מורה נבוכים - DanoGameLab

סיכום מקוצר ל-DanoGameLab

מה זה Dano Game Lab?

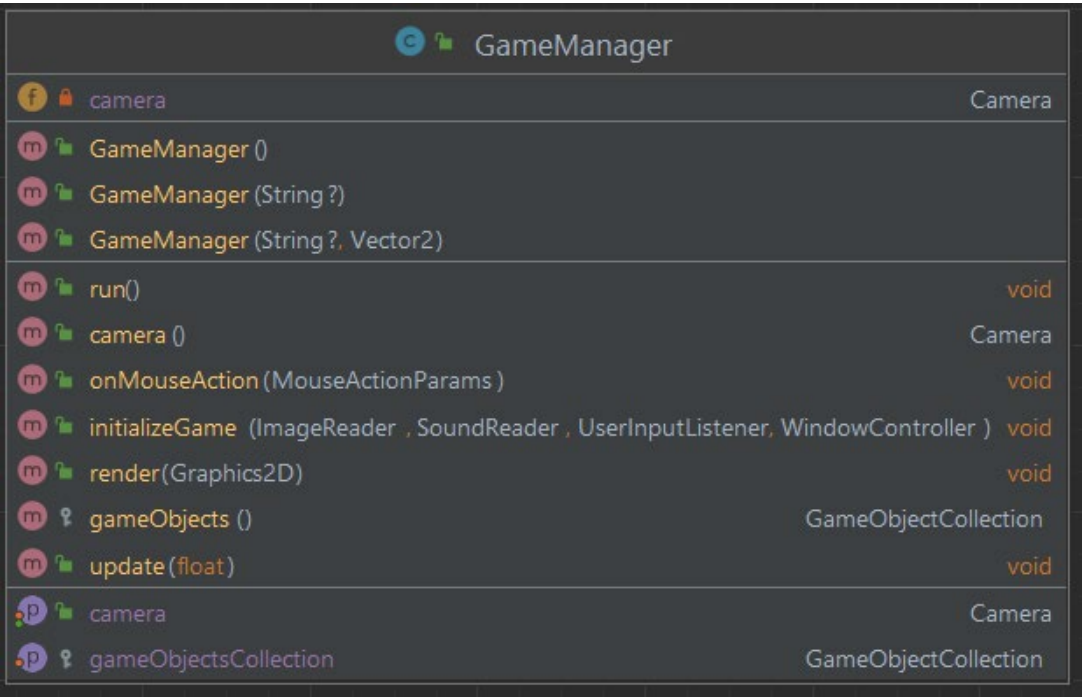
- הספרייה היא מנוע משחק, כלומר ספרייה חיצונית שמטרתה לספק לנו תשתית נוחה (farmework) לכתיבת משחקים.
- בפרט, היא מאפשרת לנו לעשות המון abstraction, ולא להצטרך לטפל בנושאים כמו התנהגות מצלמה, סימלוץ פיזיקה והתנגשויות במשחק, רינדור של המשחק שלנו ועוד...
- כדי לדעת להשתמש ב-(Dano Game Lab) DGL בצורה נכונה ויעילה, עלינו להבין, לפחות באופן כללי, את אופן הפעולה של DGL.

מבנה הספרייה – מבט עילי

GameManager	כשמה כן היא – אחריות המחלקה היא לנהל את יציאת המשחק וריצתו
GameObject	המחלקה הבסיסית ביותר של אובייקט משחק בספרייה
Package Components	חבילה המוקדשת לקומפוננטות, אובייקטים המקושרים ל-GameObjects
Package Collisions	חבילה המוקדשת להתמודדות עם התנגשויות של אובייקטים במשחק
Package Util	חבילה המכילה מחלקות בעלות שימוש כללי, למשל Vector, Counter ועוד
Package gui	חבילה המכילה מחלקות הקשורות לממשק המשתמש של המשחק, למשל רינדור, קבלת קלט מהמשתמש וכו'

