PEPSE: Precise Environmental Procedural Simulator Extraordinaire

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!), העלים נעים ברוח, נופלים בשלכת, וגדלים חזרה.

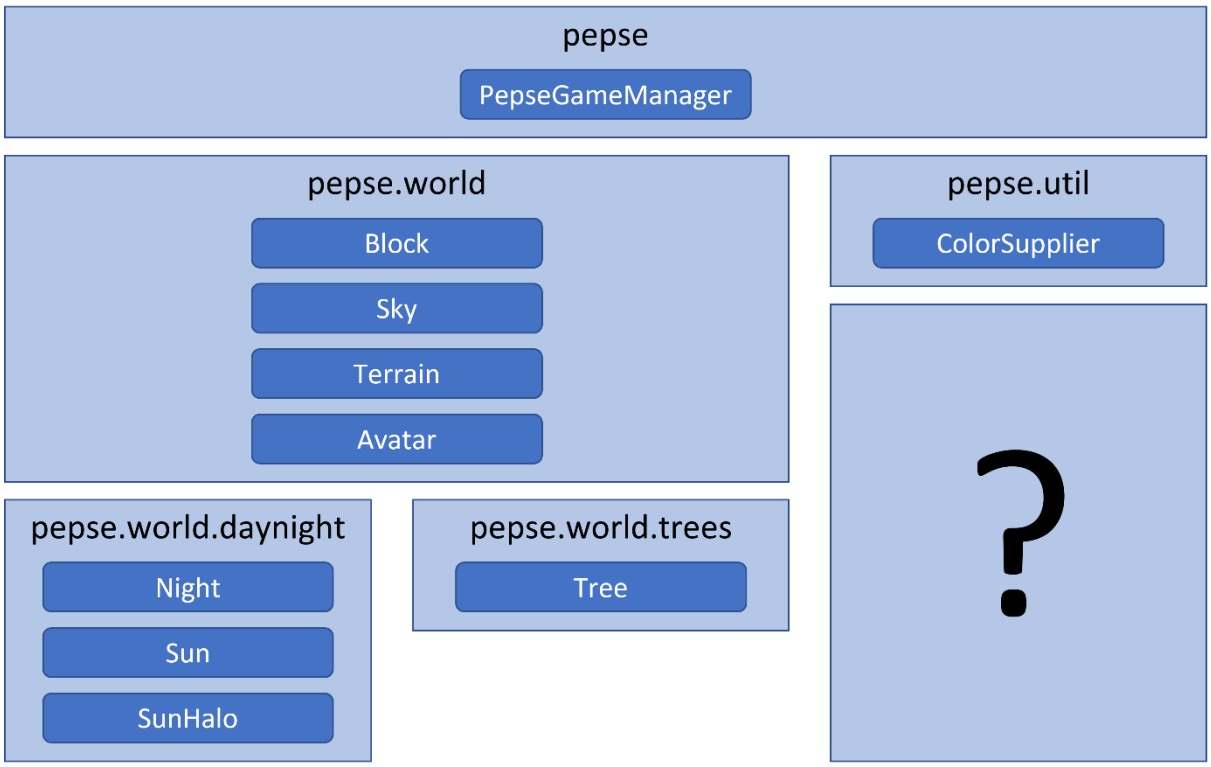
חלקכם תרצו לפתח אותו לאחד מכמה כיוונים מתבקשים: הוספת אלמנט של מים וגשם, פיתוח משחק פלטפורמר, משחק בניה, או משחק הישרדות. פונקציונלית, הסימולציה מאוד מודולרית.

המטרות של התרגיל הן:

1) פיתוח הסימולציה הוא תירגול מצוין לתכנות באופי מעט אחר, שבו ריבוי האסטרטגיות הופך את היצירה של עצם לכל כך מורכבת, 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.
2) עד סוף המדריך, אתם אמורים להרגיש בנח עם נושאים כמו הגדרת למבדות, שימוש ב- method references, ושליחת callbacks.
3) אולי החשוב מכולם: מהכנת הסימולציה תמשיכו ללמוד להנות מתכנות.
4) שימו לב שהתרגיל הנ"ל להגשה בזוגות! יש להגיש בזוג אלא אם כן קיבלתם אישור מהמתרגל האישי שלכם להגיש לבד. בקשה להגשה לבד צריכה להיות מסיבה מוצדקת ולא רק העדפה אישית.

עיצוב ויצירת שלד הפרויקט

על הפרויקט שלכם לכלול את המחלקות והחבילות הבאות:



- pepse: החבילה הראשית עם מנהל המשחק (PepseGameManager).
- pepse.util: מחלקות עזר. כרגע מכילה מחלקה אחת שמייצרת וריאציות של צבעים.
- 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.

הכנת **UML**:

לפני שתתחילו במימוש, קראו את הוראות התרגיל עד הסוף, חישבו מה החבילות, המחלקות והשיטות שתזדקקו להם (אלה שהגדרנו לכם ואלה שתוסיפו בעצמם) והכינו תרשים UML של מבנה הפרויקט שלכם כפי שדמיינתם אותו לפני תחילת העבודה. בסיום העבודה, הכינו תרשים UML מעודכן, והסבירו בקובץ ה-README על ההבדלים בין התכנון המקורי שלכם לזה שיצא בפועל. שימו לב שתצטרכו להגיש את ה**UML** של לפני ואחרי אז שימרו על שתי הגרסאות.

