

תכנות מונחה עצמים – תרגיל 4

**PEPSE: Precise Environmental
Procedural Simulator Extraordinaire**

תאריך ההגשה: 04.01.26 בשעה 23:55

תאריך הגשת מגן התכנון המוקדם הוא 28.12.2025 בשעה 23:55
(ראו הסבר בפרק 9.3)

תרגיל זה מומלץ להגיש בזוגות

הקדמה 0.

היום אנחנו מכינים סימולציה, או כלי משחק (משהו שמשחקים בו אבל שלא מכיל תנאי ניצחון או הפסד, בשלב זה). הכירו את הסימולטור הפרוצדורלי הנאמן לחלוטין למציאות של... המציאות. הנה היא:



הסימולציה כוללת מחזור יום-לילה, לרבות החשכה של המסך (כמו במציאות!), דמות אוטאר שיכולה לרוץ ולקפוץ, עצים עם עלים הנעים ברוח, ופירות שנותנים אנרגיה לדמות. סימולציה כזו אפשר יהיה לפתח בעתיד לאחד מכמה כיוונים מתבקשים: הוספת אלמנט של מים וגשם, פיתוח משחק פלטפורמר, משחק בניה, או משחק הישרדות. פונקציונלית, הסימולציה מאוד מודולרית. ניתן לראות סרטון דמו [כאן](#).

0.1. מטרות התרגיל

0.1.3. פיתוח הסימולציה הוא תירגול מצוין לתכנות באופי מעט אחר, שבו ריבוי האסטרטגיות הופך את היצירה של עצם למורכבת, עד שלא נוכל לדחוף עוד את הכל למחלקה אחת – נרצה לחלק את מלאכת יצירת העצמים עצמה לקבצים שונים ושיטות שונות. למעשה, הגדרת האסטרטגיות ויצירת העצמים תהווה חלק נכבד מהקוד שלנו, ובפני עצמה לא תיכתב בצורה מונחית עצמים. התרכזנו עד כה בתכנות מונחה עצמים כי זה הנושא שלנו, אבל בהמשך תכירו פרדיגמות תכנות אחרות וכשזה יקרה תיזכרו בנוסטלגיה שאמרנו לכם שאין הכרח שקוד יהיה 100% בפרדיגמה אחת; כל פרדיגמה היא כלי בארגו. התרגיל יהיה הצצה לכך.

0.1.4. עד סוף המדריך, אתם אמורים להרגיש בנח עם נושאים כמו הגדרת למבדות, שימוש ב-method references, ושליחת callbacks.

0.1.5. אולי החשוב מכולם: מהכנת הסימולציה תמשיכו ללמוד להנות מתכנות.

0.2. עיצוב ויצירת שלד הפרויקט

```
src/  
├── assets/  
│   └── avatar images  
└── pepse/  
    ├── PepseGameManager.java  
    ├── pepse.utils/  
    │   ├── ColorSupplier.java  
    │   └── NoiseGenerator.java  
    └── pepse.world/  
        ├── Sky.java  
        ├── Block.java  
        ├── Terrain.java  
        ├── pepse.world.daynight/  
        │   ├── Night.java  
        │   ├── Sun.java  
        │   └── SunHalo.java  
        ├── pepse.world.avatar/  
        │   └── Avatar.java  
        └── pepse.world.trees/  
            └── Flora.java
```

README

על הפרויקט שלכם לכלול את החבילות הבאות:

- pepse: החבילה הראשית עם מנהל המשחק (PepseGameManager).

- `pepse.utils`: מחלקות עזר.
 - כרגע מכילה מחלקה אחת שמייצרת וריאציות של צבעים, ומחלקה נוספת שתעזור לכם לייצר אקראיות למשחק.
 - `pepse.world`: אחראית על יצירת העולם.
 - `pepse.world.daynight`: מכילה פונקציונליות הנוגעת למחזור יום-לילה.
 - `pepse.world.avatar`: מכילה את הקוד הנוגע לדמות.
 - `pepse.world.trees`: מכילה את הקוד הנוגע ליצירת עצים.
- שימו לב:** כדי לאפשר לכם להתאמן גם בהחלטות עיצוב, סיפקנו לכם תרשים חלקי בלבד. כדי לכסות את כל דרישות התרגיל, כנראה שתמצאו את עצמכם מוסיפים חבילות ו/או מחלקות נוספות.
- דגש חשוב:** כל מה שלא מוגדר באופן מפורש בהגדרות התרגיל (מיקום/גודל/התנהגות וכד') נתון להחלטתכם.

0.3. תחילת עבודה

ועכשיו, ניגש למלאכה.

שימו לב! יש לקרוא את כל חלקי התרגיל לפני שמתחילים במימוש.

במיוחד חלקים 6 והלאה, חשוב לתכנן יחד את העיצוב שלהם לפני המימוש.

בתור התחלה, הגדירו פרויקט בשם `Pepse` עם תלות ב-`DanoGameLab`. אפשר להיעזר בפרק 6 של [מדריך ההתקנות](#). לאחר מכן צרו את החבילות והמחלקות לעיל; השאירו את המחלקות ריקות. אנחנו נמלא אותן בהמשך. יוצאת הדופן היא המחלקה `ColorSupplier` שנשתף איתכם. לבסוף, הגדירו את `PepseGameManager` כמחלקת בת של `GameManager`, הוסיפו את ה-`main` הבא:

```
public static void main(String[] args) {
    new PepseGameManager().run();
}
```

ודרוסו באופן ריק את השיטה `initializeGame`. הריצו כדי לוודא שנפתח חלון ריק במסך מלא, ממנו אתם יכולים לצאת עם `Esc`.

נעת נתחיל להגדיר את הדרישות על פי הסדר הבא :

1. [שמיים](#)
2. [קרקע](#)
3. [אור וחושך](#)
4. [שמש](#)
5. [הילת השמש](#)
6. [הוספת דמות](#)
7. [צמחיה](#)
- 7.3. [תנודות העלים ברוח](#)
- 7.4. [הוספת פירות](#)

8. [עולם אינסופי](#) לאחר שנממש את הפונקציונליות הבסיסית, נחזור אחורה ונעדכן את המחלקות ששימשו אותנו ליצירת הקרקע והעצים, כך שיתמכו בעולם אינסופי שנבנה עם התנועה של הדמות.

כעת נתחיל בהגדרת המחלקות:

1. שמיים

להלן חלקי קוד והצעות שיעזרו לכם לממש את הממשק. שימו לב שאנחנו נותנים לכם את הקוד כדי שלא רק תעתיקו ותדביקו אלא שגם תיזכרו איך לעשות את זה בעצמכם.

כדי ליצור מלבן תכול ברקע, נתחיל ונגדיר במחלקה `pepse.world.Sky` שיטה סטטית בשם `create`:

```
public static GameObject create(Vector2 windowDimensions)
```

השיטה מקבלת את גודל החלון, ומחזירה את השמיים שנוצרו.
צבע השמיים בו השתמשנו הוא:

```
private static final Color BASIC_SKY_COLOR = Color.decode("#80C6E5");
```

הגדירו ב-`create` את ה-`GameObject` שמייצג מלבן בצבע השמיים:

```
GameObject sky = new GameObject(  
    Vector2.ZERO, windowDimensions,  
    new RectangleRenderable(BASIC_SKY_COLOR));
```

ליתר ביטחון הגדירו שהמלבן זז עם המצלמה, כך שלא יישאר מאחור אם וכאשר המצלמה תזוז:

```
sky.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
```

במהלך הדיבוגים, מכיוון שהרבה מהעצמים הולכים להיות מופעים ישירים של `GameObject`, עשוי לעזור לכם להבדיל ביניהם באמצעות השדה `tag`:

```
sky.setTag("sky");
```

התגית חסרת משמעות מבחינת המנוע – היא רק לשימושכם.
לבסוף החזירו את `sky` מהשיטה.

אז הגדרנו פונקציה שאחראית לייצור השמיים. קראו לה מ-`initializeGame` של `PepseGameManager` ודאגו להוסיף את האובייקט שחזר מהפונקציה לשכבה מתאימה ע"י קריאה ל:

```
gameObjects().addGameObject(sky, skyLayer);
```

(שימו לב שאתם משתמשים ב- `windowController` כדי לקבל את `dimensions` של המשחק).
הריצו; קיבלתם שמיים בצבע שמיימי כרקע לחלון הריק? (בשלב הזה אמור להיות לכם מסך עם רקע כחול בלבד).
אם תרצו, בשלב זה תוכלו להוסיף לשמיים גם עננים, ציפורים, צלחות מעופפות וכל דבר שעולה בראשכם.