מחלקת Game Manager

- אחראית על יצירת המשחק וניהולו באופן מתמשך.
- מתודות חשובות לציון:
  - Public GameManager(): הבנאי של המחלקה, קיימים מימושים שונים המקבלים גם את גודל החלון ואת שם החלון.
  - public void initializeGame: המתודה המאתחלת את המשחק, יוצרת את ה-GameObjectsCollection.
  - public void run(): לאחר יצירת מופע של המחלקה עם הבנאי, קריאה למתודה זו תתחיל את ריצת המשחק.
  - public void update(float deltaTime): ככל הנראה המתודה החשובה ביותר במחלקה, נתעמק בה בשקף הבא.
  - public void render() – רצה על כל GameObject ב-gameObjectCollection וקוראת למתודה render שלו, בפועל גורמת לכל האובייקטים להופיע על המסך.



- השדה החשוב ביותר של המחלקה הוא GameObjectsCollection – זהו מבנה נתונים מתוחכם המאפשר אחסון ואיטרציה על GameObjects, הוא מכיל את כל האובייקטים במשחק.
- בנוסף המחלקה מכילה getter-ים, setter-ים ומתודות נוספות שלא נתעמק בהן.

המתודה GameManager.update

המתודה אחראית על עדכון על ה- GameObjects של המשחק (שנמצאים ב-GameObjectsCollection), ועל עדכון המצלמה, היא מקבלת כ-float את הזמן שעבר מאז העדכון האחרון, בשניות.

נעקוב אחרי ריצה של המתודה:

1. המתודה עוברת על כל אחד מהאובייקטים בתוך ה-GameObjectsCollection, שהוא GameObject
2. לכל GameObject, המתודה קוראת למתודת update שלו (שימו לב שזו לא אותה מתודת update, שכן אין ירושה בין GameManager ו-GameObject).
3. המתודה מעדכנת את המצלמה של המשחק.
4. המתודה מעדכנת את ה- gameObjectCollection - עיקר העדכון הוא שינוי השכבות בתוך מבנה הנתונים במידת הצורך.
5. המתודה קוראת ל-gameObjects.handleCollisions, כלומר מבקשת לבדוק האם היו התנגשויות בין אוביקטים בעדכון האחרון, ואם כן לטפל בהם (למשל עם onCollisonEnter)

```
public void update(float deltaTime) {
    //update all objects and look for collisions.
    for(GameObject obj : gameObjects)
        obj.update(deltaTime);
    if(camera != null)
        camera.update(deltaTime);
    gameObjects.update(deltaTime);
    gameObjects.handleCollisions();
}
```

❗ נציין כהערת שוליים שהמנגנון של update אמור להזכיר לכם תבנית עיצוב שכבר ראיתם - Observer

מחלקת GameObject

- מחלקה זו מייצגת את כל אובייקטי המשחק – בגדול, כל מי שנרצה שהמשחק יעדכן באופן קבוע בכל פריים או שנרצה שיוצג למסך.
- למחלקה בנאי אחד, שמקבל את המיקום של האובייקט, הגודל שלו, וה-Renderable שיהיה לאובייקט.
  - ניתן ליצור GameObject ללא Renderable (כלומר לשלוח null בבנאי) ולקבל אובייקט שהמשחק קורה למתודת update שלו בכל פריים, אך הוא לא באמת מופיע על המסך.
  - שימו לב שמנוע המשחק לא "יודע" בעצמו לקרוא ל-update של ה-GameObject בכל פריים, כדי לעשות זאת, חובה להוסיף את ה-GameObject אל ה-gameObjectCollection של ה-GameManager של המשחק.

מחלקת GameObject - שדות

- שדות של GameObject:
  - renderer – אובייקט המטפל בכל הקשור בהצגת ה-GameObject.
  - transform – אובייקט המטפל בתנועות בסיסיות של ה-GameObject , ומסייע במעבר בין מערכות קואורדינטות של המצלמה של המשחק ובין העולם של המשחק.
  - physics – אובייקט המטפל בפיזיקה והתנועה של האובייקט, בד"כ בהקשרים יותר מורכבים מאשר מה שמטופל ב-transform.
  - tag – מחרוזת המייצגת תגית של האובייקט. מטרתה היא לסייע לכם לזהות אובייקטים לפי התגית שלהם (למשל כדי להבדיל בין 2 אובייקטים שונים במהותם, אשר שייכים לאותה המחלקה. לדוגמא אם יש לנו מחלקת Ball, ואנחנו יוצרים כדורגל וכדורסל, ששניהם מופעים של Ball, נוכל להבדיל בניהם לפי התגית). חשוב לשים לב שאין לתגית שום משמעות מבחינת מנוע המשחק, כך שהשימוש בה הוא עבורכם בלבד.