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 ApproximateColor שנשתף איתכם.

לבסוף, הגדירו את PepseGameManager כתת מחלקה 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) שמשTransitions) מורכבים יותר)
5) הילת השמש (שימוש באסטרטגיות)
6) שתילת עצים (תרגול חלוקה לשיטות ומחלקות)
7) תנודות העלים ברוח (שימוש מורכב יותר ב-callbacks)
8) נשירת העלים (כנ"ל)
9) הוספת דמות (קלט מהמשתמשים)

לאחר שנממש את הפונקציונליות הבסיסית, נחזור אחורה ונעדכן את המחלקות ששימשו אותנו ליצירת הקרקע והעצים, כך שיתמכו בעולם אינסופי שנבנה עם התנועה של הדמות. לבסוף, נדון קצת בפרדיגמות השונות ששימשו אותנו.

שמיים

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

```
public static GameObject create(GameObjectCollection gameObjects,
                                Vector2 windowDimensions, skyLayer)
```

השיטה מקבלת את gameObjects בשביל להוסיף אליו את השמיים, את גודל החלון, ואת השכבה בה צריך להוסיף את השמיים, ומחזירה את השמיים שנוצרו, ליתר ביטחון (אם מישהו ירצה לעשות איתם משהו, למרות שבדרישות המינימליות של התרגיל לא נעשה בהם עוד שימוש).

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));
```

שימו לב שאנחנו נותנים לכם את הקוד כדי שלא רק תעתיקו ותדביקו אלא שגם תיזכרו איך לעשות את זה בעצמכם.

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

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

והוסיפו את עצם המשחק בשכבה הנכונה:

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

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 וגמרנו!

הריצו; קיבלתם שמיים בצבע שמיימי כרקע לחלון הריק?

אם תרצו, בשלב זה תוכלו להוסיף לשמיים גם עננים, ציפורים, צלחות מעופפות וכל דבר שעולה בראשכם.

קרקע

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:

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

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.
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.

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?

הגדירו את המחלקה Block.

המחלקה Terrain

אחריות המחלקה:

1) לייצר את כל בלוקי הקרקע הדרושים.
2) 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).

ראשית, נגדיר בנאי אשר שומר את אוסף עצמי המשחק, את השכבה בה יש לשים בלוקים כאלו, ומקבל גם את ממדי

```
public Terrain(GameObjectCollection gameObjects,
               int groundLayer,Vector2 windowDimensions,
               int seed) { … }
```

החלון:

את שני הפרמטרים הראשונים הוא ישמור בשדות.

בנוסף, הוא יכול לאתחל שדה בשם 0groundHeightAtX שמכיל את גובה הקרקע הרצוי ב-x=0. למשל, אם ערכו של השדה יהיה גובה החלון כפול ⅔, אז ב-x=0 האדמה תכסה בערך את השליש התחתון של המסך.

ב-seed תוכלו להשתמש כדי לאתחל מחולל מספרים אקראיים כפי שנסביר בהמשך.

בואו ניגש דווקא לתפקיד השני של המחלקה – להגדיר לעצמה ולעצמים אחרים מה גובה הקרקע הרצוי. נגדיר במחלקה Terrain את השיטה:

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

זו שיטה שרק מחשבת מה אמור להיות גובה הקרקע במיקום אופקי מסוים. חישבו על השיטה כמו פונקציה מתמטית – היא מקבלת x ומחזירה y. מצורפת לכם דוגמה לפונקציה, מדובר בפונקציה קבועה – לכל x, גובה הקרקע הוא כמו גובה הקרקע ב-x=0. במימוש שלכם, 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, ותוכלו למצוא ברשת מימושים רבים עבורו בכל שפת תכנות (בפרט בג'אווה). אין בעיה שתשתמשו במימוש שמצאתם באינטרנט, אך שימו לב כי בחלק מהמימושים האקראיות מתחילה מאיזשהו seed, ולכן עליכם לוודא כי אתם מעבירים את ה-seed הזה לבנאי של מחולל המספרים האקראיים.

השיטה השנייה של Terrain מייצרת בפועל את כל בלוקי הקרקע בתחום x-ים מסוים, על פי הגבהים שמגדירה groundHeightAt. שמה createInRange:

```
public void createInRange(int minX, int maxX) { … }
```

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".

אבל 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. בתור groundLayer כדאי לשלוח את Layer.STATIC_OBJECTS. קראו ל-createInRange; אתם רואים את הבלוק שיצרתם?

רק עכשיו נממש את createInRange כך שתיצור שטח אדמה שלם על פי הגבהים שמגדירה groundHeightAt.

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

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

למשל בקואורדינטת Y של:

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. קיבלתם נתח אדמה יפה?

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: השימוש כאן במופע של Terrain מעט מנוון. איזה יתרון יש לגישה הזו – למה לא הגדרנו את שיטות המחלקה כסטטיות וחסכנו את המופע המיותר למראה של Terrain?

אור וחושך

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 0.5f.

במחלקה Night הגדירו את השיטה הסטטית create:

```
public static GameObject create(
        GameObjectCollection gameObjects,
        int layer,
        Vector2 windowDimensions,
        float cycleLength)
```

תפקיד השיטה לייצר את המלבן לעיל על פי *windowDimensions*, לשלב אותו בין עצמי המשחק בשכבה *layer*, ולגרום לאטימות שלו להשתנות באופן מעגלי עם זמן מחזור של *cycleLength*(מספר השניות שלוקחת "יממה"). השיטה מחזירה את עצם המשחק שיצרה, למקרה שמישהו ירצה לעשות איתו משהו.

החלק המעניין כאן הוא שינוי השקיפות, אבל קודם נתחיל מכל היתר: צרו בשיטה עצם משחק (מופע ישיר של GameObject) בשם *night* עם התכונות הבאות:

1) ה-Renderable הוא מלבן שחור (עם אטימות ברירת מחדל של 1).

2) מרחב הקואורדינטות שלו הוא זו של המצלמה (gameObject.setCoordinateSpace עם הפרמטר CoordinateSpace.CAMERA_COORDINATES).
3) הוא ממלא לחלוטין את כל החלון.
4) הכניסו את *night* ל-gameObjects בשכבה layer.
5) הגדירו תגית מתאימה (setTag) שתבדיל אותו ממופעים אחרים של המחלקה לצורכי דיבוג.

קראו לשיטה Night.create מ-initializeGame של ה-GameManager. בתור אורך המחזור של יממה השתמשנו ב-30 שניות, אם כי כמובן שכרגע עוד אין לפרמטר הזה ביטוי. עבור הפרמטר layer שלחו את Layer.FOREGROUND. הריצו; קיבלתם מסך שחור?

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:

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*.

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) 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).

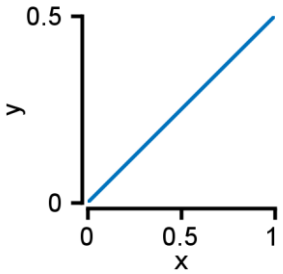
4) T finalValue

ערך הקצה השני של האטימות הוא חצי:

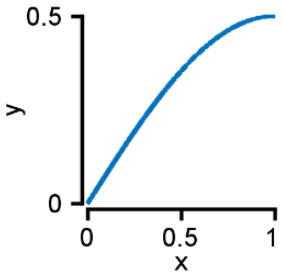
```
private static final Float MIDNIGHT_OPACITY = 0.5f;
```

5) Interpolator<T> interpolator

יש הרבה (אינסוף!) דרכים לעבור מ-0 ל-f0.5 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:



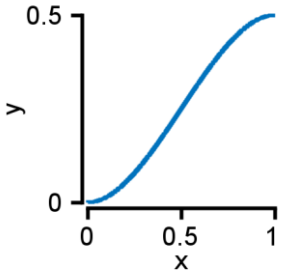
אולי פולינום מדרגה גבוהה יותר? אולי סינוסידיאלית:



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-f0.5 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.

6) float transitionTime
הזמן שלוקח לעבור מהערך initialValue (כלומר 0) ל-finalValue (כלומר f0.5). חשבו מה הערך המתאים כאן.

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.

8) Runnable onReachingFinalValue
Callback שיש להריץ כאשר מגיעים לערך הסופי. לא רלוונטי במקרה הספציפי שלנו, אז נשלח null.

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
        ???,  // transtion 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.

שמש

השמש תיוצר במחלקה Sun, עם השיטה create:

```
public static GameObject create(
        GameObjectCollection gameObjects,
        int layer,
        Vector2 windowDimensions,
        float cycleLength)
```

שמחזירה את השמש, אחרי שיצרה אותה והוסיפה אותה ל-gameObjects.

גם השמש יכולה להיות מופע ישיר של GameObject. בתור התחלה, צרו עיגול צהוב סטטי מעל הקרקע (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).

קיראו לשיטה משיטת האתחול של המשחק, וודאו שיש לכם שמש (סטטית). איזו שכבה לשלוח ל-Sun.create? אם אתם מייצרים את השמש אחרי שייצרתם את השמיים, אפשר לשלוח שוב את Layer.BACKGROUND. אחרת, אפשר לשלוח את 1+Layer.BACKGROUND, שיבטיח שבלי קשר לסדר היצירה השמש תרונדר אחרי (ולכן על גבי) השמיים.

עכשיו נגרום לשמש לנוע בשמיים במסלול מעגלי. בסוף השיטה Sun.create, צריך ליצור מופע חדש של Transition (מעבר). מעבר של מה? חישבו על הווקטור שמתחיל ממרכז המסך ומצביע אל מיקום השמש. המעבר יהיה על פני זווית הווקטור הזה ביחס לווקטור Vector2.UP. חשבו על הערכים המתאים לערך הזווית הראשוני, הערך הסופי, סוג המעבר (כלומר איבר של `TransitionType`) וזמן המחזור. בתור אינטרפולציה, השתמשו באינטרפולציה לינארית פשוטה, עם Transition.LINEAR_INTERPOLATOR_FLOAT.

אבל מה המעבר צריך לעשות עם כל הזויות בתחום הזה? הוא שולח אותם ל-callback שמתקבלת בפרמטר השני של הבנאי של Transition.

ה-callback היא מסוג Consumer<Float>, כלומר הפונקציה צריכה לקבל את ערך הזווית ולא להחזיר כלום.

למדא פשוטה, שנשלחת לבנאי של Transition, יכולה לעשות כאן את העבודה. הלמדא מקבלת את הזווית בשמיים, וצריכה לחשב ממנה איזה וקטור לשלוח לשיטה sun.setCenter (בהנחה ש-sun הוא שם העצם של השמש).

איך לחשב את מרכז השמש הרצוי כפונקציה של הזווית? קצת אריתמטיקה של וקטורים. נשאיר לכם לפענח את הפרטים.

לחילופין, אפשר לממש את ה-callback גם בכל צורה אחרת שנוחה לכם לחישוב מרכז השמש הרצוי. מי שמסתבכים עם השיטות לעיל של 2Vector יכולים לכתוב ידנית טיפונת טריגונומטריה ולממש שיטה כמו הבאה:

```
private static Vector2 calcSunPosition(Vector2 windowDimensions,
                                       float angleInSky)
```

לאחר מכן אפשר לשלוח ל-Transition למדא שמקבלת את הזווית וקוראת לשיטה לעיל. גם במקרה הזה אתם מוזמנים לחשוב על הפרטים בעצמכם.

כעת, אחרי שייצרתם שמש שעושה מסלול מעגלי ברדיוס קבוע, עדכנו את ה-Transition שהגדרתם כך שהמסלול שלה יהיה "אליפטי", או "סגלגל" (לא במובן הגיאומטרי המדויק) – כלומר, שרדיוס הסיבוב של השמש סביב מרכז המסך ישתנה לאורך מסלולה, בחרו מסלול אליפסי כרצונכם ככל שנראה הגיוני ואסתטי (ולא מעגל מושלם).

הילת השמש

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

```
public static GameObject create(
       GameObjectCollection gameObjects,
       int layer,
       GameObject sun,
       Color color)
```

שדומה בהתנהגותה לכל יתר שיטות ה-create שלנו.

כמו במקרה של השמש, נתחיל מלייצר הילה סטטית. גירמו לשיטה לייצר עיגול פשוט, בגודל כרצונכם, בצבע color, במקום כלשהו במסך (למשל בראשית הצירים שהוא הפינה השמאלית-עליונה), והוסיפו אותו ל-gameObjects בשכבה layer. ה-CoordinateSpace שלו הוא כמובן זה של המצלמה (בדומה לשמיים, לשמש, ולמלבן החושך). הגדירו להילה תגית שתייחד אותה.

בשביל לקרוא לשיטה SunHalo.create מ-PepseGameManager.initializeGame נצטרך כמה פרמטרים. 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)
```

איזו שכבה לשלוח לשיטה SunHalo.create?

בואו נעשה סוף סוף סדר בשכבות הרצויות של הסימולציה, מהאחורית לקדמית:

1) שמיים (בשכבה האחורית ביותר, כלומר ה-layer הקטן ביותר)
2) שמש
3) הילת השמש
4) אדמה
5) גזעי עץ
6) עלים
7) עצמי משחק על גבי האדמה
8) מלבן החושך
9) רכיבי UI, ככל שישנם

כלומר השכבה של ההילה צריכה להיות גדולה מזו של השמש (שכבת השמש היא Layer.BACKGROUND או Layer.BACKGROUND + 1) וקטנה מזו של האדמה (Layer.STATIC_OBJECTS).

מכיוון שערכה המספרי של Layer.BACKGROUND קטן ב-100 מזה של Layer.STATIC_OBJECTS, יש לנו 99 שכבות חופשיות ביניהן. אפשר בנוחות לקבוע את שכבת ההילה להיות Layer.BACKGROUND + 10, מה שישאיר די והותר שכבות לעצים.

הריצו; קיבלתם הילה סטטית תקועה בשמיים?

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 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, נשלח 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 (שימו לב שפתרון בשורה אחת, היא איננה דרישה ופתרונות אחרים גם יתקבלו).

אולי תהיתם: האם לא ייתכן שההילה תעקוב אחרי השמש בדיליי, אם בפריים נתון קודם כל תתעדכן ההילה, ורק אז השמש? לא: שכבות אחוריות יותר תמיד מעודכנות לפני קדמיות (ובתוך כל שכבה, על פי סדר ההוספה).

