



חלליות

רקע למשחק-

<נכתוב כאן רקע למשחק היריות>
במסמך הקרוב ננסה לבנות בסיס למשחק יריות בין גלקטי.

יצירת דמות-

בתור התחלה נבנה בסיס לדמות- ניצור אובייקט משחק חדש (GameObject) שיהווה בסיס לשחקן הראשי ולאויבים. לצורך הדוגמא ניצור אובייקט מסוג קובייה, כמובן שבהמשך יהיה ניתן להחליף את הקובייה בדגם/תמונה של חללית, אך כיוון שעכשיו אנחנו עובדים רק על התשתית של המשחק די לנו באובייקט משחק פרימיטיבי. נקרא לקובייה בשם, לשם הפשטות נקרא לה player. נחליט על גודל שנראה לנו מתאים לשחקן הראשי, ונמקם אותו במקום שהכי קל יהיה לחשב ממנו שהוא הווקטור (0,0,0), כלומר בחלק של ה-position ב-inspector נשים את הערכים 0,0 ו-0 במקום ה-x, y ו-z בהתאמה. במידה ולא מרוצים מהצבע של השחקן הראשי שלנו נוכל להוסיף לו חומר(מומלץ להציץ שוב במסמך 'מה זה unity'). לפני שניכנס לקוד ולהפעלת הדמות נצטרך לשנות את הרקע למשהו שיהיה לנו יותר קל לראות דרכו את המשחק בנתיים. בשביל לשנות את רקע נכנס לאובייקט המצלמה, וב-inspector נבחר clear flags < Solid Color, ומתחת נבחר ב-background את הצבע שנראה לנו הכי מתאים.

הזזת השחקן-

לאחר שסיימנו לבנות את גוף השחקן נתעסק בלבנות את השכל שמנחה אותו- הקוד. ניצור סקריפט חדש לשחקן ונקרא לו בשם זהה. כמו שכבר הזכרנו בשיעורים קודמים מומלץ לשמור את הסקריפטים שלנו בתיקייה ייעודית לסקריפטים. בכדי להתאים את הסקריפט לאובייקט עליו הוא מפעל פשוט נגרור אותו ל-inspector של האובייקט עצמו. נרצה לשחקן שלנו יהיו הדברים הבאים: (1) יכולת הזזה- שנוכל להזיז את השחקן בעזרת לחיצה על החצים במקלדת, או לחיצה על העכבר. (2) מערכת חיים- כלומר כמה חיים יש לשחקן. בד"כ המערכת חיים נעה בין חמש לשלוש, וכל פסילה מורידה לו נקודת חיים אחת מהמשחק, וכאשר השחקן גומר את חייו המשחק נגמר. (3) נקודות- כדי שנוכל לדעת איפה אנחנו עומדים, האם השתפרנו בין משחק למשחק. בשביל להזיז את השחקן שלנו נצטרך להתעמק קצת בפונקציית update() ובשתי מחלקות חדשות: input ו-transform. ניזכר בפונקציה update(), הפונקציה נקראת כל פריים של המשחק כלומר בכל פעם שהמסך מתעדכן אנחנו מבצעים מחדש את המתודה(כמין לולאה שעובדת לכל אורך חיי האובייקט). אם נרצה שהאובייקט שלנו יזוז במסך נצטרך בעצם לעדכן את הפוזיציה שלו(של ה-x או ה-y) בכל פעם. כאן בדיוק נכנסת מחלקת transform: לכל אובייקט משחק(בין אם אובייקט ממשי, מצלמה, תיאורה וכו') יש transform, אשר משמשת לביצוע מניפולציות על האובייקט בין אם שינוי גודל, מיקום או סיבוב האובייקט. למעשה כבר נפגשנו עם המחלקה לפני ב-inspector של האובייקט אך עדיין לא ראינו כיצד ניתן לשנות בזמן ריצת המשחק. אחת המתודות של המחלקה היא translate שהיא מתודה שמקבלת איזשהו וקטור של שלושה ממדים (Vector3) או של שני ממדים (Vector2), ומחברת בין המיקום הנוכחי של האובייקט לווקטור ששלחנו כפרמטר.