מחלקת components - GameObject

- שדה נוסף וחשוב של GameObject הוא components.
- זהו מערך של אובייקטים המממשים את הממשק הפונקציונלי Component (ובעברית – רכיב), שמכיל מתודה יחידה: void update(float deltaTime). מתודה זו כנראה נראית לכם מוכרת, והיא מבטיחה שנוכל לרוץ על components שונים בכל פריים של המשחק ולעשות update לכל אחד מהם.
- מחלקות נפוצות המממשות את הממשק: Transition, ScheduledTask

❗ שימו לב כי המימוש הנ"ל של components הוא דוגמא למימוש של תבנית העיצוב "אסטרטגיה".

המתודה GameObject.update

המתודה אחראית על עדכון על ה-GameObjects, היא מקבלת שינוי קטן בזמן: float deltaTime, ומעדכנת מספק דברים בהתאם לזמן שחלף.

נעקוב אחרי ריצה של המתודה:

```
public void update(float deltaTime) {
    transform.update(deltaTime);
    transform.setAccelerationEnabled(true);
    renderer.update(deltaTime);
    if(components != null) {
        for (var component : components)
            component.update(deltaTime);
        components.flushChanges();
    }
}
```

1. המתודה עושה update ל- transform ול- renderer לפי ה-deltaTime, כלומר משנה לפי הצורך את המיקום, מהירות וכו' של האובייקט, ואז מרנדרת את אותם השינויים.
2. המתודה עוברת על כל אחד מה-components שלה, אם ישנם, ולכל אחד היא עושה update עם הזמן שחלף מהעדכון הקודם.
3. לאחר העדכון של כל ה-components, המתודה קוראת ל-flushChanges של מערך ה-components כדי לסנכרן את כל השינויים שקרו בכל אחד מה-components בנפרד.

התנגשויות של GameObjects

- תכונה ייחודית (פחות או יותר) של GameObject היא היכולת של מופעים של המחלקה להתנגש זה עם זה. כדי לחשב התנגשות כזו, מנוע המשחק מחשב האם כל 2 אובייקטים מתנגשים על המסך, כאשר הוא יודע את המיקום והגודל שלהם.
- מתודות הקשורות להתנגשויות הן - onCollisionEnter\Stay\Exit. החתימה של שלושתן זהה מבחינת ערך ההחזרה והארגומנטים, והן נבדלות במועד הקריאה שלהן:
  - onCollisionEnter – תיקרא פעם אחת פר התנגשות, בפריים הראשון בו ההתנגשות קרתה.
  - onCollisionStay – תיקרא בכל פריים, כל עוד ההתנגשות נמשכת, כולל בפריים הראשון.
  - onCollisionExit – תיקרא רק בפריים שהפריים הקודם לו היה הפריים האחרון של ההתנגשות.
- באופן דיפולטיבי, המימוש של שלושתן ריק, והן נתונות למימושכם לפי הצורך.
- שלושת המתודות מקבלות 2 ארגומנטים:
  1. other – זהו ה-GameObject שהתנגש ב-this.
  2. collision – זהו מופע של מחלקה המממשת את הממשק Collision, ובפרט יכיל את המידע על ההתנגשות עצמה – איפה ההתנגשות קרתה, מה הכיוון שלה, גודל החפיפה של האובייקטים, המהירות היחסית בה הם התנגשו ועוד.

