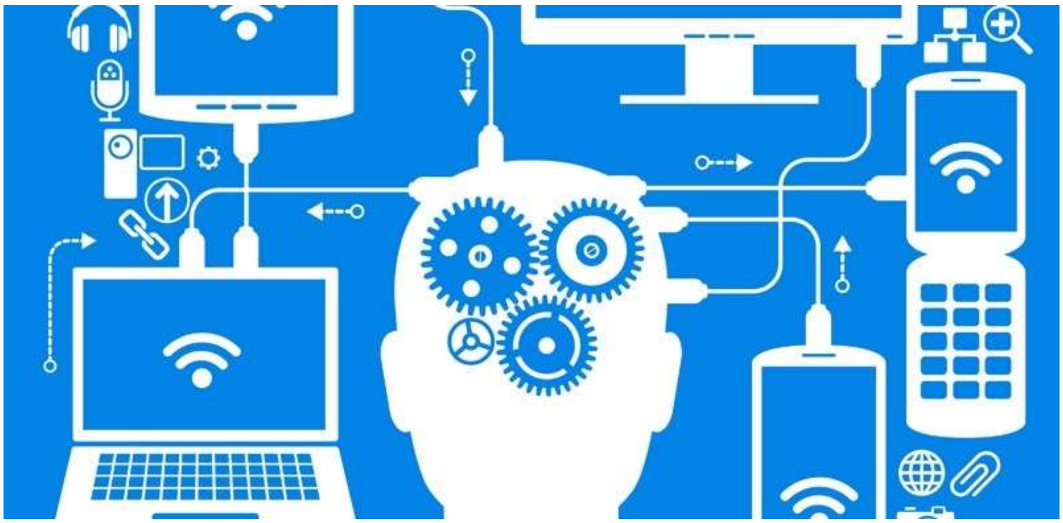
Introduction to Software Engineering

JERUSALEM COLLEGE OF TECHNOLOGY
LEV ACADEMIC CENTER



Recitations

- *Week 0: Home assignment 0 (not graded): Introduction to OOP with Java.*
- *Week 1: Home assignment 1: Implementation of primitives with operations, geometries*
- *Week 2: Home assignment 2: Implementation of geometries primitives' unit testing for primitives (JUnit). Implementation of normal calculation and their unit testing through geometries.*
- **Week 3: Home assignment 3**: Implementation of ray-geometry intersections and their unit testing through geometries.
- *Week 4: Home assignment 4: Implementation of camera class, rays through view plane construction and unit testing of camera*
- *Week 5: Home assignment 5: Implementation of ambient light, scene, render and image-writer classes with their appropriate test units*
- *Week 6-7: Home assignment 6: Adding material support. Implementation of Phong model, with light emission, directional, point and spot lighting, multiple light sources*
- *Week 8: Home assignment 7: Implementation of shadow rays, reflection and refraction*
- *Week 9: Mini-project 1: Implementation of a picture improvement algorithm*
- *Week 10-11: : Mini-project 2: implementation of a performance improvement algorithm*
- *Week 12-13: Mini-projects presentation*

Home assignment 3

Refactoring the geometry package, implementation of Geometries composite class, implementation of ray-geometry intersections' unit testing and implementation of ray-geometry intersections for Sphere, Plane and Triangle.

Bonus1: Ray-Polygon intersections (1pt)

Bonus2: Ray-Tube intersections (2pt: tests + implementation)