2. קרקע

2.1. המחלקה `Block`

עכשיו נגדיר את המחלקה `Block` בקובץ `Block.java`, שתהיה הבסיס לכל המלבנים הללו שרואים על המסך:

```
public class Block extends GameObject{
    public static final int SIZE = 30;

    public Block(Vector2 topLeftCorner, Renderable renderable) {
        super(topLeftCorner, Vector2.ONES.mult(SIZE), renderable);
        physics().preventIntersectionsFromDirection(Vector2.ZERO);
        physics().setMass(GameObjectPhysics.IMMOVABLE_MASS);
    }
}
```

כפי שניתן לראות, אובייקט `Block` כמעט זהה ל-`GameObject` רגיל, רק שמוגדרים עבורו כמה מאפיינים ייחודיים:

2.1.1. הגודל שלו קבוע (ר' הקריאה ל-`super`).

2.1.2. בשורה הבאה מוגדר שעצם אחר לא יכול לחלוף על גבי העצם הזה, מאף כיוון (פרמטר אחר מ-`Vector2.ZERO` היה מגדיר שהבלוק הזה חוסם תנועה רק מכיוון אחד). שימו לב שתנועה כזו תיחסם רק בין שני עצמים שמעוניינים בכך (קרי, שהשיטה הזו נקראה עבור כל אחד מהם בנפרד). בנוסף, לשורה הזו אין אפקט כך או כך עבור עצמים שבלאו הכי לא מוגדרת התנגשות בינם לבין עצמם זה, למשל עצמים שבשכבה (`Layer`) שלא מוגדרת התנגשות בינה לבין השכבה של עצמם זה.

2.1.3. השורה האחרונה אומרת שאם וכאשר המנוע אכן חוסם התנגשות בין עצם זה לאחר, העצם הזה לא אמור להידחק או לזוז בשל ההתנגשות, כי אם רק העצם השני.

2.1.4. שתי השורות האחרונות יחד נועדו לגרום לכך שהשחקן, מים, עלים, או כל דבר אחר שאמור שלא ליפול דרך הקרקע אכן יתנגש בה, ושהקרקע לא תידחף מטה בשל כך.

2.1.5. הגדירו את המחלקה Block.

שאלה למחשבה: Block לא דורסת שום שיטה של GameObject, וגם לא מוסיפה שיטות או שדות משלה. בקלות ניתן להגדיר שיטה סטטית בשם create שמייצרת GameObject בעל אותם מאפיינים ומחזירה אותו. מה היתרון של הגדרת מחלקת בת במקרה הזה?
רמז: איך בגישה החלופית הזו תגדירו בהמשך "בלוק" עם התנהגות ייחודית עבור OnCollisionEnter למשל?

2.2. המחלקה Terrain

אחריות המחלקה:

- לייצר את כל בלוקי הקרקע הדרושים.
 - בנוסף, המחלקה תאפשר לעצמים אחרים לדעת מהו גובה הקרקע בקואורדינטת X נתונה.
- שימו לב שמופע של Terrain אינו GameObject בעצמו; הוא רק אחראי לייצור GameObjects אחרים (בלוקי הקרקע).
- 2.2.1. ראשית, נגדיר בנאי:

```
public Terrain(Vector2 windowDimensions, int seed)
```

ושדה בשם groundHeightAtX0 שמכיל את גובה הקרקע הרצוי ב-x=0. למשל, אתם יכולים לאתחל אותו עם גובה החלון כפול 2/3, ואז ב-x=0 האדמה תכסה בערך את השליש התחתון של המסך (ב-seed תוכלו להשתמש כדי לאתחל מחולל מספרים אקראיים כפי שנסביר [במלק](#) של עולם אינסופי).

2.2.2. בואו ניגש דווקא לתפקיד השני של המחלקה – להגדיר לעצמה ולעצמים אחרים מה גובה הקרקע הרצוי. נגדיר במחלקה Terrain את השיטה (זו דוגמה, אתם תצטרכו לחשוב על משהו מורכב יותר אח"כ):

```
public float groundHeightAt(float x) { return groundHeightAtX0; }
```

נסביר: השיטה הנ"ל מקבלת מיקום על ציר x של החלון שלנו, היא מחפשת עבורנו מה הגובה של הרצפה. בדוגמה הנ"ל, לכל מיקום בחלון המשחק נקבל גובה אחיד.

לקראת ההגשה, על תוואי הקרקע להיות מעניין יותר. הנה דרך שלא תעבוד: גובה האדמה בכל קואורדינטה יהיה אקראי. נכון, הקרקע תהיה אחרת בכל הרצה, אבל היא לא תהיה רציפה. דרך אחרת להפוך את תוואי הקרקע לפחות צפוי היא פשוט להשתמש בהרכבה של פונקציות סינוס עם גדלים, זמני מחזור, ופאזות שונים. כדי לקבל בכל פעם קרקע אחרת, אפשר להפוך את הפאזות לאקראיות.

הצורך בפונקציה שהיא אקראית למראה מחד, ורציפה מאידך, הוא צורך נפוץ. פונקציה כזו נקראת "פונקציית רעש חלקה" (smooth noise-function), ורבים וטובות מימשו סוגים רבים של פונקציות כאלו. **Perlin Noise** הוא אלגוריתם פופולרי לפונקציה כזו. בקבצים המסופקים לכם תוכלו למצוא את המחלקה [NoiseGenerator](#) שבעזרתה תוכלו לייצר Perlin Noise לפי התיעוד במחלקה.

2.2.3. השיטה השנייה של Terrain מייצרת רשימה של כל בלוקי הקרקע בתחום x-ים מסוים, על פי הגבהים שמגדירה groundHeightAt. שמה createInRange. עדיין לא נגדיר אותה, נתחיל מלבנות מלבן בודד:

```
public List<Block> createInRange(int minX, int maxX) { ... }  
(שימו לב לייבא את java.util.List ולא להשתמש ב-awt.List)
```

כדי לייצר בלוקים, כדאי להגדיר את הצבע הבסיסי של אדמה:

```
private static final Color BASE_GROUND_COLOR = new Color(212,  
123, 74);
```

ואז אפשר להגדיר את ה-Renderable:

```
new RectangleRenderable(ColorSupplier.approximateColor(  
BASE_GROUND_COLOR))
```

אתם עשויים לרצות גם להגדיר תגית לבלוק האדמה (setTag), עם מחרוזת מאפיינת כמו "ground".

2.2.4. לפני שנממש את השיטה כפי שהיא אמורה להיות, ממשו אותה כרגע כך שתיצור רק בלוק אדמה יחיד במקום שיהיה נראה לעין כמו ראשית הצירים (פינה שמאלית עליונה) או מרכז המסך. ב-PepseGameManager.initializeGame צרו מופע של Terrain. קראו ל-createInRange והוסיפו את הבלוקים שחזרו מהרשימה ל-gameObjects. שימו לב שאתם צריכים לבחור את השכבה המתאימה לאדמה כך שעצמים אחרים שנופלים עליה יתנגשו בבלוקים של האדמה (Layer.STATIC_OBJECTS); אתם רואים את הבלוק שיצרתם?

2.2.5. **רק עכשיו נממש את createInRange** כך שתיצור שטח אדמה שלם על פי הגבהים שמגדירה groundHeightAt, ותחזיר את רשימת הבלוקים שיצרה. טיפ: **תמיד** מקמו בלוקים בקואורדינטות שמתחלקות ב-Block.SIZE! הבלוקים אם כן לא צריכים להיות מוגדרים **בדיוק** בקואורדינטות minX ו-maxX שקיבלה השיטה createInRange, וגם לא **בדיוק** בגבהים שמחזירה groundHeightAt. כן כדאי לוודא **שלפחות** כל תחום האיקסים המבוקש יכוסה באדמה. למשל, עבור minX=-95 ו-maxX=40, ואם Block.SIZE הוא 30, הרי שהבלוק הראשון יכול להתחיל ב-120-, והבלוק האחרון יכול להתחיל ב-30.

מבחינת קואורדינטת ה-Y: אם אנחנו מייצרים את עמודת האדמה של $x=60$, הרי שהבלוק העליון בעמודה יכול להתחיל למשל בקואורדינטת Y של:

```
Math.floor(groundHeightAt(60) / Block.SIZE) * Block.SIZE
```

כלומר בערך בגובה הנכון, אבל מעוגל לגודל שמתחלק בגודל בלוק.

מתחת לבלוק הזה יונחו כמובן עוד בלוקי-אדמה. כמה? נניח שכל עמודה תהיה בגובה 20 בלוקים:

```
private static final int TERRAIN_DEPTH = 20;
```

ממשו את השיטה. לאחר מכן הגדירו את הפרמטרים בקריאה לה (`initializeGame`-ב) כך שכל רוחב המסך יכוסה באדמה. הריצו את המשחק - קיבלתם נתח אדמה יפה?

2.2.6. בהמשך, אחרי שנגדיר דמות שיכולה לטייל בעולם הווירטואלי שלנו, נרצה לתמוך גם בעולם אינסופי שהולך ונפרס לנגד עיניה של הדמות שלנו. אתם יכולים לחשוב בינתיים איך הייתם רואים לנכון להוסיף פונקציונליות כזו, אך אנו ממליצים לעשות זאת רק אחרי שכבר יצרתם דמות שיכולה להסתובב בעולם.