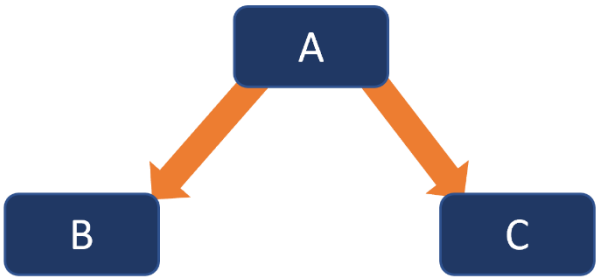
שתילת עצים - עיצוב לבחירתכם (אנא הסבירו בחירות בREADME)

את רוב העיצוב של החבילה הזו נשאיר לכם, למעט הדרישה למחלקה בשם Tree ובה שיטה עם החתימה:

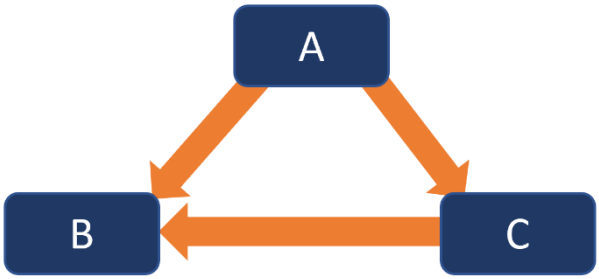
```
public void createInRange(int minX, int maxX) { … }
```

אשר כפי שנובע משמה, תשתול עצים (במיקומים אקראיים/קבועים מראש לבחירתכם) בתחום x-ים מסוים.

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:



תמיד עדיף על גרף התלויות הבא:



למה? (שאלה פתוחה)

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).

הגדירו ב-pepse.world.trees מחלקות לשם שתילת עצים סטטיים (בשלב ראשון): הם כוללים גזע בגובה אקראי וצמרת ריבועית מסביב לקצה הגזע, עם עלים שעוד לא זזים ולא נופלים. בנוסף לעיצוב, נשאיר בידיכם גם את כל פרטי המימוש, אבל כן ניתן כמה הכוונות.

הגזע יכול להיות מורכב מבלוקים (מופעים של Block) עם צבע שהוא וריאנט של:

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

ואילו העלים יכולים להיות בעלי צבע שהוא וריאנט של:

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

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.

כמובן, את העץ צריך לשים בגובה הנכון של הקרקע, ר' השיטה Terrain.groundHeightAt.

תנודות העלים ברוח

כדי לנענע את העלים ברוח, די להוסיף שני מעברים (Transitions) מסוג BACK_AND_FORTH: על זווית העלים, ועל 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. אולי שמתם לב שאם כל העלים נעים יחד בדיוק באותו קצב ותזמון זה נראה רע – אם תסתכלו בחלון תראו שלא כך הם נעים.

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.

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.

נשירת העלים

עלים חיים במעגל אינסופי:

1) לכל עלה מוגדר עם לידתו זמן חיים אקראי.

2) בתום הזמן, העלה נושר.

3) עם תחילת הנשירה, העלה מתחיל לדעוך (fadeOut) עד שלאחר זמן קבוע הוא הופך לשקוף לחלוטין.

4) בשלב זה מוגרל זמן מוות אקראי, שבמהלכו העלה ממשיך להיות בלתי נראה.

5) בתום זמן המוות, העלה נולד מחדש במקומו המקורי (כבר לא שקוף), ומשם חוזרים ל-1, כלומר נקבע לעלה זמן חיים אקראי חדש, ואז זמן מוות חדש וחוזר חלילה.

הוסיפו את הפונקציונליות הזו ליצירת העלים שלכם. במילים אחרות, בזכות ScheduledTask, אפשר להגדיר לעלים את מלוא מעגל החיים הזה כבר ביצירתם, הגם שזמן המחזור משתנה מגלגול לגלגול. אז אמנם לא מספיקה משימה מתוזמנת אחת שמוגדר עבורה repeat, אבל משימה מתוזמנת אחת יכולה לייצר עם פקיעתה את המשימה המתוזמנת הבאה וחוזר חלילה.

כמובן, זה מצריך חלוקה הגיונית וקריאה (readable) של הקוד.

כרגיל, נממש את זה שלב–שלב.

בשלב ראשון, נגרום לכל עלה להתחיל fadeOut אחרי זמן אקראי, כשמשך ה-fadeOut קבוע. לאחר שדעך, העלה פשוט יישאר דעוך. בשביל לגרום לעלה לדעוך היעזרו בשיטה:

```
leaf.renderer().fadeOut(FADEOUT_TIME);
```

בדקו!

בשלב הבא, כשהעלה נושר ומתחיל לדעוך, הוא גם יתחיל ליפול מטה במהירות קבועה (ר' leaf.transform.setVelocityY()). זה השינוי היחיד.

כשעלה מסיים לדעוך, גרמו לו להמתין זמן אקראי ואז להיוולד חזרה במקומו המקורי.

טיפ: היעזרו בהעמסה נוספת של fadeOut, שמקבלת כפרמטר שני Runnable (ממשק פונקציונלי של פונקציה שלא מקבלת כלום ולא מחזירה כלום) שתתבצע כשהדעיכה מסתיימת:

```
leaf.renderer().fadeOut(FADEOUT_TIME, ()->{ } );
```

כל נשירה יפה מאופיינת בתנועה אופקית הלוך ושוב (חשבו על transition).