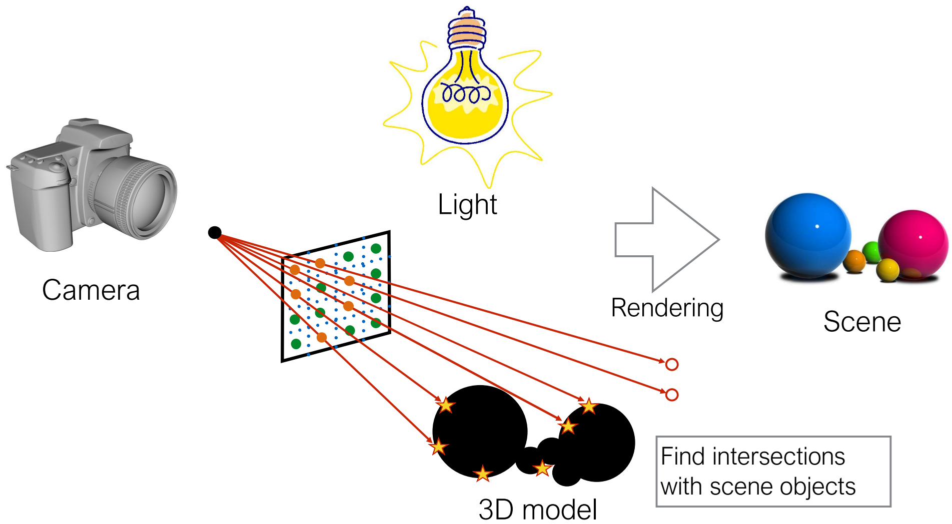
Bonus3: Ray-Cylinder intersections (2pt: tests + implementation)

Bonus4: Triangle Barycentric or Möller–Trumbore algorithm (1pt)

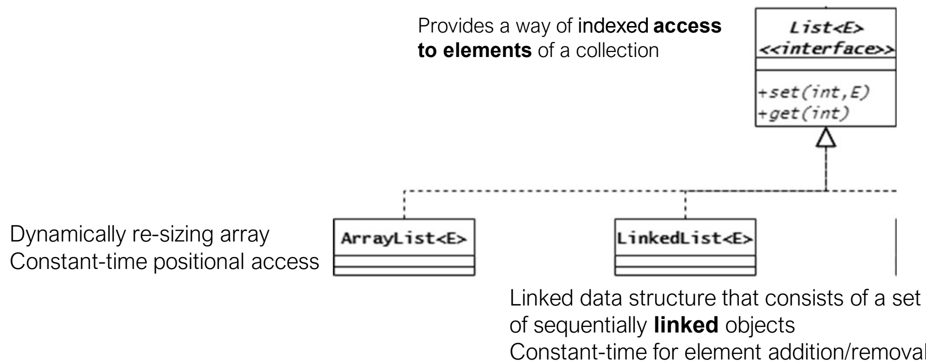
בונוסים – ניתן לבצע גם בשלבים מאוחרים יותר אך עדיין לקבל את הניקוד

בבונוס הרביעי: מדובר בפתרון ש**לא** משתמש בחישוב מקדים של חיתוך המישור של המשולש אלא מבצע חישוב ובדיקה בו זמנית, תוך הסתמכות על מנגנון מתמטי מורכב. המימוש של הפתרון אמור להיות מהיר יותר (מבחינת זמן ריצה) מאשר הפתרונות של "פירמידה" או גישה "שטוחה" של חישובים והשוואת כיוונים של נורמלים.

Ray Casting


Light
Camera
Rendering
Scene
3D model
Find intersections
with scene objects

Collections, Lists and Array Lists of objects



Java 9+: List.of(…) static function provides generation of an immutable direct access iterable list
- **Fast**
- **Simple**

