



חלליות

רקע למשחק-

משחקי יריות שבהם המטרה היא לירות בכמה שיותר אויבים, או בהגה ה"מקצועית" Shoot'em all games הם תת-ג'אנר של משחקי האקשן. אין איזושהי מסכמה כללית על איך אמורים להראות משחקים בסגנון זה, יש המגבילים את התת-ג'אנר למשחקי חלליות או משחקים בהם יש לשחקן את אותן מגבלות תנועה. שורשי המשחקים האלו הם ממשחק החלליות הראשון שיצא באותו המבנה והיה לאב-טיפוס לכל משחקי החלליות- Spacewar! שנוצר ב-1962 ע"י סטיב ראסל, מרטין גרץ ו-וויין ויטאנן. מאוחר יותר הג'אנר התפתח אף יותר עם משחקים כמו space invaders שממנו יצאו משחקים שעד היום מהווים את אבני הבניין למשחקים המודרניים, כמו: Asteroids ו-Galaxian. עצם הסוגה הזאת בסיס להרבה (מאוד) משחקים, ולא רק מאותו הג'אנר נראה במסמך הקרוב כיצד ליצור משחק באותו הסגנון תוך מתן דגש על נושאים מהותיים בבניית משחקים ב-unity.

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 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 לאובייקט. 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 שלשחקן 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 חיי 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 בפונקציית update() את קטע הקוד הבא:

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

למעשה החברה של unity ידעו כבר שהזהר 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 left ו-right 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 שposition 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 (הposition ב-float) כבר לא רואים את השחקן, אז מהנקודה y=-7.0f 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`.
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.

כדי 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. מבחינת 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 עצרים שיש לשחקן, שהמשתתף 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 יתבטא 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 דוגמא לשני prefabs מהמשחק שלנו- לייזר ואויבים.

לייזר-

יריות או לייזר אלה דוגמאות טובות לסוג של prefab היות וכל היריות צריכות להיות זהות, אנחנו נקרא להן ד"י הרבה במהלך המשחק ואין לשחקן שליטה על מנגנון הפעולה שלהן למעט לכוון אותן. ראשית נעמוד על כמה תכונות של היריות: (1) כל היריות מתנהגות באופן זהה עם אותה מטרה. (2) כל ירייה מגיעה רק אם התרחש איזשהו מאורע (השחקן הראשי ירה ברובה למשל). (3) לכל ירייה יש טווח, היריות נעלמות מהמשחק אם הן פגעו במטרה או אם הן יצאו מהמסגרת של המשחק. (4) סוגי לייזרים שונים הם פשוט העתק של אובייקט לייזר ראשוני עם דריסה של כמה רכיבים. בתור התחלה ניצור אובייקט פרימיטיבי שיהווה דוגמת ירי, אישית אני הייתי בוחר בצילינדר כי יש לו דמיון לקליע. נשנה את הגודל של האובייקט כך שיראה בגודל ראליסטי יחסית לשחקן שלנו, נגיד השחקן ב-scale של (1,1,1) נעשה את הלייזר ב-scale של 20% ממנו כלומר (0.2, 0.2, 0.2). ניתן ואף מומלץ להוסיף ללייזר חומר בכדי להבליט את הלייזר ברקע. בדיוק כמו שאמרנו כבר קודם בכדי להפוך את הלייזר שלנו ל-prefab נצטרך לגרור אותו מהחלון בסצנה (או hierarchy view) לתיקייה הייעודית ל-prefabs בחלון הפרויקט.

סקריפט-

עתה נתעסק בלוגיקה של האובייקט. ניצור סקריפט 'Laser' ונחבר אותו לאובייקט שנמצא בחלון הפרויקט (כי אחרת זו דריסה). נרצה שהלייזר שלנו פשוט יתקדם קדימה מהרגע שהתרחש המאורע שקרא לו. איך להזיז אובייקט בקו ישר כבר ראינו כאשר התעסקנו בתזוזה של הדמות הראשית. כל מה שנצטרך זה להשתמש בפונקציה transform.translate עם ווקטור 3 שמתקדם כלפי מעלה כפול הדלתא טיים ומהירות (כי הרי הלייזר אמור להתקדם מהר בהרבה מהשחקן) ומבחינת סינטקס:

```
void Update()
{
    transform.Translate(Vector3.up * Time.deltaTime*_speed);
    ...
}
```

כמובן שאנחנו רוצים לתת ללייזר שלנו טווח, הרי לא נרצה שהוא יתקדם עד אין סוף, כי זה סתם תופס משאבים. לשם כך נצטרך להכיר את המתודה Destroy שהיא משמידה את האובייקט משחק אותו היא מקבלת כפרמטר, במקרה שלנו נרצה להשמיד את this.gameObject במקרה שיצאנו מגבולות המשחק:

```
if(transform.position.y>8.0f)
{
    Destroy(this.gameObject);
}
```

אם נריץ את המשחק בינתיים נראה שאותו לייזר שיצרנו מתקדם כלפי מעלה בקו ישר. עתה אנחנו יכולים למחוק אותו מהסצנה (אבל לא מחלון הפרויקט) היות ולא נצטרך אותו אלא אם כן יתרחש מאורע שקרא לו (למשל לחיצה על רווח במקלדת). בשביל לעשות את זה נצטרך לחזור לסקריפט של השחקן הראשי.

נרצה שהשחקן הראשי יפעיל, או יותר נכון ייצור אינסטנס של הלייזר שלנו כאשר הוא מקבל קלט מהמקלדת למשל רווח. כבר נפגשנו עם מחלקת input שמתעסקת עם קלטים מהמשתמש ובמיוחד במתודה "GetAxes" שמחזירה ערך בהתאם להתקדמות לכיוון החיובי או השלילי של הצירים, כרגע נשתמש במתודה אחרת של המחלקה - 'GetKeyDown' שמקבלת כפרמטר שם של מקש (כמחזרות) ומחזירה ערך בוליאני אם הקשנו על אותו מקש או לא. במילים אחרות נרצה ליצור תנאי: אם קיבלנו ערך חיובי מהפונקציה (כלומר לחצנו על המקש) אז שיוצר אינסטנס של לייזר.

בשביל ליצור אינסטנס של prefab נוכל להשתמש במתודה instantiate(gameObject, transform, Quaternion rotation) הסבר: המתודה מקבלת כפרמטר איזשהו אובייקט משחק כלשהו, vector3 עם המיקום של האובייקט, ואיזשהו וקטור לסיבוב האובייקט אם נרצה ליצור אותו עם סיבוב כלשהו. לרוב לא נרצה שהאובייקט שאותו אנחנו מפעילים יגיע מסיבוב, לכן נוכל להשתמש במשתנה Quaternion.identity שיוצר את האובייקט החדש באותה זווית סיבובית של האובייקט שקרא לו, כלומר



הלייזר שלנו יהיה באותו כיוון כמו השחקן הראשי.

מבחינת מיקום נרצה שהלייזר יתחיל קצת מעל לשחקן אבל באותו קו שלו, לשם כך נצטרך להשתמש במיקום של השחקן ולשנות רק את הנקודה y של השחקן כך שתהיה גבוהה יותר במעט:

```
Vector3 laser_position = new Vector3(transform.position.x , transform.position.y+1, 0);
```

בשביל לשלוח אובייקט מסוג לייזר כפרמטר נצטרך לשמור gameObject מסוג לייזר כמשתנה עצם של המחלקה.

זה המקום להזכיר שמשתני עצם של המחלקה שאנחנו יוצרים כפרטיים אנחנו עדיין יכולים לראות ב-inspector אם הגדרנו מעליהם [SerializeField]. עתה ניצור אובייקט כזה, לצורך הפשטות נקרא לו laser ונחזור למנוע הגרפי.

נשים לב שבחלון ה-inspector של השחקן הראשי, מתחת לסקריפט מופיעים המשתני עצם שלו.

