finalIntersections = findIntersectionsTemp(ray); // list of intersections with final elements  
 if (finalIntersections != null) // if there are intersections  
 intersections.addAll(finalIntersections); // add them to all the intersections with the infinite elements  
 }  
 }  
 if (intersections.isEmpty()) // if there are intersections  
 return null;  
 return intersections;  
}

חשבנו שהכי נכון הוא להפעיל את השיפור ב render (וככה יהיה אפשר לכבות ולהדליק גם אחרי שהגדרנו הכל), ולכן הגדרנו שם פונקציה: set\_improvementBVH, והיא זאת שתפעיל את השיפור. (הוספנו גם פונקציה isImprovementBVHOn, לראות אם השיפור מופעל. הפונקציה הזאת תחזיר האם בgeometries מוגדר שהשיפור מופעל או לא). (בבנאי של render הגדרנו כברירת מחדל את השיפור מכובה).  
מה ש set\_improvementBVHתעשה יהיה להגדיר בסצנה שהשיפור מופעל/מכובה, וככה מהסצינה זה יגיע לצורות. לצורך כך הוספנו לסצינה פונקציה set\_improvementBVH, שהיא מגדירה ב \_geometries את השיפור מופעל/מכובה. אנחנו רוצים שבGeometries כאשר השיפור מופעל, הוא יחלק הכל לקופסאות. לכן נדרוס בGeometries את הפונקציה set\_improvementBVH של Intersectable, ונגדיר לו שכשהוא משנה לאמת, תפעיל את הפונקציה של חילוק לקופסאות:

@Override  
public void set\_improvementBVH(boolean \_improvementBVH) {  
 this.\_improvementBVH = \_improvementBVH;  
 if (\_improvementBVH)  
 separateToGeometries();  
}

התמונה:

בתמונה יש הרבה טבעות, כל טבעת מורכבת מ3600 כדורים.

קנקן: גוף הקנקן הוא כדור שחור ברדיוס 1.651 הממוקם בראשית הצירים, ושני טבעות בלמעלה של הכדור.

המכסה של הקומקום מורכב מ6 טבעות (בגודל הולך וקטן אחד מעל השני) ומ2 כדורים.

הידית שלו מורכבת מטבעת על הצד.

הזרבובית שלו מורכבת מ6 כדורים, ו4 טבעות.

והבסיס שלו מורכב מ4 טבעות. (בגודל הולך וגדל אחד מתחת לשני)

שולחן:

המשטח של השולחן מורכב משני מעגלים שמחוברים ביניהם בפס של ריבועים. כל מעגל מורכב מ3599 משולשים בעלי קודקוד משותף, שמחוברים אחד לשני במעגל. הפס העגול של הריבועים מורכב גם מ3599 ריבועים במעגל.

הרגל של השולחן עשויה משני חלקים. החלק העליון עשוי מ 9 כדורים שחורים כאשר הגודל שלהם הולך ויורד (הגדול ביותר למעלה) והם אחד בתוך השני כך שמתקבלת צורה שנראית אחידה. החלק התחתון עשוי מטבעות, ישנן 5 טבעות אחת על גבי השניה כאשר כל טבעת עשויה מ 3600 כדורים

שיחים: בשיחים יש לנו 2 שורות, השורה התחתונה מורכבת מלולאה שמייצרת 100 כדורים באותו הגודל בצבעים שונים (יש לנו מערך עם 4 אפשרויות שונות לצבעים בכל איטרציה של הלולאה הכדור מקבל את הצבע של המקום ה i mod 4 במערך של הצבעים. השורה העליונה זו לולאה שבה הגודל של הכדורים משתנה ותלוי ב i, שם יש גם 100 כדורים. הצבע של כל אחד מהכדורים נבחר באותה צורה כמו בשורה הראשונה. (אני פשוט לא זוכרת את המספרים) (אני אבדוק עוד מעט)

עננים: יש שני עננים, כל אחד מורכב מ4 כדורים לבנים עם רדיוס 250

דק: הרצפה מורכבת מרצף של שני מלבנים, מלבן אחד חום ברוחב 1.8, ומלבן אחד שחור ברוחב 0.2. ויש שם 18 זוגות של מלבנים כאלו מחוברים.

אורות: יש 4 אורות:

ספוט לייט אחד לבן קצת מעל ומשמאל לקנקן, שמאיר על הקנקן והשולחן,

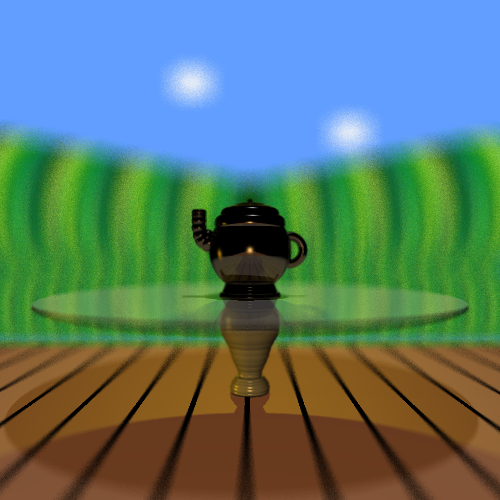
ספוט לייט אחד שמול התמונה שמאיר על השולחן והרצפה בצבע זהב,

ספוט לייט קצת מעל הרצפה, בצד ימין, מאיר לכיוון הרצפה שמאלה, בצבע חום

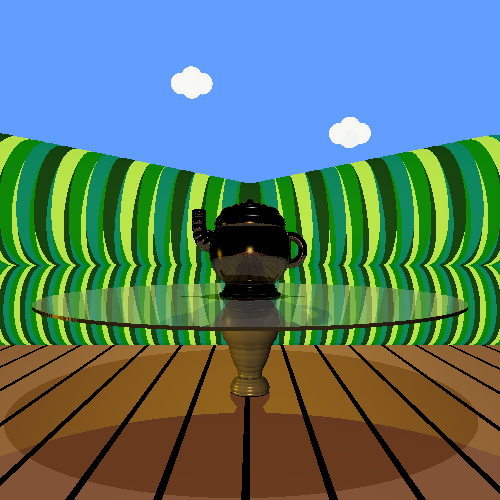
פוינט לייט קצת מעל השיחים ויותר מקדימה.

סה"כ מספר הצורות – 83059

תמונה, עם depth of field:

 זמן ריצה עם BVH: 3 שעות, 37 דקות ושניה (לא הספקנו להריץ את התמונה ללא BVH כי זה אמור לקחת כמה ימים (התחלנו להריץ, ולא היה לנו זמן לסיים. אבל גם כשעצרנו אותו, אחרי הרבה זמן, הוא עדיין סיים רק 0 אחוז מהעבודה שלו.).

ללא depth of field:

 זמן ריצה עם BVH: 3 דקות ו44 שניות. זמן ריצה ללא BVH: שעתיים, 17 דקות ו5 שניות (שיפור של פי 36.5! כלומר הBVH עובד ומשפר בהמון)