NB: See an example in Polygon constructor

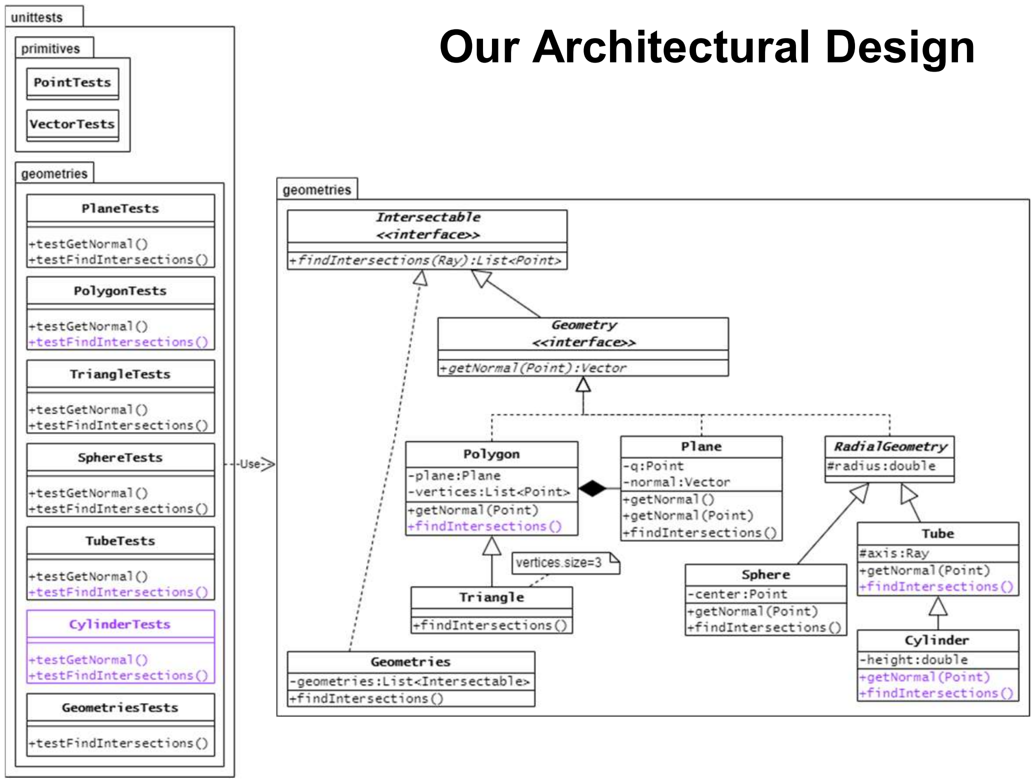
המחלקה ***ArrayList<>*** של **Java** מממשת "וקטור" – מערך דינמי. וכמו כל מערך דינמי, סיבוכיות זמן ריצה מרבית של הוספת אלמנט – הנה ***O(n)***
המחלקה ***LinkedList<>*** של **Java** מממשת רשימה מקושרת חד כיוונית. סיבוכיות זמן ריצה מרבית של הוספת אלמנט – הנה ***O(1)***

ברוב המקרים אנחנו נצטרך לבנות רשימה "קבועה" – מערך שנבנה בתחילה ולא משתנה בהמשך. ברשימה הזו נשתמש בעיקר או ע"י גישה ישירה לאלמנט באינדקס נתון, או ע"י מעבר על כל איבריה מהתחלה ועד הסוף (בעזרת לולאת איטרציה). הדרך היעילה לעשות זאת היא ליצור מערך עבור הרשימה בעזרת מתודה סטטית ***of*** של הממשק ***List***. ראו דוגמה של שימוש במתודה הזו בבנאי של מצולע (מחלקה Polygon) שקיבלתם בשלב הראשון של הפרויקט.

בחלק אחר מהמקרים נבנה רשימה בהדרגה, תוך הוספת אלמנטים. נשתמש ברשימה הזו בדרך כלל ע"י מעבר על כל איבריה מהתחלה ועד הסוף (בעזרת לולאת איטרציה) בלבד. במקרה כזה נעדיף להשתמש ברשימה מקושרת ***LinkedList***.

בכל מקרה, ברשימה כזו או אחרת, בהצהרת טיפוס המשתנה או הפרמטר או השדה או הערך המוחזר – נגדיר אותו בעזרת הממשק ***List<>***.