בכדי להגדיר שהאובייקט עצם של player הוא מסוג לייזר נצטרך לגרור מחלון הפרויקט את ה-prefab 'לייזר' לאיפה שמופיע המשתנה laser ב-inspector של השחקן הראשי. עתה שהגדרנו את האובייקט אפשר להשתמש בפונקציה instantiate:

```
if(Input.GetKeyDown("space"))
{
    Vector3 laser_position = new Vector3(transform.position.x , transform.position.y+1, 0);
    Instantiate(laser, laser_position, Quaternion.identity);
}
```

תרגיל: נסו לחשוב איך ניתן ליצור דיילי בין ירייה לירייה כך שנצטרך לחכות קצת זמן בין היריות והן לא יתחילו אוטומטית כל פעם שנלחץ 'רוח'.

אויבים והתנגשויות

אז אחרי שיצרנו דמות ראשית ומערכת ירי אנחנו רוצים שהדמות שלנו תוכל לירות על משהו ולא רק באוויר.

אויבים הם גם סוג של prefab- לשחקן אין שליטה עליהם, לרובם יש מאפיינים זהים, וככל הנראה הם מופיעים כמה פעמים במהלך משחק. לרוב האויבים 'יורשים' מאובייקט אב אחד ופשוט משכילים את הנתונים שלו, לכן במשחקים קלאסיים בד"כ נראה קבוצות של שונות אויבים אך עם מערכת פעולה דומה.

תחילה נבנה אב טיפוס לכל האויבים במשחק, ניצור אובייקט משחק פרימיטיבי שישמש כבסיס, היות והשחקן הראשי שבחרנו כדוגמא היה קובייה נראה לי מן הראוי להמשיך באותו קו ונבנה גם את האויב כקובייה.

כדי להבדיל בין השחקן הראשי לאויב כדאי שניתן לו חומר בצבע שונה ונשנה גם את הגודל במקצת.

ניתן לו שם ונגדיר אותו כ-prefab ע"י גרירה לתיקיה הייעודית.

סקריפט

ניצור סקריפט חדש ונחבר אותו לאובייקט 'אויב' שיצרנו **בחלון הפרויקט**, כי אם נחבר לאובייקט שמופיע בסצנה החיבור יחשב 'לדריסה' של המאפיינים של ה-prefab ואנחנו רוצים לשנות את אובייקט האב דווקא כדי שיריש לאינסטנסים שלו.

נרצה שהאויב יקיים את הדברים הבאים: (1) זז כלפי מטה במהירות אחידה. (2) אם הוא הגיע לתחתית העמוד שלא יושמד, אלא יחזור למעלה אך מנקודה אחרת בציר ה-X, כלומר יצוץ באופן רנדומלי מלמעלה. (3) ברגע שאובייקט 'יתנגש' בשחקן או בלייזר הוא יושמד. כבר ראינו כיצד ניתן לגרום לאובייקט לנוע בקו ישר ולכן לא נתעכב על זה יותר מידי, רק יש לזכור שלמחלקת vector3 יש כמשתנה וקטור ייעודי למקרה הספציפי שלנו: vector3.down.

בשביל לגרום לאובייקט שלנו לצוץ מלמעלה בנקודה אחרת על ציר ה-x לאחר שהוא יצא מהמסגרת של המשחק נכיר מחלקה חדשה- מחלקת Random. אומנם ל-c# יש מחלקה ייעודית להגרלת מספרים באופן רנדומלי עם אותו שם, אך החברה של unity פיתחו גם מחלקה ייעודית בכדי להקל על מפתחי המשחקים. למחלקת Random יש את המתודה range(float start, float end) שמחזירה מספר מוגרל בין שני מספרים שהיא מקבלת כפרמטרים. אנחנו נשתמש במתודה כדי לקבל איזושהי נקודת x שממנה יצוץ האובייקט שלנו. איך נעשה את זה? ראשית נבדוק מה הגבולות של המסגרת שלנו לצדדים וכך נדע את טווח המספרים שממנו אנחנו יכולים להגריל. ברמת העיקרון יש לנו כבר את הטווח הזה מהסקריפט של השחקן הראשי, כאשר רצינו לתת לו טווח תזוזה לצדדים. אח"כ ניצור תנאי בפונקציה update: אם השחקן שלנו יצא מגבולות המסגרת על ציר ה-y, אז שנפעיל את המתודה range ונשמור את הערך שקיבלנו במשתנה שייצג את נקודת ה-x הבאה של האובייקט, ובאותו תנאי גם נשנה את ה-position שלו גם לאותה נקודת x וגם את נקודת ה-y כפול מינוס אחד (כי הוא עובר מתחתית המסגרת לראש המסגרת). מבחינת סינטקס:

```
void Update()
```



```
{
    transform.Translate(Vector3.down * _speed * Time.deltaTime);
    if(transform.position.y < -8.0f)
    {
        float randomized = Random.Range(-8.0f, 8.0f);
        transform.position = new Vector3(randomized, 7, 0);
    }
}
```

התנגשויות-

נחזור ל-unity אם נסתכל על חלון ה-inspector של האובייקטים שייצרנו נראה שיש להם box collider שהוא אחראי על היכולת של אובייקט להתנגש במרחב.

אז מה זה בעצם collider? הוא אחראי על התנגשויות של גופים במרחב. ה-colliders הפשוטים (שצורכים פחות זמן עיבוד) הם ה-colliders הפרימיטיביים. ב-3d יש את ה-box collider, shape collider ו-capsule collider. ב-2D יש אפשר להשתמש ב-circleCollider2D ו-boxCollider2D.

ישנם שני סוגים של collisions (התנגשויות): 1) hard surface collision - התנגשות בין עצמים מוחשיים במרחב, למשל כדור שפוגע בקיר, או התנגשות בין שתי מכוניות. 2) trigger collision - שנותן תחושה כאילו קרה איזשהו מאורע, למשל לקבל מטבע, לקחת חפצים מהרצפה, ובמקרה שלנו - לייצר שפוגע באובייקט אויב.

בשביל להפעיל את היכולת להתנגשות של האובייקט שלנו נצטרך להוסיף לו רכיב rigidbody (גוף קשיח). בשביל להוסיף רכיב rigidbody נבחר בחלון ה-inspector של האובייקט שלנו ('אויב') rigidbody <- add component, וב-box collider לסמן את is Trigger. נשים לב שב-rigidbody יש כמה אפשרויות לבחירה, במקרה שלנו אף אחת מהם לא ממש רלוונטי אלינו, אבל נעבור עליהם בקיצור: 1) mass – אחראי על המסה של האובייקט, יותר מסה ההתנגשות יכול לגרום ליותר נזק לגוף המתנגש. 2) drag – משמש כדי להאט את האובייקט, כלומר כמה כוחות מושכים את האובייקט ומונעים ממנו התקדמות, ככל שה-drag גדול יותר ככה קצב ההתקדמות שלו יורד.

3) angular drag – משמש כדי להאט את מהירות הסיבוב של אובייקט, ככל שהוא גבוהה יותר ככה מהירות הסיבוב קטנה יותר. 4) use Gravity – האם מופעל כוח משיכה על האובייקט, כלומר האם הוא ינוע כלפי מטה. אם הוא לא מסומן, כלומר מוגדר כ-false אז האובייקט יתנהג כאילו הוא נע 'בחלל החיצון'.

5) is kinematic – מגדיר את האובייקט האם הוא מושפע מגרומים נוספים, או רק מהסקריפט ואנימציות.

נסי זאת בעצמכם, ובדקו כיצד האובייקט מתנהג בהתאם לאפשרויות השונות.