התנגשויות של GameObjects - המשך

- נשים לב כי חישוב ההתנגשויות הוא פעולה יקרה מאוד – מנוע המשחק, כל פריים, צריך לבדוק האם כל זוג אובייקטים התנגשו. במשחק מורכב עם הרבה אובייקטים, מהר מאוד נתחיל לקבל התנהגויות לא צפויות.
- כדי להתמודד עם החישובים היקרים ל-DGL יש מספר דרכים ליעל את החישוב, והעיקרית שבהם היא ניהול שכבות. כשאנחנו מוסיפים GameObject ל-gameObjectCollection, אנחנו בוחרים לאיזו שכבה הוא יכנס, ומשם ה-gameObjectCollection לא יחשב התנגשויות של אובייקטים בשכבות שונות (אלא אם צוין אחרת).
- ישנן מספר שכבות בסיסיות שהספרייה מגדירה, כפי שמתואר בטבלה, אך כפי שניתן לראות, בין שכבה לשכבה יש מקום לעוד 99 שכבות לשימושכם, כך שתוכלו ליצור אובייקטים שלא יתנגשו זה בזה במספיק רמות שונות.
- שימו לב שאופן ההצגה של האובייקטים הוא בסדר עולה, כלומר – אם יש לנו 2 אובייקטים באותו הגודל ובאותו המיקום, כאשר אחד מהם בשכבה 2 והשני בשכבה 3, אנו נראה רק את האובייקט בשכבה 3, והאובייקט בשכבה 2 יוסתר על ידו.

שם השכבה	ערך	מהות
Layer.BACKGROUND	-200	מיועדת בעיקר לרקע של המשחק – שמיים, נוף וכו'
Layer.STATIC_OBJECTS	-100	מיועדת לאובייקטים שלא עתידים לזוז במהלך המשחק
Layer.DEFAULT	0	
Layer.FOREGROUND	100	קדמת המשחק – איפה שרוב הדברים יקרו
Layer.UI	200	מיועדת לממשק המשתמש – חיים של השחקן, counter-ים וכד'

התנגשויות של GameObjects - המשך

- ישנן 2 מתודות המאפשרות לסתור את כל האמור מעלה לגבי התנגשויות בין שכבות, והן:

1. gameObjects().layers().shouldLayersCollide:
   - מתודה זו של המחלקה gameObjectCollection מקבלת 2 int-ים של 2 שכבות, ובוליאני.
   - היא מגדירה שהשכבות הנתונות כפרמטרים יתנגשו או לא לפי הבוליאני. באופן זה ניתן לגרום לאובייקטים משכבות שונות להתנגש.
2. shouldCollideWith(GameObject other) –
   1. מתודה זה שייכת למחלקה GameObject,ובהינתן GameObject אחר – other, מגדירה האם האובייקטים יתנגשו או לא.
   2. המתודה מחזירה דיפולטיבית true.
   3. שימו לב שכדי שההתנגשות תתרחש, צריך ששני האובייקטים שאנו רוצים שיתנגשו יהיו באותה השכבה, וששניהם יחזירו ממתודה זו true.

Getters and Setters- GameObject

- למחלקה getters ו- setters רבים, בין השאר:
  - get/set_center – מקבלת / משנה את המרכז של האובייקט.
  - get/set_tag – מקבלת/ משנה את התווית של האובייקט.
  - get/set_dimensions – מקבלת/ משנה את המימדים של האובייקט.
  - get/set_velocity – מקבלת/ משנה את המהירות שלה האובייקט.
  - containsPoint – מחזירה True אם הנקודה שניתנת כארגומנט נמצאת בשטח האובייקט.

GameObjectColletion

- מחלקה זו נמצאת בתוך חבילת collisions, והיא מממשת מבנה נתונים מורכב יחסית שמטרתו ניהול ההתנגשויות במשחק בצורה יעילה.
- באופן דיפולטיבי, מופע יחיד של המחלקה נוצר ב-initializeGame של GameManager, ומשם והלאה כל ניהול ה-GameObject-ים יקרה דרך ה –GameObjectColletion .
- לא ניכנס למעמקי המימוש, אך מה שצריך לדעת הוא שמבנה הנתונים הזה מחלק את האובייקטים לשכבות, ואלא אם צוין אחרת לא מחשב התנגשויות בניהן, אלא רק לכל שכבה בתוך עצמה.
- בכל פריים, מבנה הנתונים עובר על כל השכבות שלו, ומחשב את ההתנגשויות בתוכן.
- שימו לב כי המחלקה מממשת את הממשק Iterable<GameObject>