Our Architectural Design



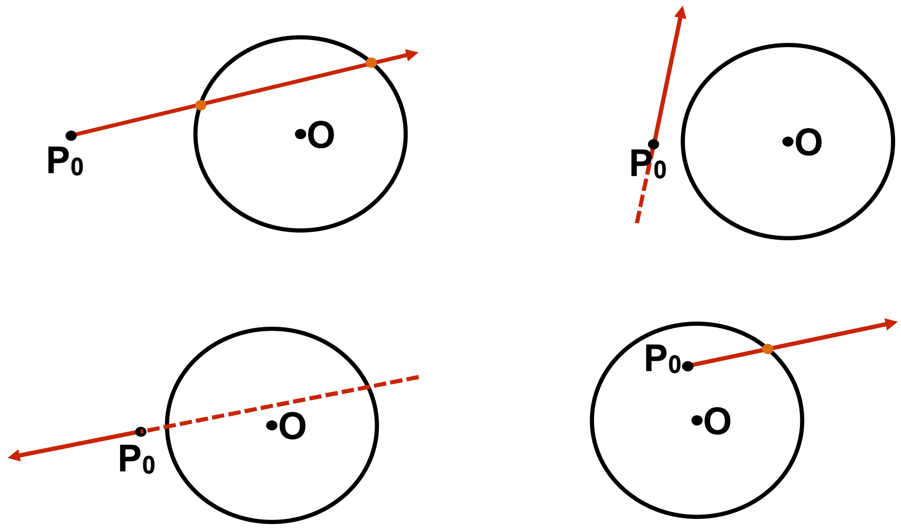
זאת הארכיטקטורה של מה שנעשה בשלב הזה:

1. נוסיף קודם את הממשק החדש ***Intersectable*** ונגדיר בו פונקציה ***findIntersections***
2. היה צריך להוסיף לממשק ***Geometry*** הגדרה של הרחבת הממשק החדש ***Intersectable*** (נוסיף ***interface Geometry extends Intersectable***), אך מכיוון שלכתחילה הגדרנו את ***Geometry*** כמחלקה אבסטרקטית – במקום כן נוסיף שהמחלקה מממשת את הממשק החדש (נוסיף ***abstract class Geometry implements Intersectable***)
3. נוסיף במחלקות Plane, Polygon, Triangle, Sphere, Tube - את המימושים הריקים ({ **return null;** }) של ***findIntersections***
4. נוסיף במחלקות הבדיקות של כל הגופים את מתודת הבדיקה עובר ***findIntersections***, כפי שמופיע בשקף
5. בהמשך המצגת נדבר על הבדיקות ולאחר מכן המימושים של ***findIntersections***

שימו לב שהבדיקות והמימושים של ***findIntersections*** במצולע הכללי, בגליל סופי ואין סופי – כולם לבונוס (אך רק בתנאי שהוספתם גם את הבדיקות הרלוונטיות).
תזכרו להמשך שרשימות של נקודות חיתוך לא כוללות את ראש הקרן ולא כוללות נק׳ השקה, למעט בגליל סופי בנקודות חיבור בין בסיס למעטפת הגליל.

בשקפים הבאים נרענן בקצרה את הבדיקות של חישובי חיתוכים עם גופים מסוימים.