שאלה למחשבה: חישבו איך אתם יכולים לעשות שימוש במנגנון השכבות על מנת לייעל את החישובים במשחק, בין איזה שכבות לאיזה שכבות צריכים לחשב התנגשויות ובאיזה לא, האם צריך לחשב התנגשויות בתוך השכבה או לא, האם צריך לחשב התנגשויות בין כל בלוקי הקרקע (לדוגמה עם ה-Avatar שניצור בהמשך) או רק עם חלק מהם וכו'. פירוט על איך לעבוד עם מנגנון השכבות תוכלו למצוא ב**[מדריך על DanoGL](#)**.

3. אור וחושך

3.1. אור וחושך ימומשו באמצעות טריק פשוט: נגדיר מלבן שחור בגודל החלון, שמוצב ממש לפני המצלמה ומסתיר את כל יתר העצמים. בהירות העולם תיעשה על ידי שינוי האטימות שלו. בצהרי היום הוא יהיה באטימות 0 (שקוף לחלוטין), ובחצות באטימות של נניח 0.5.

3.2. במחלקה Night הגדירו את השיטה הסטטית `create`:

```
public static GameObject create(Vector2 windowDimensions, float cycleLength)
```

תפקיד השיטה לייצר את המלבן לעיל על פי `windowDimensions`, ולגרום לאטימות שלו להשתנות באופן מעגלי עם זמן מחזור של `cycleLength` (מספר השניות שלוקחת "יממה"). השיטה מחזירה את עצם האובייקט שיצרה. החלק המעניין כאן הוא שינוי השקיפות, אבל קודם נתחיל מכל היתר.

3.3. צרו בשיטה עצם משחק (מופע ישיר של `GameObject`) בשם *night* עם התכונות הבאות:

- ה-`Renderable` הוא מלבן שחור (עם אטימות ברירת מחדל של 1).
- מרחב הקואורדינטות שלו הוא זו של המצלמה:

```
setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES)
```

- הוא ממלא לחלוטין את כל החלון.
- הגדירו תגית מתאימה (`setTag`) שתבדיל אותו ממופעים אחרים של המחלקה לצורכי דיבוג.

3.4. קראו לשיטה `Night.create` מ-`initializeGame` של ה-`GameManager`. בתור אורך המחזור של יממה השתמשו ב-30 שניות (אם כי כמובן שכרגע עוד אין לפרמטר הזה ביטוי), והוסיפו את האובייקט שנוצר לרשימת `gameObjects`. חשבו לאיזו שכבה הוא שייך כדי ליצור את האפקט המתאים.

3.5. הריצו; קיבלתם מסך שחור?

3.6. שינוי האטימות קל משהייתם חושבים. הכירו את המחלקה **Transition** של `DanoGameLab`. השיטה המעניינת היחידה של המחלקה היא הבנאי שלה:

```
public Transition(  
    GameObject gameObjectToUpdateThrough,  
    Consumer<T> setValueCallback,  
    T initialValue,  
    T finalValue,  
    Interpolator<T> interpolator,  
    float transitionTime,  
    TransitionType transitionType,  
    Runnable onReachingFinalValue)
```

בגדול, יצירת מופע של המחלקה מניעה שינוי כלשהו במשחק (למשל הזזה של עצם, סיבוב שלו וכן הלאה). השינוי נעשה על ידי הרצת `callback` נתון על תחום ערכים מסוים, כשהערכים הם מטיפוס `T`. גנרי `T`.

נכיר את המחלקה והפרמטרים של הבנאי על ידי יצירת `Transition` שישנה את האטימות של *night* שלנו בצורה מעגלית. הפרמטרים, לפי הסדר:

3.6.1 GameObject gameObjectToUpdateThrough

על מנת של-Transition יהיה אפקט, הוא צריך להיות מקושר לעצם-משחק כלשהו, בדרך כלל נזין כאן את עצם המשחק שה-Transition עושה בו שינוי - במקרה שלנו, נשלח את מלבן הלילה *night*.

3.6.2 Consumer<T> setValueCallback

תפקיד ה-Transition הוא להביא לשינוי בערך מסוים; זו הפונקציה שמשנה את אותו ערך. במקרה שלנו, אנחנו רוצים לשנות את האטימות של *night*. ניתן לעשות זאת עם השיטה:

```
night.renderer().setOpacity(float opacity)
```

כמו בכל *callback*, אנחנו לא מעוניינים לקרוא בפועל לשיטה *setOpacity*, אלא ליידע את Transition באיזו שיטה הוא יכול להשתמש בשביל להביא לשינוי. לכן בתור הפרמטר השני נשלח את ה-*method reference*:

```
night.renderer()::setOpacity
```

3.6.3 T initialValue

מה צריך להיות הערך הראשוני, בתחילת ה-Transition. אם נניח שהמשחק מתחיל בצהרי היום, הרי שה-*opacity* הראשוני של *night* צריך להיות 0 (שקוף לחלוטין).

3.6.4 T finalValue

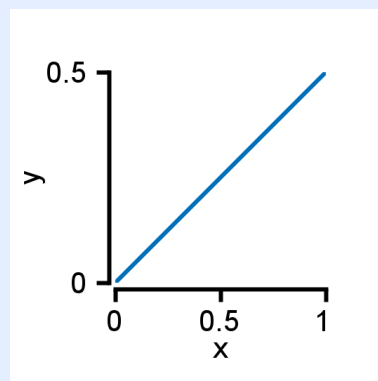
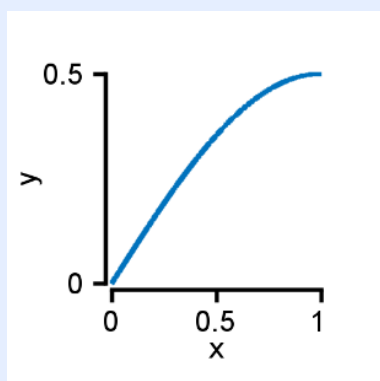
ערך הקצה השני של האטימות הוא חצי:

```
private static final Float MIDNIGHT_OPACITY = 0.5f;
```

3.6.5 Interpolator<T> interpolator

יש הרבה (אינסוף!) דרכים לעבור מ-0 ל-0.5f תוך זמן נתון. דמיינו פונקציה שערכה ב- $x=0$ הוא 0, וב- $x=1$ ערכה 0.5. מה צורת הפונקציה הרצויה? האם היא:

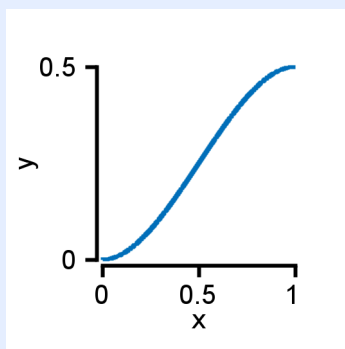
קו לינארי: אולי פולינום מדרגה גבוהה יותר? אולי סינוסידיאלית:



טיפוס הפרמטר, הממשק `Interpolator`, הוא ממשק פונקציונלי שמאפשר לשלוט על צורת הפונקציה הזו. לא ניכנס לממשק הזה לעומק כי לנוחיות המשתמשים המחלקה `Transition` כבר כוללת מספר קבועים מסוג `Interpolator` שעונים על הרבה מהצרכים. במקרה שלנו נשתמש בקבוע:

```
Transition.CUBIC_INTERPOLATOR_FLOAT
```

שישנה את האטימות מ-0 ל-0.5f על פי הפולינום מהמעלה השלישית הבא:



כפי שאתם רואים, שיפוע הפונקציה הוא קרוב ל-0 גם ב- $x=0$ וגם ב- $x=1$. אם כן משמעות השימוש באינטרפולטור הזה על פני אינטרפולטור לינארי היא שהבהירות תהיה כמעט 1 גם מעט אחרי הצהריים, והחושך יהיה כמעט במלואו גם מעט לפני חצות, קצת כמו במציאות.

3.6.6 float transitionTime

הזמן שלוקח לעבור מהערך `initialValue` (כלומר 0) ל-`finalValue` (כלומר 0.5f). חשבו מה הערך המתאים כאן.

3.6.7 TransitionType transitionType

הפרמטר הוא enum בעל הערכים האפשריים הבאים:

```
enum TransitionType {  
    TRANSITION_ONCE,  
    TRANSITION_LOOP,  
    TRANSITION_BACK_AND_FORTH  
}
```

כלומר הוא מאפשר לנו לבחור את אופן המעבר: האם יש לבצע את המעבר מהערך הראשוני לסופי רק פעם אחת (TRANSITION_ONCE), האם אחרי שמגיעים לערך הסופי צריך להתחיל שוב מערך ההתחלה, כלומר אחרי שמגיעים ל-0.5 לחזור מיד ל-0 וחזור חלילה (TRANSITION_LOOP), או האם אחרי שמגיעים לערך הסופי צריך לחזור ברוורס אל הערך הראשוני וחזור חלילה (TRANSITION_BACK_AND_FORTH). חשבו אילו מהערכים לעיל מתאימים עבורנו.