מתודות - GameObjectColletion

- מתודות חשובות של GameObjectCollection:
  - public void addGameObject(GameObject obj, int layerId) – מכניסה את obj לשכבה layerId.
  - public boolean removeGameObject(GameObject obj, int layerId) – המתודה מנסה להוציא את האובייקט obj מהשכבה layerId. המתודה תחזיר true אם אכן הצליחה, ו-false אם לא הצליחה, בין אם האובייקט כבר עומד לצאת מהשכבה בפריים הנוכחי, ובין אם האובייקט לא נמצא בשכבה.
  - ל-2 המתודות הנ"ל יש מימושים זהים רק שהחתימה שלהם היא ללא הארגומנט של השכבה. בקריאה למתודות אלה המנוע מניח שהשכבה הרצויה היא Layer.DEFAULT.

⚠ אזהרת טריגר! ⚠

- 2 השקפים הבאים עמוסים מאוד, והם מפרטים מה קורה בחישוב של פריים בודד במהלך המשחק, ובפרט מסכמים ומחברים יחד את כל מה שראינו עד כה.
- השקף הבא הוא מעבר מפורט על ה-stack בעת קריאה לפונקציה GameManager.update, השקף שאחריו זהה לו, אבל צבענו את שמות המתודות והאובייקטים כדי לסמל לאיזה מחלקה הם שייכים, נתון לבחירתכם על איזה שקף תרצו לעבור.
- אם הצלחתם לעבור על כל השקף ולהבין מה קורה, סיכוי טוב מאוד שתצליחו לדבג כמעט כל בעיה שאתם חווים ב- DanoGameLab, אז שווה להתאמץ.
- נשימה עמוקה והמשיכו הלאה ☺

מה קורה במהלך ריצת פריים?

- בכל פריים של המשחק, נקראת המתודה GameManager.update, המתודה עוברת על כל ה-GameObject -ים בתוך gameObjects ועושה להם GameObject.update אחד אחרי השני, לפי שכבות, מהשכבה הנמוכה ביותר (שמספרה קטן ביותר) לגבוהה ביותר (שמספרה גדול ביותר):
  - אם בוצעה דריסה של המתודה GameObject.update של ה-GameObject ע"י מחלקה יורשת, ה-GameObject מבצע את מה שעליו לעשות, וקורא בתוך המימוש ל-super.update():
  - בקריאת super.update() מתוך הדריסה של המתודה, או בקריאת update במידה ולא בוצעה דריסה ה-GameObject:
    - קורא למתודה update של ה-transform ושל ה-renderer שלו.
    - במידה ויש לאובייקט components:
      - הוא עובר עליהם אחד אחד וקורא ל-update שלהם.
      - הוא מעדכן את מערך ה-components שלו בהתאם לשינויים שקרו.
- המתודה GameManager.update מעדכנת את המצלמה
- המתודה GameManager.update קוראת ל-GameObjectCollection.update של gameObjects:
  - המופע של GameObjectCollection מעדכן את כל אחת מהשכבות שלו לפי העדכונים שקרו בכל ה-GameObjects, שכבה אחת בכל פעם, מלמטה למעלה.
  - אם שכבות שונות אמורות להתנגש, כאן יטופלו ההתנגשויות בין שכבות.
- המתודה GameManager.update קוראת ל-GameObjectCollection.handleCollisions של gameObjects:
  - המופע של GameObjectCollection יעבור שכבה שכבה, ויטפל בכל ההתנגשויות שקרו, אם קרו:
    - בכל התנגשות, אם המתודה GameObject.shouldCollideWith של 2 הצדדים החזירה true, תיקרא המתודה GameObject.onCollisionEnter\stay\exit של 2 ה-GameObjects המעורבים בהתנגשות, ע"פ הצורך.

מה קורה במהלך ריצת פריים?