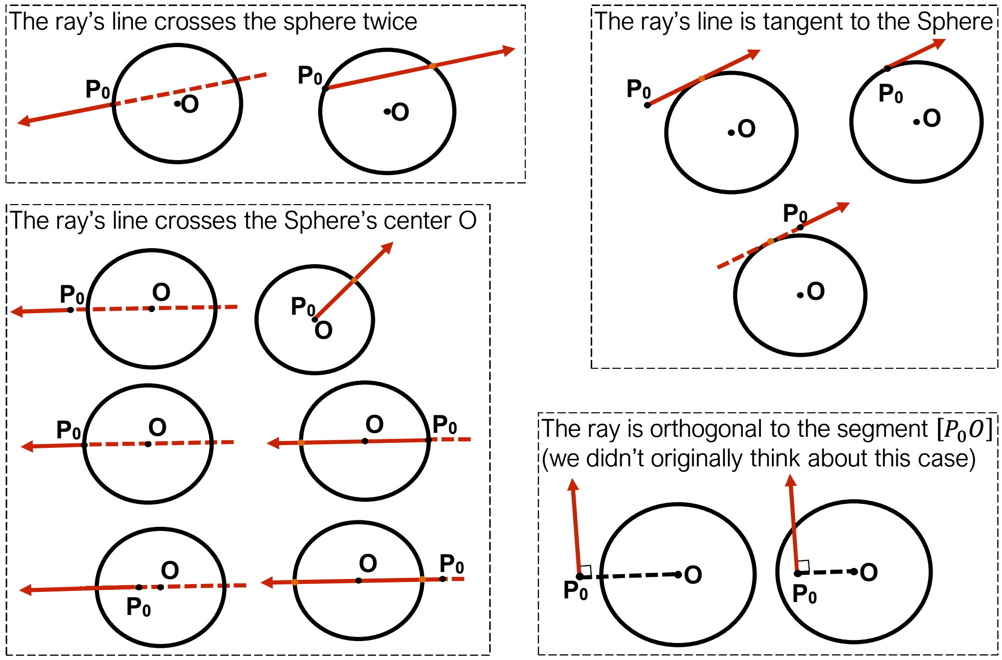
Ray-Spere intersections: EP



נתחיל ממחלקות שקילות בבדיקות חיתוכים בין כדור לקרן. יש לנו 4 מחלקות שקילות:

1. קרן על הישר שכולו מחוץ לכדור
2. קרן על הישר שחותך את הכדור, כאשר ראשית הקרן נמצאת "לפני" הכדור
3. קרן על הישר שחותך את הכדור, כאשר ראשית הקרן נמצאת "בתוך" הכדור
4. קרן על הישר שחותך את הכדור, כאשר ראשית הקרן נמצאת "אחרי" הכדור

Ray-Sphere Intersections: 13 BVA



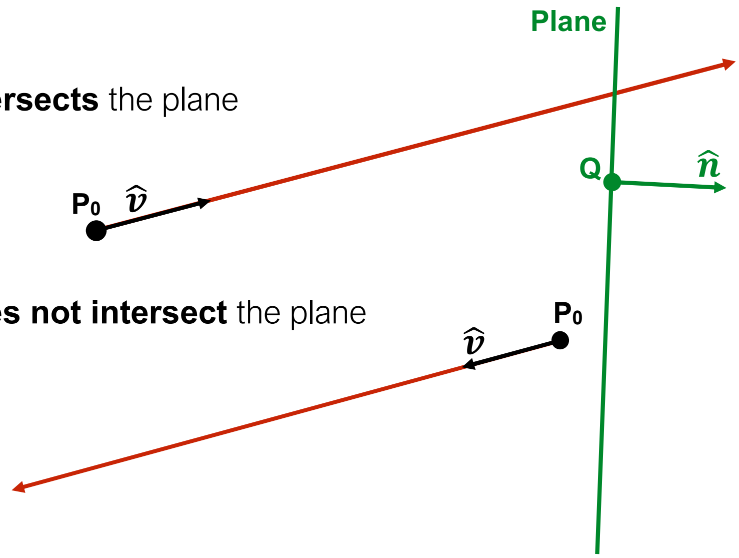
מקרי קצה וגבול בחיתוכים עם כדור מתחלקים ל-4 קבוצות:

1. שני מקרים של קרן על הישר שחותך את הכדור אך לא עובר במרכז הכדור
2. שישה מקרים של קרן על הישר שעובר דרך מרכז הכדור
3. שלושה מקרים של קרן על הישר שמשיק לכדור
4. ושני מקרים נוספים של קרן על הישר כאשר הקטע ממרכז לראשית הקרן מאונך לקרן (זהו מקרי גבול ייחודים שהתגלו במהלך עבודת הסטודנטים על הפרויקט לפני כמה שנים)

Ray-Plane Intersections

EP: The Ray must be **neither orthogonal nor parallel** to the plane

- Ray **intersects** the plane
- Ray **does not intersect** the plane



עבור חיתוכים עם מישור:

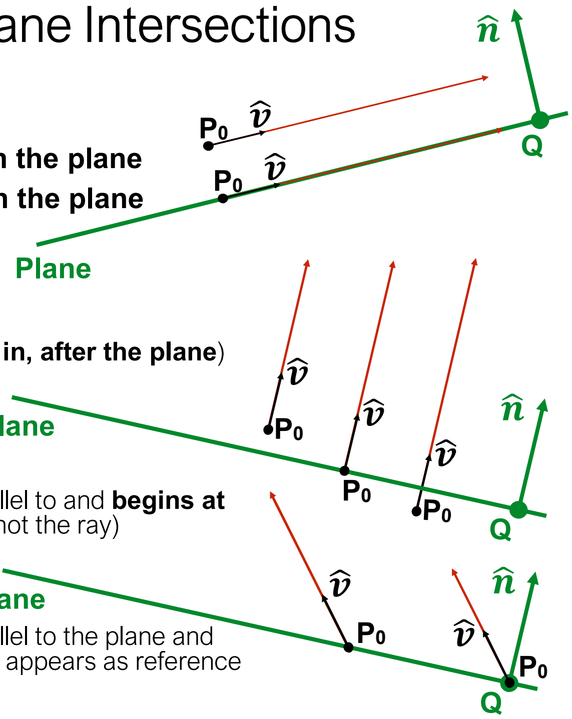
מחלקות שקילות:

1. קרן שמתחילה מחוץ למישור, לא מקבילה למישור, מייצרת זווית לא ישרה עם המישור, וחותכת את המישור
2. קרן שמתחילה מחוץ למישור, לא מקבילה למישור, מייצרת זווית לא ישרה עם המישור, **ולא** חותכת את המישור

Ray-Plane Intersections

BVA: 7 cases

- Ray is **parallel to the plane**
  2 cases: the ray is **included in the plane**
  or **is not included in the plane**
- Ray is **orthogonal to the plane**
  3 cases: according to P_0 (**before, in, after the plane**)
- Ray is neither orthogonal nor parallel to and **begins at the plane** (P_0 is in the plane, but not the ray)
- Ray is neither orthogonal nor parallel to the plane and **begins in the same point** which appears as reference point in the plane (**Q**)



עבור חיתוכים עם מישור:

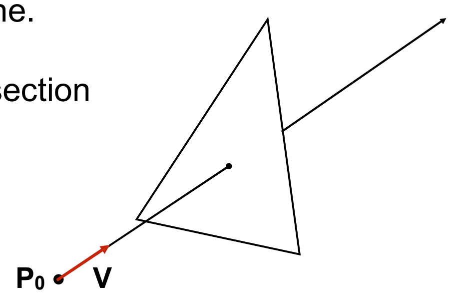
מקרי קצה וגבול:

1. שני מקרים של קרן שמקבילה למישור (מקרה של קרן מחוץ למישור ומקרה של קרן בתוך המישור)
2. שלושה מקרים של קרן המאונכת למישור (מתחילה "לפני", בתוך, ו"אחרי" המישור)
3. מקרה אחד של קרן שלא מקבילה ולא מאונכת למישור אך מתחיל בתוך המישור
4. ומקרה אחד שדומה למקרה הקודם, אך ראשית הקרן בדיוק ב"נקודת הייחוס" של המישור (הנקודה ששמורה באובייקט של המישור בנוסף לווקטור הנורמל, או במילים אחרות - נקודה נתונה בתוך המישור)

Ray-Polygon/Triangle Intersections

First, intersect ray with the plane.

Then, check whether the intersection point is inside triangle or not



הבדיקות של מצולע ושל משולש יתבצעו לפי גישה דומה מאד. שימו לב שחייבים לכתוב את הבדיקות גם עבור מצולע כללי (אם עושים את הבונוס של חיתוכים עם מצולע כללי) **וגם** עבור המשולש.

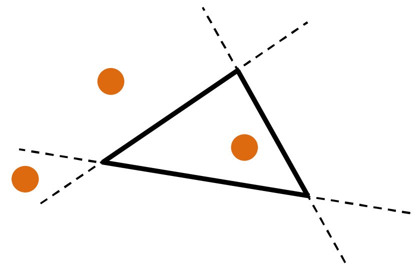
בבדיקות האלה נסתמך שכבר מצאנו את נקודת החיתוך של הקרן עם המישור ה"מוכל" (מבחנת תכנות מונחה עצמים) במצולע או במשולש. מהסיבה הזוה, וגם מכיוון שאנחנו לא כוללים את ראש הקרן ברשימות נקודות החיתוך – לא נבצע בדיקות עם קרן שמתחיל בתוך המישור הזה, אלא רק מחוץ למישור. וגם לא נבדוק מקרים כאשר הקרן לא חותכת את המישור – ניצור רק קרניים שחותכים את המישור ה"מוכל".