לאחר שהוספנו את ה-rigidbody ושחקנו קצת באפשרויות השונות שלו עתה נתעסק בהתנגשות עצמה של האובייקט. אנחנו רוצים ברגע שהאובייקט שלנו יתנגש באובייקט אחר יקרה איזשהו מאורע, למשל כשהלייזר או השחקן הראשי פוגע באויב האויב מושמד. ל-unity יש פונקציה מיוחדת בדיוק למקרה הזה: private void OnTriggerEnter(Collider other). המתודה פועלת באופן עצמי, כלומר לא צריך להפעיל אותה בפונקציית update() כי היא מופעלת אוטומטית במקרה של התנגשות עם גוף זר (אם מאפשרים is trigger ב-box collider) ומבצעת את הפקודה שמגדירים לה בפונקציה. הפונקציה מקבלת כפרמטר איזשהו collider אחר, collider יכול להיות כל אובייקט משחק אחר שמתנגש עם האובייקט שלנו ובלבד שנוכל זהות אותו, אבל כיצד נזהה שהאובייקט שהתנגשנו בו הוא לייזר או השחקן? בשביל זה נצטרך להשתמש בתגיות. נחזור ל-inspector של השחקן הראשי. אם נסתכל למעלה, שורה מתחת לשם האובייקט, נראה שמופיע שם tag, התגיות מאפשרות לנו לזהות את שם האובייקט. הן משמשות כמו משתנה name מסוג מחרוזת לאובייקט שלנו. על מנת שנוכל להגדיר לאובייקט שלנו שם ייחודי נבחר איזשהו Tag, במידה ואין תגית שמתאימה לנו, למשל player עבור השחקן הראשי, נוכל להוסיף תגית חדשה ב-Add tag. נוסיף תגיות לכל אחד מהאובייקטים שלנו ונחזור לקוד של האויב. נרצה שבפונקציה OnTriggerEnter יתרחש הדבר הבא: אם התג של collider שנכנס הוא כמו של הלייזר אז שיושמד האויב:

```
if(other.tag=="Laser" )
{
    Destroy(this.gameObject);
}
```

אם נריץ עכשיו את המשחק נראה שכשאנחנו פוגעים באויב הוא נעלם אבל הלייזר ממשיך לנוע קדימה, זה משום שלא הגדרנו לו collider ו-trigger. נעשה את אותו תהליך שעשינו לאויב גם ללייזר, כלומר נוסיף לו rigidbody ונסמן is trigger. ניכנס לקוד של הלייזר ונוסיף לו המתודה OnTriggerEnter כך שאם הוא התנגש באובייקט עם התגית 'אויב' הוא יושמד גם:

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "Enemy")
```



מבוא לפיתוח משחקי מחשב
ד"ר סגל הלוי דוד אראל

```
{
    Destroy(this.gameObject);
}
```

מערכת חיים-

חלק מהמשחק הוא גם לדעת מתי הוא אמור להיגמר. בדר"כ המשחקים בסגנון המשחק שלנו נגמרים כאשר הגענו לסוף המסלול, או כאשר השחקן הראשי מעבד את כל הנקודות חיים שלו.

בשביל ליצור לשחקן הראשי מערכת חיים נצטרך להוסיף לו משתנה עצם חדש. נצטרך להוסיף לאובייקט גם מתודה שמורידה לו מהחיים נקודה אחת, לה נקרא כאשר תהיה התנגשות בין השחקן שלנו לאויב, לצורך העניין נקרא למתודה `public void Damage()`:

```
public void Damage()
{
    life--;
    if(life<1)
    {
        Destroy(this.gameObject);
    }
}
```

במשחק שלנו ספציפית אין צורך להוסיף לשחקן הראשי collider ניתן להשתמש ב-collider של האויב. בשביל להפעיל מתודה של אובייקט משחק מתוך סקריפט של אובייקט אחר אנחנו צריכים לקבל את אותו אובייקט, ל `unity` יש מתודה מיוחדת במיוחד בשביל זה: `transform.GetComponent<GameObject>()`. אנחנו רוצים שבפונקציה `onTriggerEnter` של האויב, מתי שקורת התנגשות בין אויב לשחקן אז תופעל המתודה `Damage()` של השחקן ונהרוג את האויב שלנו. לשם כך נצטרך לבקש אובייקט 'שחקן' והרכיב שלו הרלוונטי לנו (במקרה שלנו הסקריפט נשמר ב-inspector ברכיב "Player") ולשמור אותו במשתנה מסוג שחקן, אח"כ נפעיל את המתודה שלו ונשתמש במתודה `destroy()` כדי להשמיד את האובייקט 'אויב':

```
if(other.tag=="Player")
{
    Player player= other.transform.GetComponent<Player>();
    player.Damage();
    Destroy(this.gameObject);
}
```

– Spawn manager

אז לאחר שיש לנו כבר שחקן שמסוגל לירות ואויב שניתן להרוג השלב הבא שמתבקש הוא להכניס עוד כמה אויבים למשחק. אז איך נעשה את זה? אם נכניס אותם פשוט אחד אחד מחלון הפרויקט אומנם יהיו לנו כמה אויבים במסך, אך כולם יופיעו ישירות על המסך, והשאיפה היא להכניס אותם בסדר מסוים כך שכל פעם יעלו מספר האויבים הפוטנציאליים בקצב אחיד ולא בפעם אחת, מה גם שזאת סתם עבודה קשה להכניס עכשיו עשרים ומעלה אויבים. למעשה אנחנו כבר מכירים שיטה ליצור אינסטנס של prefab במהלך המשחק, אם ניזכר בשלב יצירת הלייזר השתמשנו בפונקציה `Instantiate` בכדי ליצור את היריות, למה שלא נשתמש בה גם כאן? למה שלא ניצור אינסטנסים של אויבים בתדירות של כל חמש שניות למשל? ברמת העיקרון המתזמן הוא לא אובייקט, לא נראה אותו על המסך עם סטופר מזניק תור של אויבים כל אחד בתורו. למזלנו `unity` דאגו גם לזה ונתנו לנו את האפשרות ליצור אובייקט ריק (`Empty object`). כדי ליצור אובייקט ריק, נלחץ על מקש ימני-`create empty`. ניתן לאובייקט שם, בהגה המקצועית נהוג לקרוא למתזמן `spawn manager`, ונחבר לאובייקט שיצרנו סקריפט עם אותו שם.

יצירת `thread` היא בעייתית במקצת במיוחד במשחקים מורכבים, היות והיא דורשת מהאובייקטים להיות מסונכרנים לתרד הראשי, פעולה שצורכת יותר מידי משאבים יחסית למה שהיא מועילה לנו, למרבה המזל ישנן דרכים נוספות שנוכל להשתמש בהן בכדי לקבל את האפקט של התזמון, דרך נוספת היא באמצעות מתודות `coroutines`.



כשאנו קוראים לפונקציה היא רצה עד שהיא משלימה את עצמה, ורק אז מחזירה ערך אם בכלל. למעשה זה אומר שכל פעולה שמתרחשת בפונקציה קורת בתוך פריים אחד של המתודה `update()`. קריאה לפונקציה לא יכולה לשמש בכדי להכיל תהליך של אנימציה או רצף אירועים התלויים בזמן.

מתודות `coroutines` הן מתודות שמאפרות להחזיר ערך 'זמני' מתוך הפונקציה עד הקריאה הבאה למתודה שמשם היא ממשיכה מאותו מקום שבו עצרה בפעם הקודמת. למשל ספירה עד עשר: המתודה תחזיר כל פעם שנקרא לה מספר מאחד עד עשר. בכדי שהמתודה תוכל להחזיר משתנה זמני עליה לקיים שני דברים: (1) המשתנה בחתימת הפונקציה אותו היא מחזירה צריך להיות מסוג `IEnumerator` שהוא כמו משתנה מסוג `iterable` ב-`c++`, כלומר משתנה שמציג את הערך האחרון בו אנחנו נמצאים במתודה שלנו. (2) במקום שאנחנו מחזירים ערך עם `return` אנחנו מחזירים עם `yield return`. כברירת מחדל מתודות `coroutine` נקראות מחדש כל פריים, אך ניתן להשהות את זמן הקריאה שלהן, כך שנקרא להן אחת לפרק זמן מסוים ע"י שנחזיר משתנה `waitForseconds` עם כמה זמן להשהות עד הקריאה הבאה:

```
yield return new WaitForSeconds(float time);
```

חזרה לענייננו: בדיוק כמו שעשינו עם השחקן נצטרך לעשות גם כאן- כדי לייצר אינסטנסיים של אובייקט 'אויב', ניצור משתנה עצם חדש למחלקה `spawnManager` מסוג `GameObject` (לצורך הדוגמה נקרא לו `enemyPrefab`), נגדיר אותו כ-`[SerializeField]` ונגרור את האובייקט 'אויב' מחלון הפרייקט להיכן שמופיע האובייקט שיצרנו ב-`inspector` של ה-`spawnManager`. אח"כ נוסיף את פונקציה `coroutine` שתיצור אויב כל פרק זמן שאותו נבחר ותגריל לו מיקום חדש להתחיל ממנו(על ציר ה-x כי אנחנו מתחילים בראש המסגרת בציר ה-y). בכדי שהמתודה תמשיך עד לסוף המשחק נכניס את האיטרציות של המתודה ללולאה, כך בפעם הבאה שנקרא לה היא תתחיל מתחילת הלולאה, ולא תמשיך מ-`yield return` האחרון עד לסוף המתודה, ואז לא יהיה ניתן להשתמש בה יותר בתזמון רציף:

```
IEnumerator SpawnRoutine()
{
    while (true)
    {
        Vector3 postspawn = new Vector3(Random.Range(-8f, 8f), 7, 0);
        GameObject new_enemy = Instantiate(_enemyPrefabs, postspawn, Quaternion.identity);
        yield return new WaitForSeconds(time);
    }
}
```

עוד לא סיימנו. כדי לקרוא למתודה שעשינו צריך להשתמש במתודה המיוחדת `StartCoroutine` שמפעילה מתודות מסוג `coroutine`, המתודה מקבלת כפרמטר את הפונקציה עצמה אליה היא קוראת (ניתן גם לשלוח לה מחרוזת עם שם המתודה). למתודה `StartCoroutine` נקרא דווקא מהפונקציה `start()` היות ואין צורך לקרוא לה כל פריים, אלא היא קוראת לעצמה בכל פרק זמן החל מהפריים הראשון של המשחק:

```
void Start()
{
    StartCoroutine("SpawnRoutine");// StartCoroutine(SpawnRoutine())is also valid.
}
```



חלק שני

עד עכשיו התעסקנו בבניית שלד למשחק שלנו- תנועה של הדמויות, לוגיקת משחק ועיצוב קל, כדי לדמות איך המשחק שלנו יתנהל. עכשיו נתעסק בחלק ה"אומנותי" יותר של המשחק- עיצוב דמויות ורקעים, UI, תפריט ראשי ועוד. מכאן נוכל לקח את המשחק שלנו לשני כיוונים- או שנמשיך עם אותו כיוון שהתחלנו אותו וניישם משחק תלת-ממדי, שהיתרונות בו ברורים-ממשיך באותו הקוד שהשתמשנו בו קודם(אין שינויים קטנים),מרהיב יותר וראליסטי יותר, אך הוא ישקול בסופו של דבר הרבה יותר. או שנשנה כיוון ונבנה את המשחק שלנו דו-ממדי, שהוא הרבה יותר קל ליישום, שוקל פחות, ומתאים ליותר פלטפורמות.

הבחירה היא בידיים שלכם ושתי הדרכים לגיטימיות, גם ההבדלים מבחינת התהליך לא גדולים במיוחד, אך לשלב הזה עדיף שנתמקד שבמשהו שיותר קל ליישם ולכן אנחנו ממשיכים כדו-ממדי, אבל שוב מי שמעדיף ליישם את המשחק כתלת-ממדי מומלץ לו להמשיך אתנו כי רוב החומר ד"י זהה.

מעבר מ3D ל2D

בשביל להמיר את הדמויות שלנו לדו-ממדי נצטרך assets דו-ממדיים בשביל המשחק שלנו. מומלץ ביותר לחפש אם קיימים assets שעונים לנו על רוב הדרישות בחנות של unity (unity asset store), כמעט בטוח שנמצא שם ערכה שכוללת הכל, החל מדמויות וכלה בעזרים כמו אודיו או אנימציות במיוחד למשחק שלנו.

חשוב לציין שרוב asset בחנות בתשלום, אך יש מגוון ענק של asset בחינם שניתן לייבא למשחק, ואפשר למיין לפי מחירים. במידה ולא מצאנו, או שאנחנו רוצים לייצור בעצמנו אל דאגה גם בזה נטפל. במשחק נצטרך כמה דברים שיחליפו את האובייקטים הפרימיטיביים שהשתמשנו בהם עד כה : 1) תמונת רקע- התמונה לא חייבת להיות בפורמט ספציפי, כל זמן ש-unity יכול לקרוא אותה, לרוב עדיף להשתמש בפורמט png כי הוא שומר על איכות התמונה המקורית. קל למצוא תמונות שמדמות חלל חיצון (בגוגל. 2) דמויות או אובייקטים- התמונות של הדמויות/אובייקטים צריכות להיות בפורמט png ספציפית, **ובלי רקע**, לרוב קשה למצוא תמונות כאלה, לכן נצטרך ליצור כאלה בעצמנו ע"י תוכנת גרפיקה כלשהי, המומלצות הן Photoshop או Krita, אומנם לפוטושופ יש הרבה יותר מדריכים שימושיים והיא עם תמיכה טכנית, אך אישית אני ממליץ על krita משתי סיבות עיקריות : א) היא חינמית- היא תוכנת open source שרצה על כמה מערכות הפעלה (linux בניהן) ואין צורך ברישיון מיוחד כדי לעבוד איתה. ב) יש לה פיצ'רים במיוחד לאנימציה ועיצוב גרפי שמתאימים לבניית משחקים אינדיים (עצמאים). לעניינו: איך נפריד בין התמונה לרקע שלה? ברמת העיקרון אם אנחנו בונים בעצמנו את הדמויות אין לנו צורך בפורמט ספציפי (IPJE,PNG,GIF) ובלבד שהרקע יהיה בצבע שונה מהאובייקט שישמש אותו.

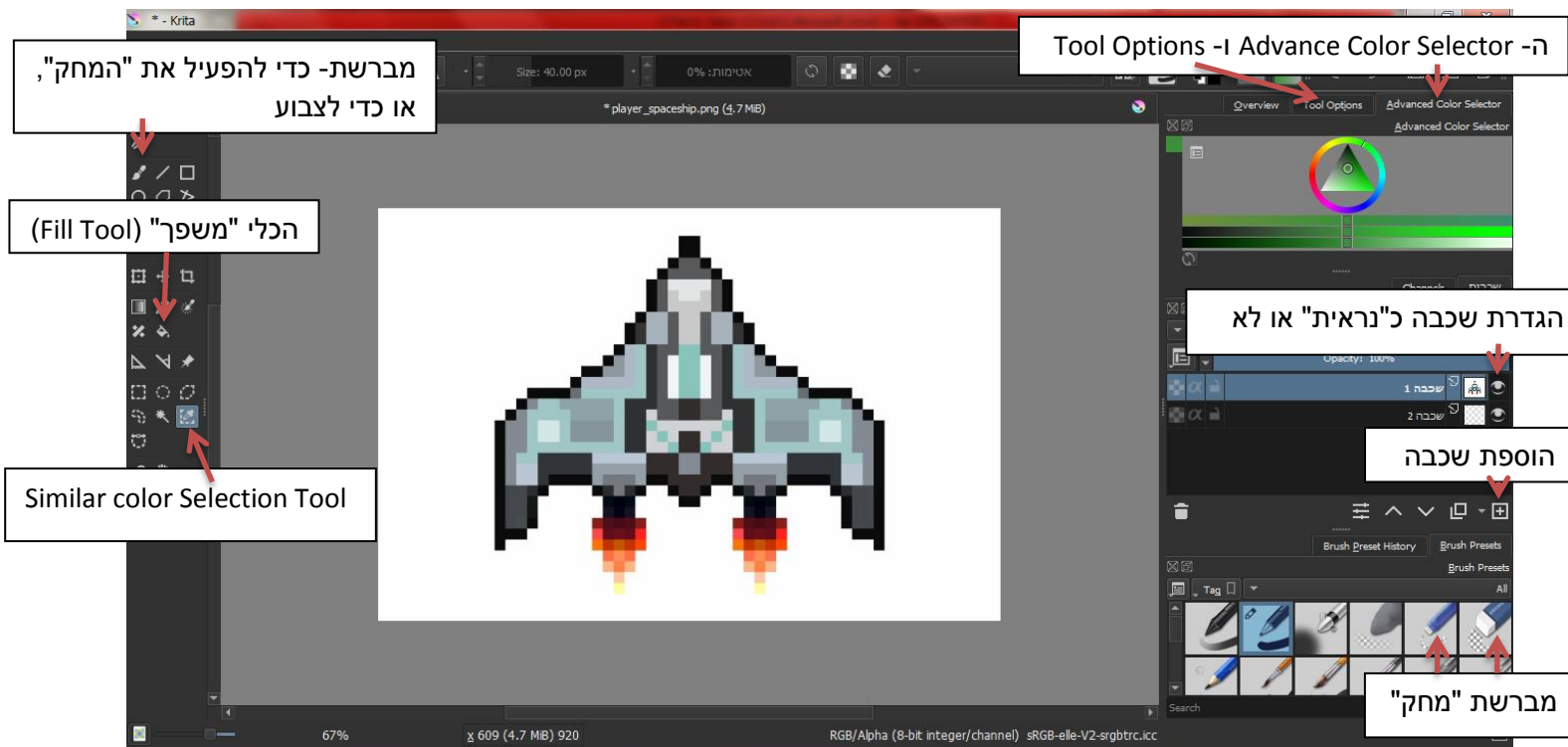
תמונה לדוגמא:



נוריד את התמונה למחשב ונפתח אותה באמצעות קריטה. לקריטה יש כמה כלים שיכולים לעזור לנו להפריד בין הרקע לתמונה עצמה. ראשית נצטרך לדאוג שתהיה לנו שכבה חדשה מתחת לתמונה, לכן ניצור שכבה חדשה ונגרור אותה שתהיה מתחת לשכבה של התמונה ונגדיר אותה כ"לא נראית" (אייקון שדומה למין עין בצד של האייקון של השכבה). נחזור לשכבה של התמונה, נבחר בכלי similar color selection שמסמן בשכבה את אותם המקומות שיש להם את אותו הצבע. צריך לדאוג שה-fuzziness שלו יהיה נמוך (בין 1 ל-5), כך הוא יבדיל כמה שיותר בין גוונים שונים, אפשר לעשות את זה ב-tool option. נלחץ על הרקע עם כלי, הוא אמור לסמן לנו רק את הרקע(זה נראה כמו מלא קווים מסביב לדמות). נמחק את מה שסימנו, או ע"י delete או ע"י מברשת "מחק" ונמחק פשוט אקטיבית את האזור המסומן. כדי לבדוק שלא התווספו לנו "שאריות מהרקע" נסמן את השכבה של התמונה כ"לא נראית", ונעבור לשכבה מתחת ונסמן אותה כ-"נראית", נבחר בכלי משפך ונתן לו צבע בולט ששונה לגמרי מהצבע של הדמות (לרוב ירוק כמו מסך ירוק בסרטים הוליוודיים). ונחזיר את התמונה להיות "נראית", במידה ויש שאריות פשוט נמחק אותם ע"י המברשת "מחק", אם אין שאריות פשוט נסמן את השכבה התחתונה כ"לא נראית" ונשמור



את התמונה כקובץ png ב- File -> save as (או ctrl+shift+s) ובחלון שנפתח נבחר ב- save as type כ-png. זה אמור לשמור לנו את התמונה ללא הרקע שלה. באיור למטה יש לנו תצוגה של הכלים של קריטה שהשתמשנו בהם.



בשביל לעלות לunity (בין את התמונות רקע ובין את התמונה של האובייקטים) פשוט נגרור את האובייקט למנוע הגרפי והוא יופיע בחלון הפרויקט תחת השם המקורי שנתנו לתמונה, וב-inspector שלו נשנה את ה Texture Type ל-Sprite(2D and UI) אח"כ נוכל לגרור את האובייקט לחלון הסצנה ולראות אותו בחלון המשחק גם כן. יכול להיות שלמרות שהגדרנו את התמונה כמו שצריך, וכן ניתן לגרור אותה לחלון הסצנה אך עדיין אנחנו לא רואים את האובייקט בחלון כמו שרצינו. כנראה שהסיבה לכך היא משום שהתמונה לא נמצאת מעל לשכבת הרקע. אם נסתכל על ה inspector של הרקע נראה שיש שם רכיב שקוראים לו additional setting ולו שני אלמנטים: Sorting Layer ו-Order in Layer, הראשון אחראי על איזו שכבה יופיע בה הרקע, מומלץ ליצור שכבה חדשה במיוחד לרקע ע"י <Sorting Layer-Add Sorting Layer. השני משני האלמנטים הוא המיקום של האובייקט יחסית לשכבה בה הוא נמצא כאשר 0 הוא הכי נמוך 1 מעליו וכן הלאה. נגדיר את הרקע כמיקום 0 ואת האובייקט כמיקום 1 או יותר (תלוי כמה אובייקטים הגדרנו), כך נוכל לראות אותו גם בחלון המשחק.

המרת הדמויות ל 2D

אחרי שיש לנו כבר תמונות לייצוג השחקן האויבים והלייזר אנחנו רוצים להחליף בין האובייקטים הפרימיטיביים שמשמשים את הדמויות, לאובייקטים הייעודיים להם. בשביל ליצור את player פשוט נגדיר אובייקט player חדש ונגרור אליו את הסקריפט של player. אם נמחק את האובייקט player הישן שהשתמשנו בו, נגרור את האובייקטים הרלוונטיים ל-inspector של השחקן החדש (במקרה שלנו לייזר), נציין בתג שלו שהוא player ונתחיל לשחק, נראה שהשחקן אומנם יוצר לייזר תלת-ממדי כמו מקודם שמצליח לפגוע 'באויב' הפרימיטיבי שעוד לא החלפנו, אך אם השחקן פוגע בגופו באויב לא מתרחשת "התנגשות" בין האובייקטים, למה זה? זה משום שהשחקן שלנו היה מוגדר כאובייקט תלת-ממדי ועכשיו הוא דו-ממדי, לדו-ממדי ולתלת-ממדי יש רכיבים שונים להתנגשויות, כפי שנראה עוד מעט באובייקטים הפרפאביים (מלשון prefab) שלנו.

נבחר ב-prefab "אויב" ובחלון inspector נסיר ממנו כל אלמנט תלת-ממדי שמגדיר אותו: הקובייה, mesh renderer, rigid body ו- box collider כדי להסיר אותו פשוט נלחץ מקש ימני מעל אותו רכיב-> Remove Component. לאחר שהסרנו את הרכיבים הנ"ל אנחנו רוצים שיהיה ניתן לראות את האובייקטים, לכן נצטרך להוסיף להם רכיב sprite שבו נאחסן את



התמונה שמייצגת אותו. לשם כך נבחר ב-Add Component ובשורת החיפוש נכתוב sprite renderer ונוסיף. נגזור את התמונה של האויב לתוך התפריט של spriten והתמונה תופיע מולנו, חשוב להגדיר את האובייקט בשכבה המתאימה לו. אם נריץ את המשחק התמונה של האויב שהגדרנו מופיעה מולנו, אך עכשיו אפילו הלייזר לא פוגע בה. כדי לפתור את זה נוסיף לאובייקט עוד שני רכיבים: rigidbody2D ו-box collider2D (ב-box collider2D נסמן גם את is Trigger). גם לשחקן נוסיף את הרכיבים האלה וגם לו נסמן את is Trigger כדי לא סיימנו! עכשיו נצטרך לשנות את הסקריפט בהתאם. בסקריפט נשנה את שם המתודה `OnTriggerEnter(Collider other)` ל-`OnTriggerEnter2D(Collider2D other)`. וזהו, אם נריץ עכשיו נראה שהאויב יכול להתנגש בשחקן, אבל משום שלא שינינו את הלייזר להיות דו-ממדי הוא עדיין לא יושפע ממנו. יכול להיות שהתנגשות של האויב בשחקן קורת לפני הזמן, כלומר עוד לפני שהשחקן ממש נוגע באויב האויב מושמד. כנראה שהבעיה שגבולות collider מוגדרות מעבר לגבולות התמונה, כדי לשנות את גבולות colliders של השחקן או האויב צריך ללחוץ על Edit Collider ב-inspector ולשנות אותו לרצוי לנו. עכשיו לאחר שראינו איך לשנות את ה-prefab "אויב" יהיה לנו קל שלנות את הלייזר בהתאם. ניצור אובייקט לייזר חדש עם התמונה של הלייזר, נמחק את האובייקט הישן של הלייזר, נוסיף לו sprite renderer, box collider2D ו-rigidbody2D, נסמן אותו כ-is trigger, נוודא שלא מסומן gravity, ונשנה את הקוד שלו בהתאם. גם כאן- אם האובייקט מתנגש מוקדם מידי נערוך את גבולות collider שלו, ונבדוק כמובן שהמשחק רץ כמו שצריך.

-Collider

לא תמיד נשתמש ב-box collider2D לפעמים השימוש במסגרת ריבועית לא מתאים לגבולות הדמות שלנו, כלומר אם הדמות שלנו עגולה או עם עקומות דווקא נעדיף להשתמש ב-collider מעוגל כמו polygonCollider או circleCollider. להלן רשימה עם הסברים ל-colliders הדו-ממדיים שניתן להשתמש בהם: BoxCollider2D - יוצר גבול מלבני ל-collider. CircleCollider2D - כמו ה-box רק שגבולות collider הן עגולות. EdgeCollider2D - נותן אפשרות לסמן את גבולות collider בצורה אינטראקטיבית ע"י הוספת קדקודים והזזתם. PolygonCollider2D - יוצר את גבולות collider בצורה גסה לפי צורת האובייקט.

-Power ups

אלמנטים שמוסיפים למשחק כדי לשפר את החוויה. למשל במהלך המשחק ניתן לקחת אייטמים שמחזקים את השחקן, או מוסיפים לו חיים וכדו'. אנחנו נראה שני סוגים של power-ups, כדאי מאוד להוסיף סוגים שונים כדי להעצים את חווית המשחק.

-Triple shot

ה-power up הראשון שנעשה הוא ירייה משולשת, כלומר בכל פעם שהשחקן שלנו מקבל את ה-Power up (או יותר נכון מתנגש אתו) הוא יוכל לירות שלוש יריות במקביל במקום שתיים. דבר ראשון שנעשה הוא למצוא איזושהי תמונה שתייצג את ה-power up עם כל השלבים הנלווים. שלב הבא יהיה לבנות את הירייה המשולשת עצמה, נגזור שלושה לייזרים לחלון הסצנה ונסדר אותם יחסית ל-player. ניצור אובייקט ריק בשם triple shot ונגזור את הלייזרים אליו שיהפך להיות אובייקט האב שלהם. נגדיר את האובייקט שיצרנו כ-prefab ונמחק אותו מהיררכיה (hierarchy view). לפני שניכנס לקוד כדי שנבין מה התהליך שנרצה שיקרה עם קבלת ה-power up - אנחנו רוצים שהאובייקט "ילקח" ע"י השחקן, הדבר שמדמה לנו לקיחה, כפי שראינו בפרקים הקודמים, הוא בעצם "התנגשות" של האובייקטים, כלומר מהרגע שהופעל הטריגר של אחד האובייקטים (שקלט שהייתה כאן התנגשות) האובייקט יפעיל איזשהי מתודה או משנה של השחקן שיאותת לו שהחל מעכשיו הוא ישתמש בירייה המשולשת במקום בירייה רגילה. ברמת השחקן נצטרך את הדברים הבאים: (1) משתנה בוליאני שמסמן אם עכשיו יורים ירייה משולשת. (2) משתנה עצם מסוג ירייה משולשת שאותו הוא יאתחל בכל פעם שנלחץ על מקש ספציפי (במקלדת). (3) מתודה שתפעיל מתודת IEnumerator למשך זמן שבו הערך של המשתנה הבוליאני יהיה אמת, עד שייגמר הזמן והמשתנה יחזור להיות שקר. לא קשה להבין אם ככה איך ליישם את זה מבחינת סינטקס:

```
[SerializeField]
private GameObject _tripleShot;
private bool _tripleShotActive = false;
```



ובמתודה shoot() או ב-update() (תלוי אם הכנסתם את הקוד של הירי למתודה בפני עצמה והפעלתם אותה ב-update) , נעשה את השינויים נעשה תנאי: אם TripleShotActive חיוב, ניתן ווקטור כיוון לירייה המשולשת, ונשתמש ב-instantiate כדי לאתחל האובייקט, אחרת נעשה את אותה פרוצדורה שעשינו עד עכשיו עם ירייה אחת:

```
if (!_tripleShotActive)
{
    Vector3 laser_position = new Vector3(transform.position.x, transform.position.y + 1, 0);
    Instantiate(laser, laser_position, Quaternion.identity);
}
else
{
    Vector3 laser_position = new Vector3(transform.position.x, transform.position.y + 1, 0);
    Instantiate(_tripleShot, laser_position, Quaternion.identity);
}
```

(ייתכן ונצטרך לאתחל את הירייה המשולשת במרחק יותר גדול מהשחקן).

עכשיו נצטרך להוסיף את המתודה הייעודית לירייה המשולשת, לצורך הדוגמא נחכה חמש שניות עד שנגמרת הירייה:

```
public void TripleShotActive()
{
    _tripleShotActive = true;
    StartCoroutine(TripleShotRoutine());
}

IEnumerator TripleShotRoutine()
{
    yield return new WaitForSeconds(5f);
    _tripleShotActive = false;
}
```

נחזור לגוף ה-power up שלנו אנחנו צריכים להוסיף לו רכיבים שמאפשרים התנגשות כלומר collider2D כלשהו, rigidbody וכמובן סקריפט שעליו הוא ירוץ. לאחר שהוספנו לו את הרכיבים הרלוונטיים נתמקד בקוד שלו. דבר ראשון אנחנו רוצים שה-power up שלנו ינוע בקו ישר כלפי מטה, ובניגוד לאובייקט שמגיחים ממקומות שונים על המסך, ה-power up יופיע רק עד שהוא חוצה את המסך ואז יושמד.

היות וכבר ראינו איך לעשות את זה כל כך הרבה נראה לי שמיותר לציין את זה כאן. מי שעדיין לא זוכר כדאי לו להסתכל בפרק על הלייזר. בדיוק כמו שעשינו אם האויב גם כאן נשתמש בפונקציה המפורסמת OnTriggerEnter2D ובדיוק כמו באויב גם כאן נברר תחילה אם ל-other יש את התג של השחקן, במידה וכן נרצה לקבל את הרכיב של הסקריפט מother ולהפעיל את המתודה triplshotactive() של השחקן. ולאחר שהפעלנו נשמיד את ה-power up כי כבר השתמשנו בו:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        Destroy(this.gameObject);
        Player player = other.transform.GetComponent<Player>();
        if (player != null)
            player.TripleShotActive();
        Destroy(this.gameObject);
    }
}
```

