תכנות מונחה עצמים – תרגיל 4

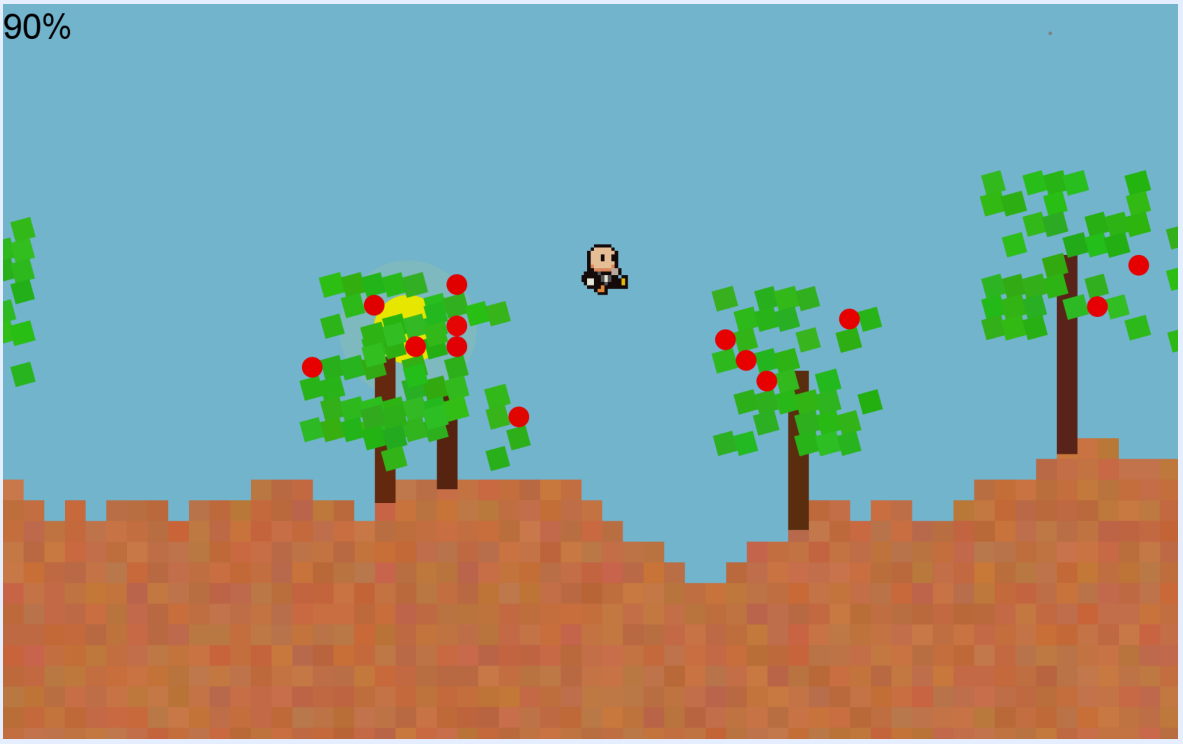
PEPSE: Precise Environmental Procedural Simulator Extraordinaire

תאריך ההגשה: 17.03.24 בשעה 23:55

תרגיל זה מומלץ להגיש בזוגות.

0. הקדמה

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 אווטאר 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