הנחת ייסוד נוספת עבור הבונוס של מצולע כללי: הפרויקט שלנו תומך רק במצולעים קמורים. מי שלא זוכר מה זה מצולע קמור – תבדקו בוויקיפדיה.

Ray-Triangle intersections
(First: test intersections with triangle's plane
as in the plane-ray intersection slide)

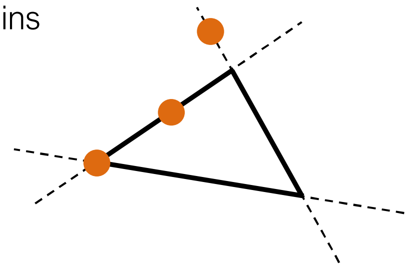
EP: Three cases:

- Inside polygon/triangle
- Outside against edge
- Outside against vertex



BVA: Three cases (the ray begins "before" the plane)

- On edge
- In vertex
- On edge's continuation



עבור חיתוכים עם מצולע או משולש:

יש שלוש מחלקות שקילות:

1. נקודת החיתוך עם המישור ה"מוכלל" נמצאת בתוך המצולע\המשולש
2. נקודת החיתוך עם המישור ה"מוכלל" נמצאת מחוץ המצולע\המשולש – "מול" אחד הצלעות
3. נקודת החיתוך עם המישור ה"מוכלל" נמצאת מחוץ המצולע\המשולש – "מול" אחד הקודקודים. זאת אומרת – בין המשכי שתי הצלעות שמתחילים בקודקוד הזה

מקרי קצה וגבול:

1. נקודת החיתוך עם המישור ה"מוכלל" נמצאת על אחד הצלעות
2. נקודת החיתוך עם המישור ה"מוכלל" נמצאת בתוך אחד הקודקודים
3. נקודת החיתוך עם המישור ה"מוכלל" נמצאת על המשך של אחד הצלעות

Refactoring

Ray points: $\boldsymbol{P} = \boldsymbol{P_0} + \boldsymbol{t} \cdot \boldsymbol{v}$

Why doing it every time? DRY principle!

Where? RDD – Responsibility Driven Design

Ray class should be responsible for it.

```
class Ray {
...
   /** ...
    **/
   public Point getPoint(double t) { ... }
}
```

לאחר מכן נממש את חישובי נקודות החיתוך בכל גוף וגוף על פי מה שנלמד בקורס התיאורטי.

ולבסוף נבצע ארגון הקוד מחדש (***refactoring***). בכל החיתוכים אנחנו מחשבים את המרחק מראשית הקרן לנקודת החיתוך - $\boldsymbol{t}$. ולאחר מכן אנחנו מחשבים את הנקודה לפי הנוסחה בתחילת השקף: $\boldsymbol{P} = \boldsymbol{P_0} + \boldsymbol{t} \cdot \boldsymbol{v}$. במקום לחזור על אותו החישוב של הנוסחה בהרבה מקומות, נפריד את החישוב למתודה נפרדת. לאחר מכן נחשוב – איפה מתודה כזו שייכת. הרי מדובר במציאת נקודה (כלשהי – לאו דווקא חיתוך) על הקרן הנמצאת במרחק נתון מראשית הקרן. ע״פ RDD מדובר בפעולה שישות הקרן צריכה להיות אחראית עליה. לכן נוסיף את המתודה במחלקת Ray, ונשנה את הקוד ב**<u>כל</u>** מימושי החיתוכים – שישתמש במתודה הזו. זה גם יקצר את הקוד וגם יסלק את סירחון הקוד של חזרות לא חיוניות בקוד (needless repetition).

שימו לב שבמימוש של המתודה יש להתייחס למקרה כאשר המרחק $\boldsymbol{t}$ קרוב מאד לאפס (**if (isZero(t)) …**). במקרה הזה פשוט נחזיר את נקודת ראשית הקרן.