אם נריץ את המשחק ונוודא שהכל כשורה, נראה שעדיין יש לנו בעיה אחת שמציקה לנו- האובייקטים של הירייה המושלשת עדיין מופיעים לנו על חלון ההיררכיה. זה משום שהרסנו את הלייזר, אבל לא את האבא שלו, כלומר ה-power up עצמו של הירייה המשולשת. לשם כך נצטרך לחזור לסקריפט של הלייזר ולבדוק האם יש לנו גם אובייקט אב ללייזר, במידה וכן נשמיד את האובייקט האב ביחד עם הלייזר כאשר הוא יוצא מגבולות המסך:

```
void Update()
{
    transform.Translate(Vector3.up * Time.deltaTime*_speed);
    if (transform.position.y>8.0f)
    {
```



```

        if (transform.parent != null)
        {
            Destroy(transform.parent.gameObject);
        }
        Destroy(this.gameObject);
    }
}

```

-Spawn power up

כמו שיצרנו מתזמן לאויב נוכל ליצור גם power up. למעשה נוכל להשתמש באותו מתזמן שהשמנו בו לפני, מה שנצטרך זה להוסיף משתנה עצם של power up ופונקציה coroutine חדשה שתרוץ פעם בכמה שניות כמו שעשינו ל-enemy. בשביל לעשות את זה מעניין אפילו יותר, נדאג שהקריאה למתודה תהיה באופן רנדומלי בין 3 ל-8 שניות:

```

IEnumerator SpawnPowerUpRoutine()
{
    while (true)
    {
        Vector3 postspawn = new Vector3(Random.Range(negative, positive), 13, 0);
        GameObject new_powerUp = Instantiate(_powerUpPrefabs, postspawn, Quaternion.identity);
        yield return new WaitForSeconds(Random.Range(3, 8));
    }
}

```

וכמובן לא לשכוח לעדכן את start להפעיל את הפונקציה.

מגן-

ה power up השני שנעשה הוא מגן- כל פעם שניקח אותו תהיה לנו הגנה למשך זמן מוגבל מפני התנגשויות מאויבים. בהתחלה נבצע את אותם צעדים שעשינו עם ה-power up הקודם: נמצא לו תמונה, נתאים אותה למסך, נוסיף לו collider2D ו-rigidbody2D מתאים, נוודא שה-gravity שווה אפס, וה-is trigger מסומן. לפני שנתעסק בכל התהליך של ה'מגן' נשנה את ה-spawn manager. אין סיבה באמת שנעשה מתודת coroutine חדשה עבור כל power up חדש שנוסיף, זה לא יעיל, ובעיני לאסוף כמה power ups במקביל הורס קצת את החוויה. אז למה שלא נשתמש באותה מתודה שאיתה אנחנו מתזמנים את ה-power ups כך שתגדיל איזשהו אחד כל כמה שניות. אם ככה נצטרך לשנות את המשתנה power up למערך של game object שמכיל את כל סוגי ה-power ups שיש לנו. נעשה את זה ונראה עכשיו שקיבלנו הערה מה visual studio היכן שאנחנו מאתחלים את ה-power up. נסמן אותה כהערה(עם שני קווים אלכסוניים) בינתיים ונחזור לunity. נשים לב שכשיו inspetor של ה spawn manager מופיע לנו המשתנה Power up עם אפשרות להגדרת גודל. בינתיים יש לנו שני power ups לכן נקבע את הגודל ל-2, ובהמשך אם נחליט להוסיף נגדיל את הגודל.

נגרור את האובייקטים למקום המתאים להם ב-inspetor ונחזור לקוד. כעת מטרתנו היא שנאתחל power up אחד מתוך המערך כל פעם, ושנעשה את זה באופן אקראי. במילים אחרות אנחנו צרכים איזשהו משתנה אינטגרי שמוגרל בין 0 לגודל המערך (או קרוב לגודל המערך), ואז נאתחל את האובייקט של המערך שנמצא במקום של הערך שיצא לנו. מבחינת לוגיקת המגן צריך שיקרו שני דברים: (1) מתי שהשחקן לוקח את ה-power up מופיעה תמונה של מגן, או כל דבר שמסמל שהסטטוס של השחקן השתנה ועכשיו הוא חסין אויבים באופן זמני. (2) לדאוג מבחינת קוד שהוא לא יפגע, אבל אויבים יפגעו ממנו במשך כמה שניות מהאויבים.

ראשית נתעסק בחלק השני כי הוא יותר קל ליישום, וכבר ראינו דבר דומה עם הירייה המשולשת. נצטרך להוסיף איזשהו משתנה בוליאני כך שמסמן לנו בקוד שעכשיו אנחנו במצב 'מגן' ומתודה damage() נוודא שאם הוא מופעל (עם ערך 'אמת') אז לא ירדו לנו חיים כלומר: return; if(!_isShilded) (ואז שאר הקוד עם - - life וכו').

ובדיוק כמו שעשינו עם הירייה המשולשת גם כאן נוסיף מתודת coroutine בשביל להגביל את זמן השימוש במגן. באשר לקוד של המגן- ברמת העיקרון אין באמת צורך לבנות סקריפט חדש במיוחד למגן, נוכל פשוט להוסיף לקוד הישן של הסקריפט power up .



נגרור את הסקריפט לאובייקט 'מגן' ונערוך את הסקריפט באופן הבא: לכל סוג של power up ניתן איזשהו id ייחודי, שכן של כל אחד מפעיל מתודה אחרת של השחקן. לכן נייצר משתנה אינטגרי חדש ונקרא לו _id, נדאג שנוכל לראות אותו בunity. במתודה OnTriggerEnter נצטרך לחלק למצבים לפי ה-id של הpower up. יש כמה שיטות לעשות את זה, הקלה שבהם היא ע"י switch-case (לאו דווקא השיטה החכמה שבניהם). מבחינת סינטקס זה יראה כך:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        Destroy(this.gameObject);
        Player player = other.GetComponent<Player>();

        if (player != null)
        {
            Switch(_id)
            {
                case 0:
                    player.TripleShotActive();
                    break;