למשל אם נרצה לקדם את האובייקט שלנו ביחידה אחת כלפי מעלה כל פריים נצטרך להוסיף בפונקצייה `update()` את קטע הקוד הבא:

```
transform.Translate(new Vector3(0,1, 0));
```

למעשה החברה של unity ידעו כבר שהזזה ביחידה אחת לכיוון מסוים היא שכיחה מאוד ביצירת משחקים, ולכן הם גם יצרו משתנים שמקצרים את כתיבת קוד במעט: למחלקת `Vector3` יש את המשתנה `up` שהוא למעשה הווקטור $(0,1,0)$ והמשתנה `down` שהוא הווקטור $(0,-1,0)$ כלומר ה- y יורד באחד. ובאותו אופן יש את המשתנים `right` ו-`left` שבהם רק ה- x משתנה בהתאמה. הרי שניתן לכתוב את אותו הקוד גם כך: `transform.Translate(Vector3.up);` אם נשמור ונריץ את המשחק נראה שאובייקט שלנו טס כלפי מעלה במהירות עצומה, אם בכלל הצלחנו לראות אותו מרוב שהוא מהיר, במקום לנוע במהירות של יחידה לשנייה למשל, זה משום שהמתודה מתעדכנת במהירות עצומה, פי כמה וכמה ממה שרצינו.

אז מה צריך לעשות כדי שהמהירות תתאים למהירות אחידה של שנייה לדוגמה? גם על זה unity כבר חשבו ויצרו מחלקה מיוחדת שקוראים לה `Time` שאחראית על מידע לגבי הזמן. למחלקה יש משתנה מיוחד שקוראים לו `deltaTime` שהוא משלים את הפריים למהירות של שניה, כלומר אם נכפיל את הווקטור שלנו באותו `deltaTime` הוא יתקדם בהרבה פחות מיחידה אחת לכל פריים, וכך מתי שתעבור שניה הוא יתקדם בעצם יחידה אחת:

```
transform.Translate(Vector3.up * Time.deltaTime);
```

אם נרצה להגדיל את המהירות נוסיף פשוט משתנה `speed` מסוג `float` ונכפיל את הווקטור גם בו:

```
transform.Translate(Vector3.down * _speed * Time.deltaTime);
```

זה המקום להוסיף כי ניתן לראות את כל המשתני העצם הציבוריים של המחלקה ב-`Inspector`, ובכדי לראות משתנים פרטיים יש להוסיף מעל למשנה את התווית `SerializeField` כך:

```
[SerializeField]
private float _speed = 5f;
```

אז הצלחנו להזיז את השחקן שלנו, אך עדיין זה לא קורה תחת השליטה שלנו, מה גם שמהרגע שהוא יוצא מהמסגרת של המשחק הוא נעלם.

נתחיל מהסוף להתחלה- השחקן שלנו נעלם משום הוא לא מפסיק לנוע כלפי מעלה, אנחנו רוצים שאם הוא זז מעבר למסגרת של המשחק שהוא יחזור מהצד השני, כלומר אם הוא עלה הוא יחזור מלמטה, ואם הוא זז ימינה מידי אז שיפיע מצד שמאל, וכנ"ל הפוך.

לשם כך נצטרך להכיר את `transform.position` שהוא למעשה משתנה מסוג ווקטור גם כן שמייצג את המיקום הנוכחי של השחקן. משום ש-`position` הוא ווקטור ניתן גם לקבל משתנה ספציפי ממנו (את ה- x , y או z), וניתן לשנות אותו, את כולו אבל לא חלק ממנו, למשל אי אפשר לשנות רק את האיקס של הווקטור.

עתה נצטרך למצוא את גבולות המסגרת- מאיזה פוזיציה השחקן שלנו יוצא מהמסגרת.