0.1 מטרות התרגיל

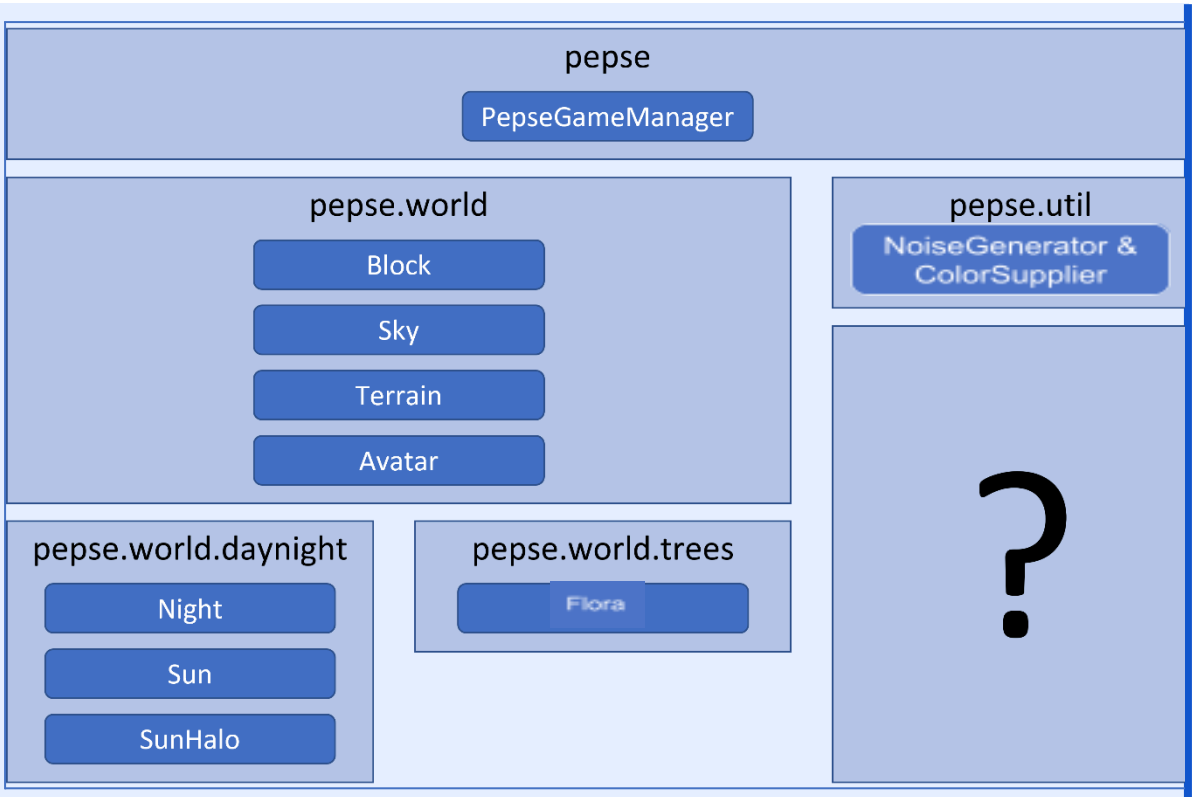
0.1.3. פיתוח הסימולציה הוא תירגול מצוין לתכנות באופי מעט אחר, שבו ריבוי האסטרטגיות הופך את היצירה של עצם למורכבת, עד שלא נוכל לדחוף עוד את הכל למחלקה אחת – נרצה לחלק את מלאכת יצירת העצמים עצמה לקבצים שונים ושיטות שונות. למעשה, הגדרת האסטרטגיות ויצירת העצמים תהווה חלק נכבד מהקוד שלנו, ובפני עצמה לא תיכתב 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 בארגז. 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



על הפרויקט שלכם לכלול את החבילות הבאות:

- `pepse`: החבילה הראשית עם מנהל המשחק (`PepseGameManager`).
- `pepse.util`: 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.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 או מחלקות נוספות.

0.3. תחילת עבודה

ועכשיו, ניגש למלאכה. 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();
}
```

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

כעת נתחיל להגדיר את הדרישות על פי הסדר הבא (בסוגריים רשומה המטרה העיקרית של כל דרישה) :

1. שמיים (חימום)
2. קרקע (חימום)
3. אור וחושך (היכרות עם Transition)
4. שמש (Transitions מורכבים יותר)
5. הילת השמש (שימוש ב-callbacks)
6. הוספת דמות (קלט מהמשתמשים)
7. צמחיה (תרגול חלוקה לשיטות ומחלקות)
   - 7.3 תנודות העלים ברוח (שימוש מורכב יותר ב-callbacks)
   - 7.4 הוספת פירות (כנ"ל)
   - 7.5 שינויים בעצים עם קפיצת הדמות (שימוש ב-callbacks עם עיצוב OOP)
9. עולם אינסופי. **רשות! אין חובת הגשה!** 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 הdimensions של המשחק).
הריצו; 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 עצם זה, למשל עצמים שבשכבה (layer) שלא מוגדרת התנגשות בינה לבין השכבה של עצם זה.

2.1.3. השורה האחרונה אומרת שאם וכאשר המנוע אכן חוסם התנגשות בין עצם זה לאחר, העצם הזה לא אמור להידחף 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 ⅔, ואז ב-x=0 האדמה תכסה בערך את השליש התחתון של המסך. (ב-seed תוכלו להשתמש כדי לאתחל מחולל מספרים אקראיים כפי שנסביר בחלק הרשות.)

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 הריצפה. 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 ש**לפחות** 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 (ב-`initializeGame`) כך שכל רוחב המסך יכוסה באדמה.
הריצו את המשחק - קיבלתם נתח אדמה יפה?

2.2.6. **רשות!** 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.

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 f0.5.

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 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, אם כי זה לא מאוד משנה לאיזה עצם משחק בדיוק. 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().setOpaqueness(float opaqueness)
```