3.6.8 Runnable onReachingFinalValue

שיש להריץ כאשר מגיעים לערך הסופי. לא רלוונטי במקרה הספציפי שלנו, אז נשלח null.

3.6.9 בסך הכל, על מנת לגרום לעצם *night* לשנות את אטימותו כפי שרצינו ניצר את המעבר הבא (החליפו את סימני השאלה בערכים המתאימים לפי ההוראות לעיל):

```
new Transition<Float>(  
    night, // the game object being changed  
    night.renderer()::setOpacity, // the method to call  
    0f, // initial transition value  
    MIDNIGHT_OPACITY, // final transition value  
    Transition.CUBIC_INTERPOLATOR_FLOAT, // use a cubic interpolator  
    ???, // transition fully over half a day  
    Transition.TransitionType.???, // Choose appropriate ENUM value  
    null  
); // nothing further to execute upon reaching final value
```

שימו לב שייצור העצם הנ"ל הוא מספיק: אין צורך "לעשות" עם העצם הנוצר שום דבר נוסף (אין אפילו הכרח לשמור את העצם המוחזר בפרנס), כי בתוך הבנאי המעבר כבר "מתעלק" על העצם *night* לצרכיו, דרך מנגנון פנימי של הספרייה. הדביקו את השורות לעיל לסוף השיטה `Night.create` (לפני ה-`return` כמובן), והפלא ופלא, אם הגדרתם את הקבוע `MIDNIGHT_OPACITY` לחצי, ייתכן שקיבלתם את מחזור האור והחושך הרצוי.

לשם הכרת המחלקה Transition הגדרנו כאן ביחד את הקריאה המדויקת, אבל עברו עליה שוב וודאו שאתם מבינים את תפקידו של כל פרמטר, כי בפעם הבאה אתם תייצרו את המעבר בעצמכם!

מעניין לדעת: כשאנחנו מגדירים Transition, מאחורי הקלעים אנחנו מייצרים "תוספת" ל-GameObject שנשלח בפרמטר הראשון של הבנאי. יצירת ה-Transition אם כן היא למעשה רק עוד חלק בהגדרת אותו GameObject (במקרה הזה, חלק מהאתחול של night). אפשר להתחיל לראות איך הרבה מהקוד שלנו בפרויקט הזה יהווה "רק" אתחול של מופעים של GameObject בדרכים שונות ומשונות.

במידה רבה, הגמישות הזו מתאפשרת בזכות למבדות ו-method references. ה-API של Transition היה כל כך פחות ידידותי בלעדיהן, שמלכתחילה המחלקה לא הייתה מוצעת כפי שהיא. איתן, מחלקה כמו Transition שמקבלת בפרמטר השני שלה **callback strategy**, היא פשוטה להכנה מצד ספק השירות, ולשימוש מצד הלקוח.

4. שמש

4.1. השמש תיוצר במחלקה Sun, עם השיטה create:

```
public static GameObject create(Vector2 windowDimensions, float cycleLength)
```

שמחזירה את השמש, אחרי שיצרה אותה.

4.2. גם השמש יכולה להיות מופע ישיר של GameObject. בתור התחלה, צרו עיגול צהוב סטטי באמצע השמיים (במרכז ציר ה-X ואמצע גובה השמיים) (בעזרת המחלקה OvalRenderable). עבור צבע השמש אפשר להשתמש פשוט ב-Color.YELLOW. קואורדינטות השמש הן במרחב המצלמה כמובן, כמו השמיים (sun.setCoordinateSpace), ורצוי להגדיר לה תגית ייחודית (setTag).

4.3. קיראו לשיטה משיטת האתחול של המשחק, הוסיפו את המופע של השמש ל-gameObjects, וודאו שיש לכם שמש (סטטית). חשבו לאיזו שכבה להוסיף את השמש.

4.4. כדי לאפשר לשמש לנוע בשמיים במסלול מעגלי, נרצה שהשמש תסתובב בעיגול סביב נקודה שהיא במרכז קו האופק (groundHeightAtX0), כלומר ציר ה-x יהיה לפי אמצע המסך, וציר ה-y יהיה לפי גובה האדמה ההתחלתי. המעבר יהיה לפי 360 מעלות מסביב לנקודה זו, בלולאה. הלולאה תתוזמן בהתאם לזמן היממה שהגדרנו- 30 שניות. השמש תבצע סיבוב מלא ותחזור לנקודת ההתחלה.

בסוף השיטה Sun.create, צריך ליצור מופע חדש של Transition שיזיז את השמש לפי זווית נתונה (להסבר לגבי הפרמטרים של הבנאי חזרו לחלק של [אור-וחושך](#)).

4.4.1. בתור `gameObjectToUpdateThrough` כמובן שנשלח את המופע של שמש עצמה.

4.4.2. עבור ה-`callback` של `setValueCallback` נשלח את הלמבדה:

```
(Float angle) → sun.setCenter  
(initialSunCenter.subtract(cycleCenter)  
    .rotated(angle)  
    .add(cycleCenter))
```

המקבלת זווית, וקובעת את מיקום מרכז השמש לפי סיבוב של השמש `angle` מעלות מהמיקום ההתחלתי שלה. שימו לב שכדי שהלמבדה תפעל יש להגדיר לפני המעבר את המשתנה `initialSunCenter`.

4.4.3. בשביל טווח הערכים `initialValue-finalValue` נשלח את הערכים 0-360 שאלו הזוויות שהמעבר עובר ביניהן.

4.4.4. עבור `interpolator` נשתמש באינטרפולציה לינארית פשוטה, עם:

```
Transition.LINEAR_INTERPOLATOR_FLOAT.
```

4.4.5. ועבור `onReachingFinalValue` נשלח `null`.

5. הילת השמש

5.1. את הילת השמש נייצר כ-`GameObject` נפרד במחלקה `SunHalo`, עם השיטה הסטטית `:create`

```
public static GameObject create(GameObject sun)
```

שדומה בהתנהגותה לכל יתר שיטות ה-`create` שלנו.

5.2. בתור צבע ההילה, נרצה משהו עם שקיפות. אטימות מיוצגת בערוץ ה-`alpha`, או בקיצור ערוץ ה-`a`, של צבע. למשל, תוכלו להשתמש בצבע צהוב עם אטימות של 20 מתוך 255:

```
new Color(255, 255, 0, 20)
```

5.3. כמו במקרה של השמש, נתחיל מלייצר הילה סטטית. גירמו לשיטה לייצר עיגול פשוט, בגודל כרצונכם, בצבע שהגדרנו, שתקיף את השמש, והוסיפו אותו ל-`gameObjects` בשכבה המתאימה (לפני השמש). ה-`CoordinateSpace` שלו הוא כמובן זה של המצלמה (בדומה לשמיים, לשמש, ולמלבן החושך). הגדירו להילה תגית שתייחד אותה.

5.4. בשביל לקרוא לשיטה `SunHalo.create` מ-`PepseGameManager.initializeGame` נצטרך לשלוח את המופע של השמש. הריצו; קיבלתם הילה סטטית תקועה בשמיים?

5.5. מכיוון שהשיטה מקבלת את עצם המשחק sun, ההילה לא צריכה לייצר Transition חדש שיזיז אותה מפורשות: די שההילה תעתיק את המרכז שלה (sunHalo.setCenter), בכל פריים, להיות כמו זה של השמש (sun.getCenter). איך?

אפשרות אחת היא לייצר תת-מחלקה של GameObject, בה נדרוס את השיטה update. העניין יצריך מספר שורות ספורות, כ-10, ויהפוך את הקוד למורכב יותר רק בקצת, בכך שמתווספת מחלקה קטנה מאוד.

ישנה גם אפשרות אחרת, מבוססת אסטרטגיה. אם נניח שלעצם שיצרתם קוראים sunHalo, תוכלו לקרוא לשיטה sunHalo.addComponent. השיטה (הנמצאת כבר במחלקה GameObject) מצפה לפרמטר מסוג Component, (סוג של Observer) שהוא הממשק הפונקציונלי הבא:

```
@FunctionalInterface
public interface Component {
    void update(float deltaTime);
}
```

השיטה addComponent מצפה ל-callback שמקבלת float ולא מחזירה דבר. השיטה מבטיחה לקרוא ל-callback שהתקבל בסוף כל עדכון (update) של העצם. כלומר, שלחו ל-sunHalo.addComponent ביטוי למדא שמקבל deltaTime ומעדכן את מרכז ההילה להיות כמרכז השמש. גוף הלמדא לא צריך לעשות שימוש בפרמטר שלו deltaTime במקרה הזה. שימו לב, אפשר לגרום לעצם sunHalo לעקוב אחרי sun גם בשורה אחת, ומבלי טיפוסים נוספים.

6. הוספת דמות (Avatar)

6.1. בחלק הזה, נוסיף למשחק שלנו דמות שיכולה לזוז בעולם.