בשביל זה נצטרך להריץ את המשחק וללחוץ על `pause` בדיוק בשלב שכבר לא רואים את השחקן על המגרש.

אנחנו צריכים למתוא את מיקום ה- y של השחקן (כי הוא מתקדם כלפי מעלה) במצבו הנוכחי, ניתן לראות את המיקום של השחקן ב-`Inspector`, ומה שמוצג שם מייצג את הגבול שממנו לא ניתן לראות את השחקן, כלומר הנקודה שממנה נרצה להפוך את המיקום של השחקן לכיוון השני.

נשים לב שהיות והתחלנו מהמיקום $(0,0,0)$, אז גבול המסגרת העליון הוא בדיוק כמו הגבול התחתון של המסגרת רק כפול מינוס אחת, כלומר אם גילינו שמהנקודה $y=7.0f$ (ה-`position` ב-`float`) כבר לא רואים את השחקן, אז מהנקודה $y=-7.0f$ גם השחקן יצא מהתחום רק למטה. לכן אם השחקן שלנו עבר בנקודת ה- y (בערך מוחלט) את $7.0f$ נכפיל את ה- y שלו במינוס אחת (להפוך צדדים), היות ולא ניתן לשנות רק איבר אחד בווקטור אלא את כולם, נשתמש באופציה לקבלת משתנה ספציפי מהווקטור, זה יראה בקוד כך:

```
if (Mathf.Abs(transform.position.y) > 7f)
```

```
transform.position = new Vector3(transform.position.x, transform.position.y*-1, transform.position.z);
```



בשביל למצוא את המסגרת האופקית נשנה את הפונקציית `translate` ל-`vector3.right` במקום `up` ונעשה את אותו התהליך. ועכשיו לשאלת השאלות כיצד ניתן לשלוט בדמות- שהדמות תזוז לאיזה כיוון שאני מכוון אותה לזוז במקום שהיא תנוע רק בקו ישר. לשם כך נצטרך להכיר מחלקה חדשה, מחלקת `input`. מחלקת `input` משמשת בכדי לקרוא קלט מהמשתמש באמצעות `axes` (צירים) עליהם היא עובדת. לכל סוג `axes` יש שם מיוחד משלו, למשל הקלט של תזוזה לכיוון ציר ה-x מהעכבר נקרא `Mouse X`, ואילו מהמקלדת נקרא `Horizontal`. ל-unity יש כ-18 קלטים דיפולטיביים וניתן להוסיף עוד. בשביל לראות את כל הקלטים האפשריים שמגיעים עם `unity` ניכנס ל-`input <-project setting <-edit`, בחלון ה-`input` ניתן לראות את שמות כל ה-`axes` ומאיזה כפתורים הם קולטים, למשל ה-`'Horizontal' axe`, שאחראי לתזוזה אופקית, מקבל קלט מהכפתורים: חץ ימינה (או המקש D במקלדת) - למקרה שהתקדמנו ימינה, וחץ שמאלה (או המקש A במקלדת) - למקרה שאנחנו מתקדמים שמאלה. בכדי שמחלקת `input` תוכל לקבל את הקלטים נצטרך להשתמש במתודה `Input.GetAxis(string AxeName)`, שמקבלת כפרמטר את שם ה-`axe` ומחזיר את הערך אחד אם קיבלנו התקדמות לכיוון החיובי (לדוגמה ב-`Horizontal` אם התקדמנו ימינה), או מינוס אחד אם התקדמנו לכיוון השלילי של הצירים. במידה ולא קיבלנו קלט בכלל הפונקציה מחזירה את הערך 0. אם כך כיצד נוכל לשלב את המידע החדש עם הקוד שלנו כך שההתקדמות של השחקן תהיה בשליטתנו? פשוט ניצור משתנה חדש שמקבל את הערך שתיתן הפונקציה ונשתמש בו בפונקציה `translate` כך שהשחקן יתקדם בהתאם לקלט אותו קיבלנו- אם קיבלנו ערך חיובי, למשל פנינו ימינה, אז השחקן יתקדם יחידת מרחק אחת חיובי מהמיקום הנוכחי שלו, ואם נגיד לא לחצנו על שום כפתור, אז הפונקציה תחזיר 0 כך שאם נחבר את הערך מהפונקציה עם הווקטור של השחקן אז השחקן יישאר במקום. ומבחינת סינטקס זה יראה כך:

```
float horizontal = Input.GetAxis("Horizontal");
float vertical = Input.GetAxis("Vertical");
transform.Translate(new Vector3(horizontal, vertical, 0) * _speed * Time.deltaTime);
```