טיפ: על איזו תכונה של העלה תבצעו Transition למימוש התנועה האופקית? האפשרות המתבקשת היא על מיקומו האופקי של העלה, אבל יהיה לכם קל עוד יותר לעבוד דווקא עם המהירות האופקית שלו (בלומר באמצעות השיטה leaf.transform.setVelocityX()).

עלים שצנחו נשארים על הקרקע

על פניו, פונקציונליות קלה יחסית למימוש: רק צריך לגרום לקרקע ולעלים להתנגש באמצעות מנגנון השכבות:

```
gameObjects().layers().shouldLayersCollide(leafLayer, groundLayer, true);
```

בנוסף כדאי לגרום לעלים לאפס את מהירותם אם הם פוגעים במשהו (כאן נדרשת דריסה של onCollisionEnter). בשביל להפסיק את תנועת העלים האופקית שהוגדרה כ-Transition על העלה, צריך לשמור את אובייקט ה-Transition כשהוא נוצר (נניח ששמו horizontalTransition), ועכשיו לשלוח אותו ל-

```
leaf.removeComponent(horizontalTransition);
```

עדיין, הקאץ' הוא שלחפש את ההתנגשויות החדשות האלה יהיה יקר. יש הרבה עלים והרבה בלוקים של אדמה, ואם העולם שלכם מתרחב שמאלה וימינה יותר מדי תראו צניחה מהירה במספר הפריימים לשנייה שהסימולציה מספיקה לחשב.

טיפ: האם העלים צריכים להיות מסוגלים להתנגש בכל בלוקי האדמה, או שמספיק שיתנגשו בשכבה–שתיים העליונות בשביל לעצור 99% מהעלים הנושרים (או 100% כתלות בתוואי השטח)? האם כל העלים צריכים להיות מסוגלים להתנגש באדמה, או רק אלו שצונחים?

בהמשך, אחרי שנהפוך את העולם שלנו לאינסופי, נרצה להוסיף עצים מפוזרים בכל העולם, ולא רק בחלון המצומצם שאנחנו רואים בתחילת הסימולציה. כאן תתעורר לנו שאלה נוספת, וחשובה עוד יותר: האם יש לכם צורך להמשיך להיאחז בבלוקים שהמצלמה כבר התרחקה מהם? בניגוד לקודמות זו לא שאלה רטורית ותלויה בפונקציונליות שאתם מוסיפים לסימולציה.

הוספת דמות (Avatar)

קל בהרבה משהייתם חושבים! תוכלו גם להיעזר בדמו פשוט של פלטפורמר בתוך ה-zip המקורי של DanoGameLab שהורדתם (בתוך תת-תיקייה codeExamples). יצירת הדמות תעשה באמצעות שיטה סטטית עם החתימה הבאה:

```
public static Avatar create(GameObjectCollection gameObjects,
                            int layer, Vector2 topLeftCorner,
                            UserInputListener inputListener,
                            ImageReader imageReader)
```

במצב הטבעי, על הדמות שלכם להיות בגובה הקרקע. בנוסף, הדמות צריכה להיות מסוגלת לבצע את הפעולות הבאות:

1) לנוע לצדדים (באמצעות החיצים)
2) לקפוץ (באמצעות מקש הרווח) כאשר המהירות האנכית שלה היא 0.
3) לעוף (באמצעות שילוב של מקש הרווח ו-SHIFT). התעופה דומה במהותה לקפיצה, למעט העובדה שהיא לא מחייבת שהמהירות האנכית של הדמות תהיה 0, במקום זאת היא גוזלת לה אנרגיה. ניהול האנרגיה יתנהל באופן הבא: הדמות תאותחל עם רמת אנרגיה 100, בכל צעד-עדכון (קריאה ל-update) שבו הדמות נמצאת במצב תעופה, רמת האנרגיה תרד ב-0.5, ובכל צעד-עדכון שבו הדמות נמצאת במצב מנוחה (על הקרקע, על

צמרת עץ, על ענן אם בחרתם להגדיר כאלה וכו') רמת האנרגיה תעלה ב-0.5. אם רמת האנרגיה התאפסה – הדמות לא תוכל לעוף עד שרמת האנרגיה תעלה בחזרה מעל 0. כלומר אם במהלך תעופה רמת האנרגיה הגיעה ל-0, הדמות תתחיל ליפול וכאשר תנחת על אובייקט כלשהו, רמת האנרגיה שלה תעלה בחזרה. כדי להדגים את ניהול האנרגיה, הוספנו לסרטון ההדגמה חיווי כתוב של רמת האנרגיה של הדמות בכל רגע ,אין צורך שתעשו זאת בעצמכם (את התוספות שיש בסרטון על הוראות התרגיל ,אבל אתם מוזמנים לעשות זאת בכל זאת).