- בכל פריים של המשחק, נקראת המתודה **GameManager.update**, המתודה עוברת על כל ה-**GameObject** -ים בתוך **gameObjects** ועושה להם **GameObject.update** אחד אחרי השני, לפי שכבות, מהשכבה הנמוכה ביותר (שמספרה קטן ביותר) לגבוהה ביותר (שמספרה גדול ביותר):
  - אם בוצעה דריסה של המתודה **update** של ה-**GameObject** ע"י **מחלקה יורשת**, ה-**GameObject** מבצע את מה שעליו לעשות, וקורא בתוך המימוש ל-**super.update()**:
  - בקריאת **super.update()** מתוך הדריסה של המתודה, או בקריאת **update** במידה ולא בוצעה דריסה ה-**GameObject**:
    - קורא למתודה update של ה-transform ושל ה-renderer שלו.
    - במידה ויש לאובייקט components:
      - הוא עובר עליהם אחד אחד וקורא ל-update שלהם.
      - הוא מעדכן את מערך ה-components שלו בהתאם לשינויים שקרו.
- המתודה **GameManager.update** מעדכנת את המצלמה
- המתודה **GameManager.update** קוראת ל-**GameObjectCollection.update** של **gameObjects**:
  - המופע של **GameObjectCollection** מעדכן את כל אחת מהשכבות שלו לפי העדכונים שקרו בכל ה-**GameObjects**, שכבה אחת בכל פעם, מלמטה למעלה.
  - אם שכבות שונות אמורות להתנגש, כאן יטופלו ההתנגשויות בין שכבות.
- המתודה **GameManager.update** קוראת ל-**GameObjectCollection.handleCollisions** של **gameObjects**:
  - המופע של **GameObjectCollection** יעבור שכבה שכבה, ויטפל בכל ההתנגשויות שקרו, אם קרו:
    - בכל התנגשות, אם המתודה **GameObjects.shouldCollideWith** של 2 הצדדים החזירה true, תיקרא המתודה **onCollisionEnter\stay\exit** של 2 ה-**GameObjects** המעורבים בהתנגשות, ע"פ הצורך.

מקרא צבעים

מחלקת GameManager.update

מחלקת GameObject

מחלקה יורשת מ-GameObject

מחלקת GameObjectCollection

סיכום ודגשים

- אם שרדתם והבנתם את השקף האחרון, כנראה שאתם מבינים די טוב איך הספרייה DanoGameLab עובדת.
- המטרה העיקרית של המצגת הזו היא לעזור לכם להבין את הספרייה, הן כדי לעשות בה שימוש נכון ויעיל יותר, והן כדי שתנסו ותצליחו, במידת הצורך, לדבג את הקוד שלכם בלי לפחד ללכת לאיבוד במעמקי הספרייה.
- נמליץ לכם, במהלך הדיבוג והעבודה עם הספרייה, לנסות לזהות עקרונות ותבניות עיצוב שראיתם במהלך הקורס, ושימוש בממשקים, ממשקים פונקציונליים ולמבדאות, יש כאלה בשפע ומעבר על המימוש שלהן עשוי להיות מלמד.

נספח 1 – מתודות שימושיות שאולי לא הכרתם

- מחלקת Vector2:
  - public Vector2 rotated(float degreesCounterclockwise) – המתודה מחזירה את הוקטור מסובב ב-degreesCounterclockwise מעלות נגד כיוון השעון.
  - public static Vector2 of(float x, float y) – מאפשר יצירה יותר "יפה" (מבחינת קוד נקי) של וקטורים, במקום עם new, פשוט Vector2 v = Vector2.of(x,y).
  - public String toString() – מאפשרת להמיר וקטור למחרוזת, ובפרט להדפיס וקטורים, מה שעשוי להיות יעיל בזמן דיבוג.
- מחלקת Counter:
  - public void increaseBy(int val) - מאפשרת הגדלה או הקטנה של הערך של ה-Counter ב-val (יכול להיות גם שלילי).