הדמות צוברת אנרגיה כאשר היא במנוחה, ומשתמשת באנרגיה כדי לבצע פעולות שונות. הדמות צריכה להיות מסוגלת לבצע את הפעולות הבאות:

- לנוע לצדדים באמצעות החיצים.
- לקפוץ באמצעות מקש הרווח.

6.2. אתחול ראשוני

6.2.1. בשלב זה תוכלו לקחת השראה מהדמות הפשוטה של פלטפורמר שסיפקנו לכם בקובץ [Platformer.java](#).

שימו לב: אין להגיש את Platformer.java.

6.2.2. בתור התחלה, הגדירו ב-pepse.world.avatar את המחלקה Avatar, עם הבנאי הבא:

```
public Avatar(Vector2 topLeftCorner,
              UserInputListener inputListener,
```

ImageReader imageReader)

כאשר `topLeftCorner` הוא המקום במסך עליו אמורה הדמות לעמוד (פינה שמאלית עליונה). כשאתם מאתחלים את הדמות, עליה להיות בגובה הקרקע.

6.2.3 בתור `Renderable` כרגע השתמשו בתמונה `"idle_0.png"` המסופקת לכם בתיקיה `assets`. הורידו את התיקיה ומקמו אותה ישירות תחת תיקיית `src` של הפרוייקט שלכם (כלומר מחוץ לחבילה `pepse`), וקראו את התמונה באמצעות ה-`imageReader`:

```
imageReader.readImage("assets\\idle_0.png", true);
```

כעת תוכלו להעביר את ה-`Renderable` שנוצר בקריאה ל-`super` (כמו ב-`Platformer.java`) או למשל באמצעות `renderer().setRenderable`. שימו לב: כאשר אתם משתמשים בתמונות שב-`assets` בקוד שלכם, עשו זאת בעזרת נתיב יחסי (רלטיבי), כמו בדוגמה לעיל.

6.2.4 הוסיפו לדמות לוגיקה בסיסית של תזוזה ימינה ושמאלה עם החיצים, וקפיצה באמצעות `Platformer.java`. בדומה ל-`Avatar` ב-`PepseGameManager`, בדקו שהדמות נוצרה במקום הנכון (חישבו איפה כדאי לאתחל את הדמות כך שהיא תיווצר בדיוק על הקרקע), ושהיא זזה ימינה שמאלה וקופצת בהתאם להוראות.

6.3 מצבי הדמות

6.3.1 לדמות יהיו 3 מצבי תזוזה שישפיעו על התגובה שלה לפעולות שונות:

- **idle (מנוחה):** הדמות עומדת במקום על הקרקע.
- **run (ריצה):** הדמות נעה ימינה או שמאלה על הקרקע.
- **jump (קפיצה/אוויר):** הדמות נמצאת באוויר, במהלך עלייה או נפילה (הדמות קופצת למעלה ואז נופלת חזרה לקרקע).

6.3.2 מצב הדמות ישפיע על האנימציה שתוצג עבור הדמות (חלק 6.7), צבירת ובזוז האנרגיה שלה (חלק 6.4) וכן על התגובה שלה לקלטים מהמשתמש (חלק 6.5).

6.4 אנרגיה

6.4.1 כעת נוסיף למשחק את המרכיב של צבירת ובזוז אנרגיה. לדמות יש משאב אנרגיה שחיוני לביצוע פעולות.

6.4.2 טווח האנרגיה נמדד באחוזים, ולכל אורך המשחק כמות האנרגיה של הדמות תנוע בין **0 ל-100**. המשחק מתחיל כך שיש לדמות אנרגיה מקסימלית של **100%**.

6.4.3 עלויות פעולות:

- **ריצה:** דורשת 2 יחידות אנרגיה, בכל עדכון.
- **קפיצה:** דורשת 20 יחידות אנרגיה, לביצוע חד-פעמי.
- **קפיצה כפולה (Double Jump):** דורשת 50 יחידות אנרגיה לביצוע חד-פעמי.

6.5. לוגיקת עדכון ההתנהגות (מתודת update)

6.5.1. כללי תנועה וקלט

1. **תנועה ימינה/שמאלה** - לחיצה על מקשי החיצים ימינה/שמאלה:
 - אם הדמות נמצאת **על הקרקע** ויש **לפחות 2 אנרגיה** לחיצה על מקשי החיצים ימינה/שמאלה גורמת לדמות לנוע בכיוון החץ. אם הדמות ללא סף האנרגיה הנדרש, לא יקרה כלום.
 - אם הדמות נמצאת **באוויר** לחיצה על מקשי החיצים ימינה/שמאלה גורמת לדמות לנוע בכיוון החץ **בלי לבזבז אנרגיה**. אם הדמות זזה ימינה/שמאלה באוויר זה יהיה בנוסף לתזוזה למעלה/למטה ולא במקום.
2. **קפיצה** - לחיצה על מקש רווח גורמת לקפיצה במקרים הבאים:
 - אם **הדמות על הקרקע** ויש **לפחות 20 אנרגיה** לחיצה על מקש הרווח גורמת לקפיצה רגילה.
 - אם **הדמות באוויר במהלך נפילה** (כלומר מצב jump עם מהירות חיובית בציר y) ויש **לפחות 50 אנרגיה** לחיצה על מקש הרווח גורמת לקפיצה **נוספת מהמיקום הנוכחי**.
 - אם הדמות ללא סף האנרגיה הנדרש, לא יקרה כלום.

6.5.2. לוגיקת צבירה ואיבוד אנרגיה (בכל update)

מצב	שינוי אנרגיה
Idle (מנוחה על הקרקע)	נצבר 1 יחידות אנרגיה
Run (ריצה על הקרקע)	מתבזבזות 2 יחידות אנרגיה
Jump (באוויר)	אין שינוי (לא נצבר ולא מתבזבז)
קפיצה חדשה (רגילה)	מתבזבזות 20 יחידות אנרגיה באופן חד-פעמי מיידי
קפיצה חדשה מהאוויר (כפולה)	מתבזבזות 50 יחידות אנרגיה באופן חד-פעמי מיידי

כמה הבהרות:

- אם הדמות עומדת ולא זזה, היא מקבלת נקודת אנרגיה אחת **רק כאשר היא על הקרקע**.
- במקרה של לחיצה על שני החצים, ימינה ושמאלה, יחד באותו update - הדמות לא אמורה לזוז ולא אמורה לרדת אנרגיה.
- לא ניתן לקפוץ קפיצה רגילה מהאוויר. היינו, מלבד קפיצה כפולה, כשהדמות באוויר ונלחץ על קפיצה (רווח), היא לא אמורה לעלות כלפי מעלה, גם אם יש לה מספיק אנרגיה לקפיצה רגילה, ועל כן היא לא אמורה לאבד אנרגיה. כלומר, הדמות תוכל לקפוץ קפיצה

רגילה שוב רק כאשר היא נחתה על הקרקע. הדמות תקפוץ באוויר רק אם יש לה מספיק אנרגיה לקפיצה כפולה.

- אם ערכי ההורדה וההעלאה של האנרגיה יוצרים אצלכם בעיות, כמו ריצוד של הדמות בזמן ריצה על הקרקע, אתם מוזמנים לשנות את הערכים ולציין זאת ב-README.
- **שימו לב!** אם לדמות אין מספיק אנרגיה אז היא **לא תוכל לבצע פעולות** כלל ותצטרך לעמוד במקום/להישאר במצב הנוכחי. למשל אם לדמות יש 5 נק' אנרגיה והיא תנסה לקפוץ (שדורש 10 נק' אנרגיה) אז היא לא תצליח לקפוץ (וכמובן לא תרד האנרגיה). וכן, אם הדמות במצב ריצה ונגמרה לה האנרגיה היא תעבור למצב מנוחה גם אם אחד החיצים נלחץ.

6.6 ממשק תצוגת האנרגיה

6.6.1 כעת נוסיף למסך שלכם חיווי שמראה בכל זמן נתון את כמות האנרגיה הנוכחית של הדמות.

כדי להיזכר איך עושים זאת, תוכלו לחזור לתרגיל 2 ולהסתכל בקוד שהכנתם עבור הצגת כמות החיים שנשארה במשחק. העיצוב של החלק הזה נתון לבחירתכם.

6.6.2 איך מי שמטפל בהצגת כמות האנרגיה יהיה מעודכן באנרגיה של הדמות שלנו? אם בעבר כדי להעביר מידע בין מחלקות שהיינו צריכים ליצור קשר של הכלה בין מחלקות (או לשבור אנקפסולציה לא עלינו...), עכשיו אתם כבר יודעים להשתמש גם בתכנות פונקציונלי, ואפשר פשוט לשלוח למחלקה callback שיספק לה נתונים ממחלקה אחרת בלי שהיא תכיר אותה בכלל.

אבל... האם ממשק תצוגת האנרגיה צריך להיות מעודכן בכל update, או שמא בכלל אפשר לעדכן אותו רק כאשר נעשה שינוי באנרגיה? חשבו על מנגנון בקוד שיוכל לאפשר עדכונים של תצוגת האנרגיה רק כאשר נעשה שינוי באנרגיה.