כמה הכוונות ספציפיות:

1) הגדירו GameObject שאתם יכולים לשלוט במהירות האופקית שלו עם קלט-משתמש (ל-300-+ למשל), ושנמצא בשכבת ברירת המחדל (Layer.DEFAULT).
2) לקפיצה: בזמן לחיצה על מקש הרווח, אם מהירות העצם האנכית היא 0 (קרי: הוא לא בקפיצה כבר עכשיו), לאתחל אותה לערך שלילי (נגיד 300-).
3) לצלילה מהשמיים, gameObject.transform.setAccelerationY(). ערך של 500 ייראה בסדר.
4) כדי שהמנוע יטפל בהתנגשויות בעצמו ולא ייתן לדמות לעבור דרך בלוקי אדמה:

```
avatar.physics().preventIntersectionsFromDirection(Vector2.ZERO);
```

5) נסו זאת עם Renderable פשוט (RectangleRenderable למשל). בשביל להוסיף אנימציית ריצה, קפיצה, תעופה ועמידה במקום, תרצו להחליף את ה-Renderable של העצם בזמן ריצה באמצעות renderer.setRenderable(): במצבים הדינמיים, ה-Renderable של הדמות יהיה מסוג AnimationRenderable, וכאשר הדמות עומדת במקום ה-Renderable יכול להיות גם תמונה קבועה (אבל לא חייב להיות). חשבנו שאחת הדמויות כאן חמודה והולכת יפה עם הסגנון הוויזואלי של הסימולציה, אבל אתם יכולים להשתמש בכל דמות אחרת (ובכל סגנון ויזואלי אחר, לצורך העניין).
6) כדי לשקף את ה-Renderable אופקית (בשביל להשתמש באותו Renderable בין אם הדמות מסתכלת שמאלה או ימינה), השתמשו ב-renderer.setIsFlippedHorizontally().