נספח 1 – מתודות שימושיות שאולי לא הכרתם

ממשק Collision – בכל התנגשות מתקבל מופע של מחלקה המממשת את הממשק הזה, ובפרט בכל התנגשות נוכל לקבל ממנו את המידע הבא:

- Vector2 getNormal() – מחזיר את הוקטור הנורמל להתנגשות, כלומר וקטור המצביע מהאובייקט הנוכחי אל האובייקט איתו קרתה ההתנגשות.
- Vector2 getCollisionPoint() - מחזיר את הנקודה בה התרחשה ההתנגשות
- Vector2 getPenetrationArea() – מחזיר את גודל השטח הנמצא בחיתוך של 2 האובייקטים (וקטור של אורך ורוחב)
- Vector2 getRelativeVelocity() – מחזיר וקטור של המהירות היחסית בין האובייקטים שהתנגשו

נספח 2 – בעיות נפוצות שנצפו בתרגילים של סטודנטים

- בעיה - התנגשויות במשחק לא עובדות כמו שצריך - עצמים נופלים אחד דרך השני? מזיזים אחד את השני למרות שהם לא אמורים?
- פתרון –
  - בדקו את היצירה של ה-GameObject – האם הכנסתם אותו ל-gameObjects ולשכבה הנכונה?
  - בדקו את השכבות שלכם – האם האובייקטים שלכם נמצאים באותה השכבה או בשכבות שאמורות להתנגש?
  - בדקו האם דרסתם את shouldCollideWith לא נכון, כך שהיא מחזירה false ומונעת את ההתנגשות.
  - סיכוי טוב, במיוחד בתרגיל 5 או בסוף תרגיל 3 שהדבר קורה בגלל ריבוי התנגשויות במשחק וחוסר היכולת של מנוע המשחק לעמוד בקצב (ראו שקף "מה קורה במהלך ריצת פריים" כדי להבין כמה דברים צריכים לקרות לפחות 30 פעמים בשנייה). הפתרון הוא למנוע התנגשויות לא נחוצות – הפרידו בין שכבות ככל שניתן, כשלא ניתן, נסו להשתמש ב-shouldCollideWith.

נספח 2 – בעיות נפוצות שנצפו בתרגילים של סטודנטים

- בעיה – יצרתם GameObject והוספתם אותו ל-gameObjects, אבל הוא לא מופיע.
- פתרון –
  - בדקו האם יכול להיות שיצרתם gameObjectCollection חדש ואליו הוספתם את האובייקט, עליכם להשתמש רק ב-gameObjects של ה-GameManager.
  - בדקו את השכבות, האם יכול להיות שהוספתם אובייקט לשכבה הלא נכונה כך שהוא מוסתר מאחורי אובייקט אחר?
- בעיה – דברים לא מופיעים / זזים במיקום או באופן בו תיכננתם שהם יהיו, אלא במקום או צורה אחרת על המסך:
- פתרון – שימו לב שב-DanoGameLab מערכת הקואורדינטות היא כזו שהראשית (0,0) נמצא בשמאל למעלה, ומשם כדי "לרדת" מגדילים את ערכי ה-y וכדי "ללכת ימינה" מגדילים את ערכי ה-x.

נספח 2 – בעיות נפוצות שנצפו בתרגילים של סטודנטים

- בעיה – דברים לא מופיעים / זזים במיקום או באופן בו תכננתם שהם יהיו, אלא במקום או צורה אחרת על המסך:
- פתרון – שימו לב שב-DanoGameLab מערכת הקואורדינטות היא כזו שהראשית (0,0) נמצא בשמאל למעלה, ומשם כדי "לרדת" מגדילים את ערכי ה-y וכדי "ללכת ימינה" מגדילים את ערכי ה-x.

נספח 2 – בעיות נפוצות שנצפו בתרגילים של סטודנטים

- בעיה – אני יוצר אובייקטים מלבניים אחד בצמוד לשני (למשל לבנים או בלוקים של אדמה), אך יש מרווחים קטנים (כמה פיקסלים לכל היותר) ולא שווים ביניהם:
- פתרון – שימו לב שכל האובייקטים במשחק נמצאים בקואורדינטות שהן מספרים שלמים, שכן אין דבר כזה "שלושה וחצי פיקסלים". לכן כאשר אתם ממקמים אובייקטים כאלה ברצף, שימו לב ל-type שבו אתם מבצעים את חישובי הקואורדינטות.