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

```
night.renderer()::setOpaqueness
```

3.6.3. `T initialValue`

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

3.6.4. `T finalValue`

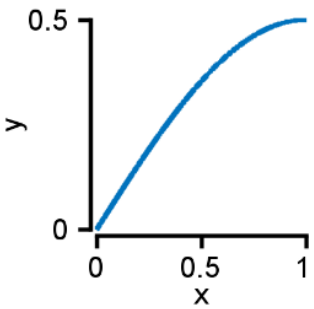
ערך הקצה השני של האטימות הוא חצי:

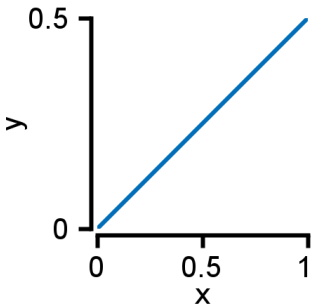
```
private static final Float MIDNIGHT_OPACITY = 0.5f;
```

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:



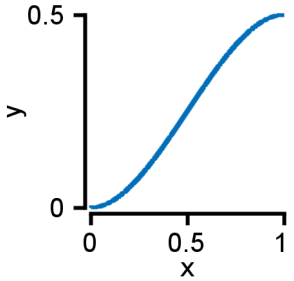


טיפוס הפרמטר, הממשק `Interpolator`, הוא ממשק פונקציונלי שמאפשר לשלוט על צורת הפונקציה הזו. לא ניכנס לממשק הזה לעומק כי לנוחיות המשתמשים המחלקה `Transition`

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 וחוזר חלילה (`TRANSITION_LOOP`), או האם אחרי שמגיעים לערך הסופי צריך לחזור ברוורס אל הערך הראשוני וחוזר חלילה (`TRANSITION_BACK_AND_FORTH`). חשבו אילו מהערכים לעיל מתאימים עבורנו.

3.6.8. `Runnable onReachingFinalValue`

Callback 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 נייצר את המעבר הבא (החליפו את סימני השאלה בערכים המתאימים לפי ההוראות לעיל):

```
new Transition<Float>(
        night, // the game object being changed
        night.renderer()::setOpaqueness,  // 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 ברפרנס), 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. <u>שמש</u>

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-`gameObjectds`, 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, 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 צבע 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`, 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, היא צוברת אנרגיה כאשר היא במנוחה, ומשתמשת באנרגיה כדי לזוז ולקפוץ על המסך. הדמות צריכה להיות מסוגלת לבצע את הפעולות הבאות:

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

תוכלו להיעזר בדמות פשוטה של פלטפורמר שסיפקנו לכם בקובץ Platformer.java.

שימו לב:

- אין להגיש את platformer.java.
- כיוון שהורדנו חלק מהתרגיל המקורי (עולם אינסופי) והפכנו אותו לרשות, אנו לא מצפים שהעולם שלכם יהיה אינסופי, ואם הדמות שלכם תצא מגבולות העולם שייצרתם - זה בסדר גמור שהיא תיפול לנצח.

6.2. בתור התחלה, צרו את המחלקה `Avatar`:

6.2.1. הוסיפו למחלקה Avatar.java את הבנאי הבא:

```
public Avatar(Vector2 pos,
              UserInputListener inputListener,
              ImageReader imageReader)
```

כאשר `pos` הוא המקום במסך עליו אמורה הדמות לעמוד (פינה ימנית למטה). במצב הטבעי, על הדמות שלכם להיות בגובה הקרקע.

6.2.2. בתור `renderable` השתמשו במחלקה `ImageRenderable` וקראו בעזרת ה-`imageReader` את התמונה מהקובץ "idle_0.png" המסופק לכם בתיקיה assets, הורידו אותה ומקמו אותה בתוך תקיית src של הפרוייקט שלכם (כאשר אתם משתמשים באחת התמונות, בקוד שלכם עשו זאת בעזרת נתיב רלטיבי).

6.2.3. הוסיפו מופע של `Avatar` ב-`PepseGameManager`, ובדקו שאכן הדמות נוצרה במקום הנכון, והיא יכולה לזוז ימינה ושמאלה עם החיצים, וכן לקפוץ בעזרת מקש הרווח.

6.3. כעת נוסיף למשחק את המרכיב של צבירת ובזבוז אנרגיה. המשחק מתחיל כך שיש לדמות אנרגיה מקסימלית של 100%. לכל אורך המשחק כמות האנרגיה של הדמות תנוע בין 0 ל-100.

לדמות יהיו 3 מצבי תזוזה:

- מצב idle - הדמות עומדת במקום. במצב זה צוברים אנרגיה.
- מצב run - הדמות רצה ימינה או שמאלה. במצב זה, מבזבזים אנרגיה.
- מצב jump - הדמות קופצת למעלה (ואז נופלת חזרה לקרקע). במצב זה, מבזבזים יותר אנרגיה.

עדכנו את הקוד כך שבכל קריאה ל-`update` כמות האנרגיה תעודכן בהתאם לתזוזת הדמות:

- אם הדמות עומדת ולא זזה, אז היא מקבלת נקודת אנרגיה אחת.
- אם הדמות זזה ימינה או שמאלה על המסך (עם הקרקע או באויר) היא מאבדת חצי נקודת אנרגיה.
- אם הדמות קופצת היא תאבד 10 נק׳ אנרגיה. שימו לב, כשהדמות באוויר ונלחץ על קפיצה, היא לא אמורה לעלות כלפי מעלה, וכן היא לא אמורה לאבד אנרגיה.

שימו לב! אם לדמות אין מספיק אנרגיה אז היא לא תוכל לבצע פעולות כלל ותצטרך לעמוד במקום. למשל אם לדמות יש 5 נק׳ אנרגיה והיא תנסה לקפוץ (שדורש 10 נק׳ אנרגיה) אז היא לא תצליח לקפוץ.

6.4. כעת נוסיף למסך שלכם חיווי שמראה בכל זמן נתון את כמות האנרגיה הנוכחית של הדמות.
כדי להיזכר איך עושים את זה, אתם יכולים לחזור לתרגיל 2 ולהסתכל בקוד שהכנתם עבור הצגת כמות החיים שנשארה במשחק. העיצוב של החלק הזה נתון לבחירתכם, אבל שימו לב לטיפ הבא:
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.

6.5. בהמשך התרגיל נרצה שתהיה לנו אפשרות שאובייקטים אחרים במשחק יוכלו להוסיף אנרגיה לדמות שלנו בגלל אירועים שיקרו במשחק, הוסיפו ל-API מתודה שתאפשר זו.

6.6. 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 בתיקיה assets:

- עבור מצב בו הדמות לא זזה (idle) יש 4 תמונות עם השמות idle_0.png, idle_1.png, idle_2.png, idle_3.png.
- עבור מצב בו הדמות עולה או יורדת (בלי תזוזה לצדדים) נשתמש בתמונות jump_0.png, jump_1.png, jump_2.png, jump_3.png.
- עבור מצב 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.
שימו לב:

- תוכלו להחליף את ה-`Renderable` של העצם בזמן ריצה כאשר הוא עובר בין מצבים באמצעות המתודה `renderer().setRenderable`.
- כדי לשקף את ה-`Renderable` אופקית (בשביל להשתמש באותו `Renderable` בין אם הדמות רצה שמאלה או ימינה), השתמשו ב:
  `renderer().setIsFlippedHorizontally`.

בקבצים שיש באתר סיפקנו לכם קבצים עבור הדמות שיש לה 3 מצבי תזוזה. תוכלו כמובן גם להשתמש בכל דמות אחרת שאתם רוצים, ואפשר למצוא רעיונות כאן.

6.7. בהמשך התרגיל, נרצה שיקרו דברים נוספים במשחק בעקבות כך שהדמות קפצה, ולכן בשלב האחרון של יצירת הדמות, אנחנו רוצים לעשות הכנה עיצובית לכך.