עולם אינסופי

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
```

בהנחה שבקריאה ל-Terrain.createInRange ול-Tree.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=0היה 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.

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=60תמיד 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.

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 בפונקציות 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.

בונוס

הסימולטור הזה יכול להיות בסיס לדברים הרבה יותר מעניינים. אתם מוזמנים לנסות לשפר אותו (בלי לפגוע בדרישות הבסיסיות) כדי לקבל בונוס לציון, אשר ייקבע לפי רמת המורכבות והיצירתיות שלכם (לא יעלה על עשר נקודות). רעיונות לדוגמה:

- הוספה של בלוק מסוג חדש למשחק: מים. בלוק מים מתאפיין באחוז מתוך הבלוק שמכיל מים, ובמצב צבירה. בעזרת המאפיינים הללו וחוקים שמגדירים את הפיסיקה של מים, מים נוזליים מנסים לזרום מטה, מצטברים, מתאדים, מייצרים עננים, שבתורם מתעבים. (יזכה בשתי נקודת בונוס)
לדוגמה, בפריים נתון, בלוק מים שמכיל 60% מים במצב צבירה נוזלי עשוי להפוך את האוויר שמשמאלו לבלוק מים נוזליים נוסף של 1%, את בלוק המים הנוזליים מימינו להגדיל מ-30% ל-31%, את בלוק האויר שמעליו לאדי מים בתפוסת 1%, והבלוק המקורי יופחת בעצמו לתפוסה של 57%. בלוק מים בתפוסת 0% נעלם. מים נוזליים הם בווריאציה של צבע כחול, אדים הם בווריאציה של צבע לבן שקוף למחצה.
- כשקרקע נוצרת, עשויים להופיע עליה לא רק עצים אלא גם חיות שנעות שמאלה וימינה ומקפצות בדפוס משתנה. (יזכה בנקודת בונוס)
- בקרקע חדשה עשויים להיווצר דמויות אנושיות שמרימות ומניחות בלוקים באופן שרירותי, או על פי דפוס מעניין אחר. (יזכה בנקודת בונוס)
- תנאי ניצחון והפסד (למשל: להגדיר משחק עם פרק זמן, ולהגיע לסוף השעון תוך התחמקות ממכשולים נעים או חיות שבדרך). (יזכה בנקודת בונוס)
- כל עלה הוא "sticky note" שבלחיצה עליו אפשר להקליד טקסט שבלחיצה על enter הופך לטקסט של העלה. לחיצה עוקבת על העלה מראה את הטקסט. כדי לסמן שהעלה מכיל טקסט, הוא מסובב ב-45 מעלות. בלחיצת כפתור ימני על העלה הוא נושר ומתחלף בעלה חדש בלי טקסט. (יזכה בשתי נקודות בונוס)
- כשמייצרים עולם על פי seed מסוים, הוא לא רק זהה לזה שנוצר אצל אחר/ת, אלא גם ה-sticky notes משותפים. כלומר תכני כל הפתקים של seed מסוים מסונכרנים בענן. באתר ייעודי תציגו טבלה של seeds ולצידם קטגוריה. למשל, עבור ה-seed המחרוזתי "movies", פתקים באותו עולם משותף נועדו להכיל שמות של סרטים אהובים על משתמשים שונים. כשמשתמש מכניס את ה-seed הזה בתחילת ההרצה, יראה את העולם שבו הפתקים מכילים סרטים אהובים של משתמשים אחרים. בלחיצה על כפתור שמאלי על עלה הוא יכול להוסיף סרט משלו, ובלחיצה על כפתור ימני הוא יכול להביא לנשירה ואתחול של אחד העלים המאוכלסים. (יזכה בשתי נקודות בונוס)
- הדמות בסימולציה מעוצבת בדמותכם או בדמות אהובה עליכם. (יזכה בנקודת בונוס)
- כל רעיון מעניין אחר שתחשבו עליו...

אם בחרתם להגיש גם בונוס, ציינו זאת מפורשות ב-README.

אופי הבדיקה הידנית

הבדיקה ברובה תהיה ידנית ותרצה לראות שהמשחק הוא ללא באגים ורץ חלק, למשל נרצה לראות שהדמות קופצת אז היא לא נופלת מתחת לקרקע, שאנחנו הולכים שמאלה או ימינה ושלא נראה את הקרקע נוצרת תוך כדי, וכדו'. בנוסף הבודקים יסתכלו לראות שכל הדרישות שביקשנו במפורש בתרגיל קיימות (למשל מסלול השמש הוא אליפטי וכדו')

מה צריך להגיש

עליכם להגיש קובץ בשם ex5.jar ובו:

- שני קבצים תמונה עם השמות uml_before.png ו-uml_after.png, הכוללים תרשים UML שהכנתם לפני תחילת העבודה ובסיומה, בהתאמה.
- קובץ טקסט בשם README (ללא סיומת) הכולל את החלקים הבאים, לפי הסדר הזה, כאשר בין כל שני חלקים מפרידות שתי שורות ריקות:
  - שתי השורות הראשונות בקובץ יהיו שמות המשתמשים של השותפים לתרגיל, כל אחד בשורה נפרדת.
  - הסבר על ההבדלים בין שני תרשימי ה-UML.
  - הסבר על האופן שבו הפכתם את העולם שלכם לאינסופי.
  - הסבר על הדרך שבה בחרתם לממש את החבילה trees, ומדוע בחרתם דווקא בדרך זו.
  - הסבר על דילמות או החלטות עיצוביות שקיבלתם במהלך העבודה.
  - אם בחרתם לממש בונוס, הסבירו מה היכולות שהוספתן וכיצד עשיתם זאת.
- תיקייה בשם pepse ובה:
  - קובץ בשם PepseGameManager.java ובו מחלקה באותו שם אשר יורשת מ-GameObject. במחלקה שלכם צריכה להופיע דריסה (override) לשיטה initializeGame וכן שיטה סטטית בשם main אשר מריצה את הסימולציה שלכם (ולא מתבססת על args []String).
  - תיקייה בשם util ובה מחלקות עזר שבחרתם להוסיף (אם בחרתם להוסיף כאלה). אין צורך להגיש את הקובץ **ColorSupplier.java** שסיפקנו לכם.
  - תיקייה בשם world ובה:
    - קובץ בשם Avatar.java ובו מימוש של הדמות שלכם, אשר מסוגלת (לכל הפחות) ללכת, לקפוץ ולעוף, כפי שהוסבר כאן. בתוך המחלקה הממומשת בקובץ, תופיע השיטה הסטטית create וכן שיטות נוספות כראות עיניכם.
    - קובץ בשם Block.java ובו מימוש של בלוק בסיסי, כפי שהוסבר כאן.
    - קובץ בשם Terrain.java ובו מימוש של הקרקע של עולם אינסופי עם צורת שטח מעניינת כלשהי. למחלקה יהיה בנאי עם החתימה שמתוארת כאן, וכן מימוש של השיטות groundHeightAt ו- createInRange לצד שיטות נוספות כראות עיניכם.
    - קובץ בשם Sky.java ובו השיטה הסטטית create המתוארת כאן.
    - תיקייה בשם daynight ובה:
      - קובץ בשם Night.java ובו השיטה הסטטית create המתוארת כאן.
      - קובץ בשם Sun.java ובו השיטה הסטטית create המתוארת כאן. על השמש לנוע במסלול אליפטי בשמיים.
      - קובץ בשם SunHalo.java ובו השיטה הסטטית create המתוארת כאן.
    - תיקייה בשם trees ובה:
      - קובץ בשם Tree.java ובו השיטה createInRange המתוארת כאן. העלים של העץ צריכים לנוע ברוח כפי שמתואר כאן, לנשור כעבור פרק זמן אקראי כפי שמתואר כאן, וכן לצנוח על הקרקע, לדעוך ולחזור למקומם המקורי כפי שמתואר כאן. בדומה לקרקע, גם העצים צריכים לתמוך בעולם אינסופי.
  - כל מחלקה נוספת או שיטה נוספת שתראו לנכון להוסיף. ה-API המתואר בתרגיל הוא מינימלי בלבד.
  - תיקייה בשם assets ובה קבצי מדיה שמשמשים את הסימולציה שלכם.

פירוט של מבנה הקבצים תוכלו למצוא פה: https://danieljannai.github.io/oop_ex5_javadoc/

בהצלחה!