6.7 אנימציה

6.7.1 כעת נרצה להציג אנימציה של הדמות במקום תמונה סטטית, וכן נרצה שהאנימציה תתחלף בהתאם לתזוזה של הדמות.

אפשר ליצור אנימציה של הדמות ע"י החלפה בלולאה של סדרת תמונות של התזוזה בה הדמות נמצאת במקום להציג את הדמות בצורה סטטית ע"י תמונה בודדת. לכל תזוזה, יש סדרת תמונות מתאימה בקבצים שסיפקנו לכם בתיקייה assets:

- עבור מצב idle בו הדמות לא זזה יש 4 תמונות עם השמות: idle_0.png, idle_1.png, idle_2.png, idle_3.png
- עבור מצב jump בו הדמות עולה או יורדת (בלי תזוזה לצדדים) נשתמש בתמונות: jump_0.png, jump_1.png, jump_2.png, jump_3.png
- עבור מצב run בו הדמות זזה לצדדים (בין על הקרקע ובין באויר) נשתמש בתמונות: run_0.png, run_1.png, run_2.png, run_3.png, run_4.png, run_5.png

כדי להציג אנימציה בעזרת סדרת תמונות השתמשו במחלקה `AnimationRenderable` המקבלת בבנאי שלה את סדרת התמונות וכן את אורך הזמן בין החלפת התמונות. (אפשר גם לתת ל-`AnimationRenderable` תיקייה בבנאי ולא רק רצף תמונות). שימו לב:

- תוכלו להחליף את ה-`Renderable` של העצם בזמן ריצה כאשר הוא עובר בין מצבים באמצעות המתודה `.setRenderable().renderer()`.
- כדי לשקף את ה-`Renderable` אופקית השתמשו במתודה: `.renderer().setIsFlippedHorizontally`
- ניתן גם להתאים לכל סט תמונות מימדים (`dimensions`) מסוג `Vector2` כדי לשמור על גודל אחיד, ולעדכן את המימדים באמצעות `.setDimensions`.
- בקבצים שיש במודל סיפקנו לכם תמונות עבור דמות שיש לה 3 מצבי תזוזה (אפשר להגדיר יותר, לדוגמה אנימציה של נפילה מול קפיצה, וכמובן גם מצבים חדשים). תוכלו כמובן גם להשתמש בכל דמות אחרת שאתם רוצים, ואפשר למצוא רעיונות כאן.

שימו לב במידה שאתם נתקלים במצב בו הדמות נכנסת לתוך האדמה תוכלו להוסיף במתודה `onCollisionEnter` של `Avatar` את קטע הקוד הבא (ודאו שאתם משנים את התג "block" לתג המתאים בתכנית שלכם):

```
if(other.getTag().equals("block") && isFalling()){
    System.out.println();
    this.transform().setVelocityY(0);
}
```

כאשר `isFalling()` היא מתודה שאתם יכולים להגדיר המתארת נפילה של הדמות. אפשר גם לנסות להשתמש במקומה בבדיקה: `collision.getNormal().y() > 0`, רק וודאו שהתנאי שאתם בודקים עובד. בנוסף אם נתקעים באובייקטים מהצדדים (ציר x) תוכלו לטפל בצורה דומה.

7. צמחיה - עיצוב לבחירתכם

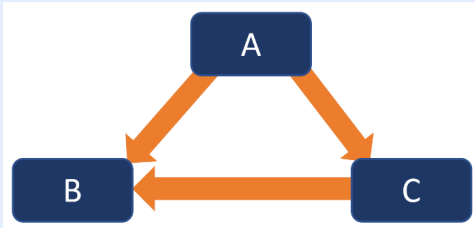
- 7.1 בחלק הזה נגדיר בשלבים את תכונות העצים כאשר בהתחלה נגדיר איך נראה עץ סטטי, שלא זז, ואז נוסיף לעץ תכונה של תזוזה של העלים. מומלץ לעבור על כל הפרק לפני שמתחילים בעיצוב כדי לקבל את התמונה השלמה של העצים.
- 7.2 את רוב העיצוב של החבילה הזו נשאיר לכם, למעט הדרישה למחלקה בשם `Flora` ובה שיטה עם החתימה:

```
public ??? createInRange(int minX, int maxX) { ... }
```

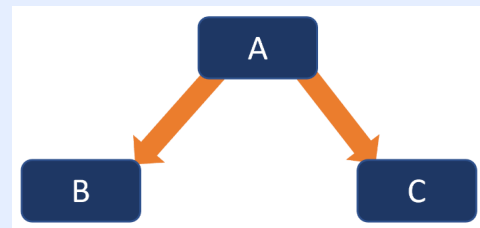
שתיצור עצים (במיקומים אקראיים/קבועים מראש לבחירתכם) בתחום x-ים מסוים ותחזיר מבנה נתונים הכולל את האובייקטים שנוצרו. סוג מבנה הנתונים וטיפוס האובייקטים תלוי בעיצוב שלכם.

הכוונה עיצובית: החופש העיצובי בתכנון הספרייה הזו לא אומר שצריך להתפרע עם איזה עיצוב גרנדיוזי, כי הפונקציונליות לא ענקית. כן ניתן עצה אחת כללית בנוגע לקשר בין העצים לבין Terrain, היות והעצים יצטרכו לדעת איפה בעולם להופיע.

עדיף על גרף התלויות הבא:



גרף התלויות הבא:



למזלנו, את הגרף השני אפשר לעתים להפוך לראשון. אם התלות של C ב-B נובעת מהצורך לקרוא לשיטה של המחלקה B, דרך טובה וקלה לחסוך את התלות היא ש-A תשלח ל-C ישירות את השיטה הדרושה, באמצעות callback. כך C תכיר את ה-callback שהיא מקבלת, הלוא היא בעלת טיפוס כללי של functional interface, ולא את המחלקה הקונקרטית B. זהו מקרה פרטי של העקרון "תכנות לממשק, לא למימוש" (program to interface, not implementation). אם ההכוונה העיצובית הזו לא מובנת לכם בשלב הזה, נסו לקרוא אותה שוב לאחר שעברתם על כל הפרק של יצירת העצים.

7.3 עצים סטטים

7.3.1 הגדירו ב-pepse.world.trees מחלקות לשם שתילת עצים **סטטיים** (כשלב ראשון): הם כוללים גזע בגובה אקראי וצמרת ריבועית מסביב לקצה הגזע, עם עלים שעוד לא זזים. בנוסף לעיצוב, נשאר בידיכם גם את כל פרטי המימוש, אבל כן ניתן כמה הכוונות. הגזע יכול להיות פשוט מלבן עם צבע שהוא וריאנט של:

```
new Color(100, 50, 20)
```

ואילו העלים יכולים להיות מורכבים מבלוקים בעלי צבע שהוא וריאנט של:

```
new Color(50, 200, 30)
```

7.3.2 בחירת מיקום העצים פשוטה: לכל עמודה אפשר להטיל מטבע מוטה (נניח עם הסתברות של 0.1), ואם הוא יוצא אמת, "שותלים" בעמודה הזו עץ (קרי: יוצרים בלוקים עבורו).

"הטלת מטבע מוטה" נעשית על ידי בחירת מספר אקראי בין 0 ל-1 ובדיקה האם הערך גדול או קטן מסף רצוי. כדי ליצור עלווה לא צפופה מידי מומלץ גם להטיל מטבע מוטה גם כדי להחליט כמה עלים ליצור בראש כל עץ. כמובן, את העץ צריך לשים בגובה הנכון של הקרקע, ר' השיטה `Terrain.getHeightAt`.

שימו לב! הדמות במשחק שלנו צריכה להתנגש בגזע העץ, אך לא בעלים, וכן העלים לא אמורים להתנגש אחד בשני. (ראו סרטון הדגמה). חשבו היטב באיזו שכבה לשים כל אובייקט שיצרתם כדי לקבל את התנהגות המשחק הרצויה.

7.4. תנודות העלים ברוח

7.4.1. כדי לנענע את העלים ברוח, די להוסיף שני מעברים (Transitions) מסוג BACK_AND_FORTH: אחד על זווית העלים, ואחד על רוחב העלים (גם מעברים שיוגדרו על תכונות אחרות של העלים יעשו את העבודה בצורה דומה). הזווית נשלטת על ידי:

```
leaf.renderer().setRenderableAngle(angle)
```

ורוחב העלים נשלט על ידי:

```
leaf.setDimensions(dimensionsAsVector2)
```

הגדירו Transitions כאלו לבלוקים של העלים וצפו בתוצאה.