חשבו כבר עכשיו איך תוכלו לעצב את הקוד של המחלקה `Avatar` כך שהיא תוכל לעדכן אובייקטים אחרים במשחק בכך שהתרחשה קפיצה. עיצוב טוב יהיה כזה שבהמשך התרגיל לא תצטרכו לעשות עוד שינויים במחלקה `Avatar` אלא רק במחלקות החדשות שתוסיפו.

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.

<u>הכוונה עיצובית</u>: החופש העיצובי בתכנון הספרייה הזו זה 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.

גרף התלויות הבא: עדיף על 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, נשאיר 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.groundHeightAt.

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 יווצר בדיליי 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).

7.6. שינויים בעצים עם קפיצת הדמות

לאחר שיצרנו עצים עם עלים ש"זזים ברוח" נוסיף עוד אפקט נחמד למשחק שלנו, החלקים השונים של העץ יגיבו בשינוי קטן בכל פעם שהדמות תקפוץ. זוכרים שבחלק של יצירת הדמות ביקשנו מכם לחשוב על עיצוב מתאים לכך שהדמות תרצה לעדכן אובייקטים אחרים בכך שהיא קפצה? אז עכשיו הגיע הזמן להשתמש בהכנה הזו ויש לכם כבר את כל הידע והכלים לדעת איך לעשות זאת:

7.6.1. העלים יסתובבו בכל פעם שדמות ה-`avatar` שלנו תקפוץ. הוסיפו לקוד תמיכה בכך שכל עלה יסתובב ב°90 סביב עצמו בכל פעם שהדמות שלנו תקפוץ.
7.6.2. הפירות ישנו צבע. הוסיפו לקוד תמיכה בכך שכל פרי ישנה את צבעו כאשר הדמות קופצת. מספר הצבעים השונים של הפירות והצורה בה נבחר הצבע השונה נתון לבחירתכם.
7.6.3. גזעי העצים ישנו את הגוון החום שלהם לגוון חום רנדומלי אחר. הוסיפו לקוד תמיכה בכך שכל גזע עץ ישנה את צבעו באופן רנדומי לגוון אחר של חום, באופן דומה לאיך שנבחר הגוון המקורי של הגזע.

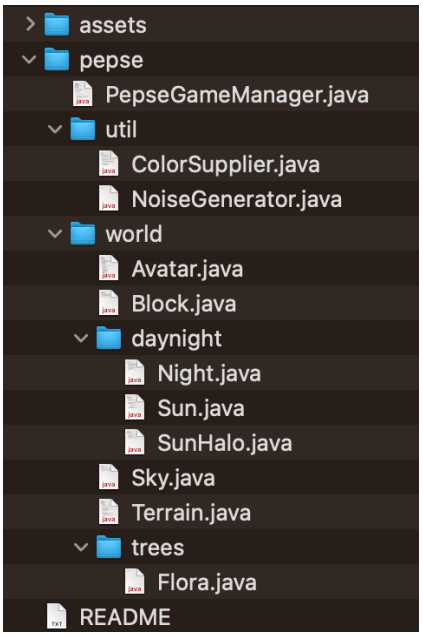
7.7. סיכום

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. הוראות הגשה

8.1. הגישו את התרגיל כקובץ jar/tar/zip בשם **ex4** (והסיומת בהתאם).

בתוך קובץ זה, ימצאו החבילות והקבצים במבנה הבא (שימו לב שמותר לכם **להוסיף** חבילות ומחלקות כרצונכם)



8.2. קובץ ה-README:

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

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

8.2.3. השורה השלישית ריקה.

8.2.4. **החל מהשורה הרביעית ענו בקובץ על השאלות הבאות**:
הסבירו על הדרך שבה בחרתם לממש את החבילה trees, התייחסו בתשובתכם לנקודות הבאות:

8.2.4.1. פרטו על המחלקות השונות שיצרת בחבילה.

8.2.4.2. הסבירו על הקשרים בין המחלקות.

8.2.4.3. האם השתמשתם בתבנית עיצוב כלשהי?

בהצלחה!

פרולוג

9. עולם אינסופי - חלק רשות. אין חובת הגשה!

9.1. 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
```

9.2. בהנחה שבקריאה ל-Terrain.createInRange ול-Flora.createInRange לא הגדרתם minx=-∞ ו-maxX=∞ , 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.

9.3. 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 שמיוצר 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.

10. דיון על שילוב פרדיגמות תכנות

אולי שמתם לב שלא השתמשנו הפעם בממשקים. למעשה, רוב הקוד 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 `Transition`, `ScheduledTask` 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 היחידי של המחלקה שלו, ושאין לו מצב (שדות) מעניין. זה היה המקרה עם החלקה `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.