במקרה לעיל השחקן שלנו יזוז למעלה או למטה בהתאם לחצים או למקשים W,D,S,A במקלדת.

-Prefabs

נח לבנות אובייקט משחק חדש (`GameObject`) בסצנה ע"י הוספת רכיב ועריכת המאפיינים שלו לערכים המתאימים. אולם זה יכול ליצור בעיות כאשר אנחנו מתעסקים עם אובייקטים כמו `NPC` - `non-playe character` כלומר דמות שנשלטת באמצעות המכונה ולא ע"י השחקן, חלק מנוף- עץ למשל או סלעים, או סתם עזרים שיש לשחקן, שהמשותף לכולם שכולם אובייקטים שיכולים להופיע יותר מפעם אחת במהלך המשחק וחולקים מאפיינים דומים. לשכפל את האובייקטים אומנם יצור העתק שלהם, אך השכפול יגרום לכל עותק לעמוד בפני עצמו, כך שאם נרצה לשנות את המבנה של האובייקטים נצטרך לעבור כל העתק בנפרד ולשנות אותו, במקום שיהיה לנו איזשהו אובייקט אב שכל שינוי שיתבצע בו יבטא בכל העתקים שלו ישירות. למרבה המזל ל-unity יש את ה-`asset` "prefab" שמאפשר לאחסן אובייקט משחק שלם עם רכיבים ומאפיינים כ'תבנית' לכל העותקים שלו. בדומה למחלקות וממשקים בשפות תכנות- כל שינוי שיתחולל במחלקת (או ממשק) האב יתבטא גם באינסטנסים שלו. בנוסף ניתן לדרוס (`override`) רכיבים ולשנות מאפיינים של `prefab` אב, דומה מאוד לירושה. אז כיצד יוצרים `prefab`? נהוג ליצור תיקיה ייעודית לכל ה-`prefabs` במשחק. בכדי ליצור אובייקט `prefab` חדש נצטרך לגרור את האובייקט מתוך הסצנה, כלומר מה-`hierarchy view` או מחלון הסצנה (`scene view`), לתיקיה הייעודית בחלון הפרויקט (`project view`). אם האובייקט מופיע בחלון הפרויקט והוא צבוע כחול ב- `hierarchy view` סימן שהפעולה הצליחה, ועכשיו כל פעם שנרצה להוסיף עוד העתק של אותו ה-`prefab` נצטרך פשוט לגרור את האובייקט מחלון הפרויקט לחלון הסצנה או ל-`hierarchy view`.

כמו שהוזכר קודם עריכה של ה-`asset` עצמו, כלומר אותו אובייקט `prefab` שניתן למצוא בחלון הפרויקט, ישתקף על כל העתקים שלו, אך ניתן גם לשנות כל העתק אינדיבידואלית, זה שימושי כאשר אנחנו רוצים ליצור כמה `NPC` דומים אבל בווריאציות שונות כדי שהמשחק יראה יותר ריאליסטי. כדי להבהיר מתי אינסטנס של `prefab` דורס את אובייקט האב שלו הוא מוצג ב-`inspector` עם תווית שם **מודגשת** (וכאשר רכיב חדש נוסף לאובייקט שהוא אינסטנס של `prefab`, כל המאפיינים שלו מודגשים). בהמשך נראה דוגמה לשני `prefabs` מהמשחק שלנו- לייזר ואויבים.