7.4.2. אולי שמתם לב שאם כל העלים נעים יחד בדיוק באותו קצב ותזמון זה נראה רע – אם תסתכלו החוצה בחלון, תראו שלא כך הם נעים בעולם האמיתי. אם כן נניח שברצוננו להתחיל ב-Transition מסוים, אבל אנחנו לא רוצים שהוא יתחיל בדיוק ברגע זה, אלא בעוד זמן כלשהו. בנוסף נניח שלכל העלים יש למעשה בדיוק את אותו Transition, אבל כל אחד מתחיל אותו בדיליי מעט אחר – למשל אחד כעבור 0.1 שניות מיצירתו, ואחר לאחר 0.5 שניות. על מנת להריץ קוד נתון **בדיליי** (לגרום לו לרוץ בעוד זמן נתון), ישנה ב-DanoGameLab מחלקה שימושית נוספת: ScheduledTask. הבנאי שלה הוא בעל החתימה הבאה:

```
ScheduledTask(  
    GameObject gameObjectToUpdateThrough,  
    float waitTime,  
    boolean repeat,  
    Runnable onElapsed)
```

הבנאי מקבל:

- עצם משחק שאליו המשימה רלוונטית (בדומה ל-Transition),
 - את מספר השניות שיש להמתין לפני הרצת המשימה,
 - דגל שאומר האם המשימה אמורה לחזור על עצמה כל *waitTime* שניות או שמא יש להריץ את המשימה רק פעם אחת ולשכוח ממנה,
 - ומופע מסוג Runnable שמייצג את המשימה המדוברת.
- מכיוון ש-Runnable הוא ממשק פונקציונלי (של פונקציה שלא מקבלת כלום ולא מחזירה כלום), המחלקה עובדת היטב עם למדות ומצביעים לשיטות. בדומה ל-Transition, אחרי שיצרתם מופע של ScheduledTask אין צורך לעשות איתו עוד שום דבר נוסף – עצם יצירת ה-ScheduledTask מספיקה להרצת המשימה

המתוזמנת, בזכות עצם המשחק שנשלח בפרמטר הראשון ושהמשימה המתוזמנת "מתעלקת" על ה-update שלו.

במקרה שלנו, נרצה לדאוג לכך שה-Transition של העלה יוצר בדילי קטן וקצת שונה לכל עלה, ואפשר לעשות זאת ע"י שליחת callback שיוצרת את ה-Transition ל-ScheduledTask.

נשאיר לכם את הפרטים המדויקים של איך לגרום לעלים להתנועע בצורה "מציאותית". אבל הנה טיפ אחרון: החכמה כאן היא לחלק את הקוד לשיטות קצרות ומחלקות מוגדרות היטב. אמנם הסגנון התכנותי שלנו בתרגיל הזה השתנה מעט, אבל שמירה על שיטות קומפקטיות, ועל קבצים עם אחריות מצומצמת היא עקרון תכנותי אוניברסלי.

7.5. הוספת פירות

כעת נרצה להוסיף למשחק שלנו פירות שיופיעו על העצים והדמות שלנו תוכל "לאכול" אותם כדי לצבור עוד אנרגיה. העיצוב של חלק זה כמו כל העיצוב של העצים נתון לבחירתכם. כמה הנחיות להתנהגות הפירות:

- 7.5.1 הפירות יהיו פשוט אובייקטים בצורת עיגול.
- 7.5.2 צבע הפירות נתון לבחירתכם.
- 7.5.3 גודל הפרי יהיה לכל היותר כגודל העלים.
- 7.5.4 הפירות יופיעו על צמרות העצים בין או על העלים. שימו לב שהעלים והפירות לא אמורים "להתנגש".
- 7.5.5 בכל פעם שדמות תתנגש בפרי, הפרי יעלם, והדמות תקבל 10 נקודות אנרגיה.
- 7.5.6 פרי ש"נאכל" יופיע מחדש במשחק בתום מחזור שלם מרגע הוא נעלם (כזכור, מחזור של המשחק הוא 30 שניות).
- 7.5.7 אם פרי מופיע בתוך גזע אחר מכיוון שהעלווה של העץ שלו חופפת לגזע, זה תקין, פשוט ייתכן והפרי לא יהיה נגיש לאכילה.

7.6. סיכום

בשלב הזה של התרגיל ננסה לעשות סדר בעניין השכבות. כזכור, הדמות שלנו צריכה להתנגש באדמה, בגזעי העצים, אך לא בעלים שלהם (כמובן שלא בשמש או בהילת השמש). אם זה לא המצב אצלכם בתכנית, בדקו באיזו שכבה כל אובייקט נמצא, ונסו לסדר אותם מחדש כדי שתקבלו את ההתנהגות הרצויה.

8. עולם אינסופי

- 8.1 כעת, הנחו את המצלמה לעקוב אחרי הדמות שיצרתם. אם שם המשתנה שבו נשמרה הדמות הוא avatar, תצטרכו להכניס שורה לקוד שלכם שורה שנראית פחות או יותר ככה:

```
setCamera(new Camera/avatar, Vector2.ZERO,  
windowController.getWindowDimensions(),
```

```
windowController.getWindowDimensions()));
```

כאשר הפרמטר השני בבנאי של Camera הוא המרחק של האובייקט הנעקב ממרכז המסך. לכן, אם נרצה שבמצב ההתחלתי הקואורדינטה (0,0) על המסך תתאים לקואורדינטה (0,0) בעולם (כמו שהיה המצב לפני שהוספנו את הדמות), נצטרך להחליף את Vector2.ZERO במשהו כמו:

```
windowController.getWindowDimensions().mult(0.5f) - initialAvatarLocation
```

8.2. בהנחה שבקריאה ל-Terrain.createInRange ול-Flora.createInRange לא הגדרתם $\infty = \text{maxX}$ ו- $\infty = \text{minX}$, כנראה שדי מהר תגלו שהגעתם לסוף העולם:



כדי להימנע ממצב זה, עלינו להגדיר עולם אינסופי. כמובן שלא נוכל לייצר מראש בלוקי אדמה ועצים אשר יכסו את כל ערכי ה-x האפשריים, ולכן אין לנו ברירה אלא להמשיך לייצר אותם בזמן ריצה. יש לכם יד חופשית לחלוטין בבחירת האופן שבו תבצעו זאת, אבל כדאי לזכור שצעדי עדכון וחישוב התנגשויות בין אובייקטים שנמצאים הרחק מחוץ לאזור הנראה של המסך סתם יגזלו לנו משאבי חישוב ויהפכו את הריצה לאיטית בלי שתהיה לכך איזושהי תרומה. לכן, אנחנו ממליצים לכם למצוא דרך לצמצם כמה שאפשר את החישובים שקשורים לאובייקטים שנמצאים מחוץ לאזור הנראה. בנוסף, העולם שלנו צריך להיות עקבי – כלומר, אם החלטתם למחוק את האובייקטים שנמצאים מחוץ לאזור הנראה ולייצר אותם מחדש רק כאשר מתקרבים אליהם, שימו לב שהם צריכים להיות זהים (לפחות למראית עין) לאובייקטים שהיו שם בפעם הקודמת שעברנו באותה נקודה.

במילים אחרות, אם גובה האדמה ב- $x=0$ היה 100, גם אם נלך קילומטרים ואז נחזור שוב לאותה נקודה, גובה האדמה בה עדיין יהיה 100, ובאופן דומה, אם ב- $x=0$ היה עץ בגובה מסוים ועם מבנה עלים מסוים, גם בפעם הבאה שנבקר באותה נקודה יהיה שם עץ באותו

גובה ועם אותו מבנה עלים (וכמובן שאם לא היה בנקודה מסוימת עץ, גם בפעם הבאה שנבקר בה לא יהיה בה עץ). הצורך הזה בעקביות מוביל אותנו לסעיף הבא.

8.3. אקראיות ניתנת לשחזור

המפתח לאקראיות משוחזרת הוא היכרות עם האופן בו מספרים פסודו-אקראיים מיוצרים ברוב המימושים. כאשר אתם מייצרים עצם חדש מסוג `Random`, הוא מאותחל עם זרע (`seed`). את הזרע ניתן להעביר בבנאי, או לחילופין אם השתמשתם בבנאי הריק, יוגדר עבור העצם זרע שמבוסס על פרמטרים משתנים כמו הזמן הנוכחי. הזרע עצמו יכול להשתנות מעצם לעצם, אבל שני עצמי `Random` שאותחלו באמצעות אותו זרע, ייצרו בדיוק את אותה סדרה של מספרים "אקראיים". כלומר, עבור לולאה שקוראת ל-`nextInt` מאה פעמים, נקבל עבור שני העצמים את אותם מספרים בדיוק.

אם כן, אם העולם כולו מיוצר על ידי עצם יחיד של `Random` (שמועבר בין המחלקות), בפעם הבאה שנריץ את הסימולציה עם אותו `seed`, ייווצר אותו עולם! אבל זה לא לגמרי מדויק. זה נכון עבור עולם שמוצר כולו מיד בקריאה ל-`initializeGame`, אבל אם שטחים נוספים בעולם מיוצרים בזמן ריצה כשגוללים את המסך, סדר הקריאות ל-`Terrain.createInRange` תלוי בקלט המשתמשים.

נניח שאנחנו מייצרים עצים בעלי צורה ייחודית שתלויה במספרים אקראיים. דרך קלה להבטיח שעץ שנוצר בקואורדינטה $x=60$ תמיד יהיה אותו עץ, עבור `seed` נתון של הסימולציה, היא לייצר את העץ באמצעות עצם `Random` שהזרע שלו הוא פונקציה של הזרע הכללי של הסימולציה, והקואורדינטה 60. למשל, אם הזרע הכללי הוא `mySeed`, אפשר לייצר את העצם הבא:

```
new Random(Objects.hash(60, mySeed))
```

עץ שייעזר בעצם הנ"ל לאקראיות שלו מובטח להיות בעל אותה צורה עבור אותו מיקום ואותו זרע ראשוני, גם בהרצות הבאות.

עדכנו את הקוד שלכם כך שהעולם שנוצר יהיה עקבי – כלומר, אם החלטתם למחוק את האובייקטים שנמצאים מחוץ לאזור הנראה ולייצר אותם מחדש רק כאשר מתקרבים אליהם, שימו לב שהם צריכים להיות זהים (לפחות למראית עין) לאובייקטים שהיו שם בפעם הקודמת שעברנו באותה נקודה.

9. הוראות הגשה

9.1. הגישו את התרגיל כקובץ `jar/tar/zip` בשם `ex4` (והסיימת בהתאם, למשל `ex4.zip`).

בתוך קובץ זה, ימצאו החבילות והקבצים כמו שמתואר בסעיף 0.2. שימו לב שמותר לכם להוסיף חבילות ומחלקות כרצונכם.

9.2. קובץ ה-`README`:

9.2.1. בשורה הראשונה בקובץ יופיע שם המשתמש CSE שלכם. אם אתם מגישים בזוג, יש להפריד בין שמות המשתמש עם פסיק (שניהם באותה שורה).

9.2.2. בשורה השניה מספר תעודת הזהות. אם אתם מגישים בזוג, יש להפריד בין מספרי הזהות עם פסיק (שניהם באותה שורה).

9.2.3. השורה השלישית ריקה.

9.2.4. **החל מהשורה הרביעית ענו בקובץ על השאלות הבאות:**

• הסבירו על הדרך בה בחרתם לממש את הדמות (Avatar), התייחסו בתשובתכם לנקודות הבאות:

- פרטו על המחלקות/חבילות השונות שיצרתם בחבילה ועל הקשרים ביניהן.
- איך בחרתם לעצב את שינוי המצבים בדמות?
- איך בחרתם לעצב את שינוי האנרגיה בדמות? איך בחרתם לעדכן אותה, ואיך בחרתם לעצב ולעדכן את התצוגה שלה?

• הסבירו על הדרך שבה בחרתם לממש את החבילה trees, פרטו על המחלקות/חבילות השונות שיצרתם בחבילה ועל הקשרים ביניהן.

• אם הוספתם חבילות/מחלקות נוספות על מה שמפורט בשאלות 1-2, פרטו והסבירו עליהן.

• אם שניתם את ה-API (בין אם חבילות/מחלקות, ובין אם חתימות של מתודות או כל שינוי אחר) נא לציין זאת כאן ולהוסיף הסבר.

9.3. **תכנון קוד ובנוס מגן**

9.3.1. אנו ממליצים תמיד לתכנן את הקוד לפני שניגשים לכתיבת הקוד עצמו. כדי לעודד אתכם לעשות זאת, בתרגיל זה תהיה אפשרות להגיש תכנון מוקדם של הקוד שבוע לפני תאריך ההגשה ולקבל על כך ציון מגל לתרגיל.

9.3.2. קראו את כל ההוראות לפני שאתם מתחילים לגשת למלאכה. את החלקים המודרכים (בהם אתם נדרשים בהוראות לכתוב קוד יחד עם הקריאה) ניתן לעשות לפני תכנון הפרוייקט אך ממש לא חייב. במיוחד כדאי לתכנן ברצינות את הקוד של חלק 6 (דמות) והלאה (כלומר לסיים לקרוא ולתכנן ורק אז לגשת לכתיבת הקוד). את תכנון הקוד כדאי לעשות בדרכים שלימדנו לאורך הקורס.

9.3.3. בנוס מגן - בתרגיל זה אנחנו נאפשר בנוס מגן, הכולל גרפים ותרשימי תכנון של הפרוייקט. אלו יכולים להיות באיזה סגנון שאתם רוצים, ולא מחייבים את הפרוייקט הסופי. המטרה היא לתכנן מראש, אבל אם נדרשים לשינוי לפעמים ניתן לחרוג מהתכנון המקורי.

9.3.4. הגרפים יכולים לכלול גרף אחד גדול לכל הפרוייקט, ויכולים להיות גם גרפים קטנים לנושאים שונים. הם יכולים להיות מסוג UML, חלוקה לחבילות ופירוט המחלקות, גרף תלויות, גרפים המתארים תבניות עיצוב שונות וכד' - יש לכם כאן חופש יצירתי לתכנון מסוג שנוח לכם.

9.3.5. הבונוס הוא בגדר 10% מגן, וניתן לקבל 100 גם בלעדיו.

ציון התרגיל (כולל בונוס ציון המגן) יהיה:

$$final_ex4_grade = \max\{ex4_grade, (0.9 * ex4_grade + 0.1 * mager)\}$$

9.3.6. תאריך הגשת הבונוס הוא **28.12.2025**, ואין עליו הארכות.

בהצלחה!

10. פרולוג

דיון על שילוב פרדיגמות תכנות

אולי שמתם לב שלא השתמשנו הפעם הרבה בממשקים. למעשה, הרבה מהקוד ייצר מופעים שמסתפקים בשדות ובשיטות של `GameObject`, והם נבדלו "רק" באסטרטגיות שלהם ולא במימוש שונה של שיטות. אז מה עושות שם `Sun`, `Sky`, `Night` וכל היתר – האם תפקידן של מחלקות אינו להגדיר עצמים מסוג חדש?

ראינו שכאשר למחלקה כמו `GameObject` יש תמיכה נרחבת באסטרטגיות, יצירתו של מופע בודד, כמו עלה, יכולה להימשך בפני עצמה עשרות או מאות שורות קוד עם לוגיקה מורכבת: הגדרנו לכל עצם `Renderable` אחר, ושלחנו לו `Components` שונים כמו `ScheduledTask`, `Transition` ואחרים, שכל אחד מהם דרש נתח קוד לא מבוטל.

את כל הקוד של ייצור עצם בודד שמנו במחלקה ייעודית שמחולקת לשיטות, אבל קוד הייצור הזה הוא לא שיטה של עצם אחר. ייצור המופעים של `GameObject` נעשה במסגרת שיטות סטטיות, או שיטות מופע של עצם "סמלי" – שהוא המופע היחיד של המחלקה שלו, ושאינו לו מצב (שדות) מעניין. זה היה המקרה עם המחלקה `Terrain` וגם במקרה של `PepseGameManager`.

אז בזמן שהקוד שלנו ייצר עצמים, הוא עצמו לא היה מאורגן בעצמים, אלא ב**פונקציות** שקוראות זו לזו. תכנות מבוסס קריאה לפונקציות נקרא **תכנות פרוצדורלי** – לא נרחיב עליו כי רובכם כבר תכנתתם באופן פרוצדורלי (בין אם קראתם לזה כך או לא). הסימולטור שלנו, אם כן, תוכנת במודל "היברידי": הליבה הייתה פרוצדורלית, והיא מצידה נשענה על עצמים ועל אסטרטגיות פולימורפיות שהן **Object Oriented Design** בהתגלמותו.

גם מי מכם שייחשפו בהמשך לתכנות **פונקציונלי**, ייזכרו בלמדות שכתבו כאן וימצאו בקוד נגיעות קלות גם של הפרדיגמה הזו.

כמו פטיש הבית המצוי, תכנות מונחה עצמים הוא כלי חשוב בארגז הכלים שלנו. ישנן משימות או חלקים בתכנה שאין הכרח לעשות עם פטיש, כמו הברגת ברגים, גזירת ניירות, וכו'. בדומה לפטיש, אם משתמשים בו למקרה הנכון אבל בצורה לא נכונה, נוצרות בעיות. בדומה לפטיש, אם עושים בו שימוש זהיר ומוצלח בשביל להבריג בורג זה עדיין יהיה יותר טרחה משניתן היה אחרת. אף על פי כן, תכנות מונחה עצמים הוא כלי חזק ונח, שבניגוד לפטיש מתאים לספקטרום רחב מאוד של משימות, ובנוסף משחק יפה עם כמות מפתיעה של כלים שונים ומשלימים – למשל תכנות פרוצדורלי כפי שראינו כאן. אז תשמרו את הראש פתוח בהמשך לשילוב של כלים מעולמות שונים, כמו מונחה עצמים, פרוצדורלי, פונקציונלי, מונחה אירועים (event-driven), מונחה מידע (data oriented) ואחרים.