                Case 1:
                    player.ShieldedActive();
                    break;
            }
        }
        Destroy(this.gameObject);
    }
}
```

ועכשיו נצטרך לעדכן ב unity את הid בהתאם.

לאחר שהגדרנו את השחקן כראוי נחזור לחלק הראשון- להוסיף תמונה שתתווסף לשחקן כאשר אנחנו לוקחים את המגן. נמצא תמונה לייצוג המגן, נערוך אותה כך שתהיה נחשבת sprite, נמקם אותה בשכבה המתאימה ובמיקום המתאים יחסית לשחקן שלנו ונגרור אותה לחלון ההיררכיה. כדי שהתמונה תזוז ביחד עם השחקן היא צריכה להיות אובייקט 'בן' לשחקן, לכן נגרור את אובייקט התמונה לתוך אובייקט השחקן. ברמת העיקרון מה שאנחנו רוצים לממש זה שכאשר לקחנו מגן אז התמונה שמייצגת אותו תופיע על המסך, וכאשר נגמר הזמן של המגן אנחנו חוזרים למצב שהתמונה לא קיימת. אם נשים לב בinspector למעלה יש ריבוע קטן ליד שם האובייקט, הריבוע הזה הוא משתנה בוליאני 'Active' של האובייקט, הוא מסמן האם האובייקט פועל עכשיו או לא. מה שאנחנו רוצים לעשות זה בעצם להפעיל את המשתנה הזה של התמונה שמייצגת את המגן, כלומר לתת לו ערך חיובי כאשר מופעל המגן, ולבטל אותו (לתת לו ערך שלילי) כאשר הוא מסיים את העבודה שלו. ל unity יש מתודה מיוחדת שמביאה ערך לאותו משתנה: GameObject.SetActive(bool status). בנתיים נסמן את האובייקט כלא אקטיבי. נצטרך כמובן גם לידע את השחקן שיש אובייקט בן כזה, כיצד נעשה את זה? כמו שעשינו עד עכשיו: ניצור משתנה עצם מסוג GameObject ונגרור אליו את האובייקט בין באינספקטור. לאחר שהשחקן מודע לקיומו של ה"מגן" ניתן להפעיל עליו את ה-setActive כך: shiled_pic.Gameobject.SetActive(true) נעשה את זה במתודת coroutine של השחקן שמפעילה את המגן, ו'נכבה' אותו בסוף המתודה.

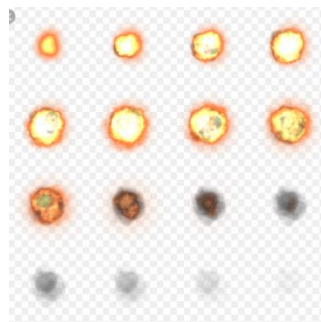


אנימציות –

אחד היתרונות הבולטים שיש ל-unity על פני מנועים גרפיים אחרים הוא הקלות שבה ניתן ליצור אנימציות. אנימציות לרוב באות באחת משתי דרכים- או אנימציה קבועה שיש לאובייקט, לרוב Prefabs של חפצים או יצורים ש"מקשטים" את המשחק, אבל לא משפיעים עליו ממש. או ע"י מכונת מצבים שמתזמנת את האנימציה המתאימה לאובייקט בהתאם לסיטואציה בה הוא נמצא- הולך, בטל (idle) קופץ וכו'. מה שיפה באנימציות ב-unity הוא האפשרות ליצור את האנימציה ע"י אוסף של sprites שמתקבצים לכדי סרטון קצר שחוזר על עצמו בלופ, וניתן גם "להקליט" את האנימציה- כלומר בצורה אינטראקטיבית ליצור את האנימציה ע"י הזזת האובייקט בכל פריים (של סרטון, לא יחידת זמן של המשחק). עוד מנגנון שיש ל-unity הוא היכולת לחבר כמה sprites שמייצגים חלקים מהאובייקט (למשל ראש, רגליים ידיים וכדו') לכדי אובייקט אחד וכך יהיה ניתן להזיז כל חלק בנפרד וליצור את האפקט של תנועה רב מערכתית מבלי לייצר תמונה במיוחד לכל פריים של סרטון.

אפקט פיצוץ-

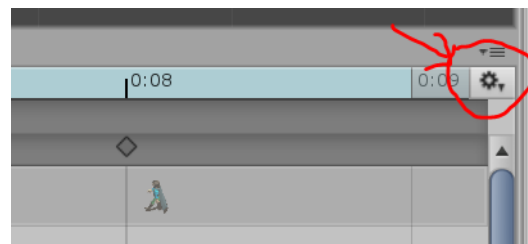
האפקט פיצוץ ישמש אותנו לשני מצבים- א. כאשר האויב יושמד, בין ע"י הלייזר ובין בעקבות התנגשות עם השחקן. ב. לשחקן- כאשר נגיע למצב שנגמרו לשחקן החיים נרצה שהוא יבצע אפקט פיצוץ. את האפקט כמעט בטוח שניתן למצוא ב-unity store (בחינם). אם לא מצאנו אל דאגה נשתמש בשיטת ה-"חפש בגוגל". בגוגל יש לחפש משהו בסגנון של "explosion sprite png". אנחנו צריכים תמונה שמכילה בתוכה מלא תתי-תמונות של מצבים בפיצוץ, אם מצאנו תמונה כזאת בלי רקע מעולה, אחרת ניצור אחת עם קריטה כמו שראינו כבר. דוגמא לתמונה:



לאחר שיש לנו כבר את התמונה נגרור אותה ל-unity ונגדיר אותה כ- sprite(2D and UI) וב- sprite mode צריך להגדיר אותה multiples (כי עכשיו אנחנו מתעסקים עם כמה תמונות ולא עם תמונה יחידה) ואז נלחץ על ה-sprite editor כדי לערוך את רצף התמונות לכדי אוסף תמונות. הערה חשובה: יכול להיות שלא מותקן לכם ה-sprite editor בפרויקט, כל מה שצריך כדי להתקין אותו זה ללכת ל- Window >- Package Manager ובשורת החיפוש לחפש 2D Sprite ואז להתקין אותו בכפתור install שמופיע בתחתית החלון. אחרי שיש לנו כבר את הספרייט נפעיל את ה-sprite editor. שם נראה שיש לנו כפתור slice, הוא אחראי לחתוך את התמונה לספרייטים קטנים בהתאם למה שנגדיר לו: ניתן לחתוך בצורה אוטומטית או ע"י איזשהו grid. אחרי שבחרנו דרך לחתוך ואישרנו את ה-slice נסגור את החלון. כעת אפשר לראות את כל הספרייטים שחתכנו מהתמונה המקורית ע"י החץ שמופיע בצד התמונה. כדי להפוך את רצף הספרייטים לכדי אנימציה אחת, נצטרך איזשהו אובייקט שיכיל בתוכו את האנימציה, כמין מכולל אנימציות. ניצור אובייקט ריק חדש ונתן לו שם, בעודנו על האובייקט נקרא לחלון עורך האנימציות, נבחר <animation-window - animation>.

אם ניתן ננסה אפילו להצמיד את החלון שיהיה באותה שורה של חלון הסצנה, כך יהיה נגיש יותר בהמשך. ניצור אנימציה חדשה ונשמור אותה. כדי לייצר את האנימציה של הפיצוץ נצטרך לגרור את כל הספרייטים הרלוונטיים לנו לאנימציה לכן נבחר בכל הספרייטים של תמונת הפיצוץ ונגרור אותם לחלון האנימציה. אם נלחץ על כפתור ה-play נראה שהאנימציה רצה לנו על המסך. יכול להיות שהאנימציה רצה מהר מידי לטעמנו או לאט מידי, אל דאגה ניתן לשנות את מהירות האנימציה ע"י ה-samples. הוא אחראי על המהירות של ריצת הספרייטים. אם לא מופיע לכם samples כשאתם נכנסים לחלון האנימציה בחרו בגלגל השיניים הקטן שנמצא בפינה הימנית העליונה בחלון





- `show sample rate` . אפשרות שניה, פחות נוחה, היא להזיז ממש את הפריימים של האנימציה. אם הגודל של האנימציה לא מוצא חן בעיניכם כמובן שניתן לשנות אותו, ואם הצבע של הפיצוץ חזק מידי או חלש מידי ניתן לשחק בצבעים של האנימציה ה-`renders` השונים שניתן להוסיף ב-`add properties` בחלון האנימציה. אם אנחנו מתכוונים להשתמש באותה אנימציה לכמה מצבים כדי להפוך אותה ל-`prefab`. אז יש לנו אנימציה, השאלה איך נחבר אותה לדמויות השונות כך שהיא תפעל ברגע המתאים? בתור התחלה נחבר את האנימציה שתהיה אובייקט 'בן' לאובייקט שעליו הוא פועל כדי שיזוז איתו. לצורך הדוגמא (בה"כ) ניקח את השחקן. השחקן אמור להתפוצץ כאשר ה"חיים" שלו נגמרים, במילים אחרות ברגע שהמשתנה חיים שווה לאפס אנחנו צריכים להפעיל את האובייקט שאחראי על הפיצוץ ואז לעשות `destroy()` על השחקן. אם ניזכר רגע באיך יצרנו את המגן, השתמשנו במתודה המיוחדת שמפעילה את האובייקט הבן: `GameObject.SetActive(bool status)`. גם כאן אנחנו רוצים להפעיל אותה על האנימציה במידה והגענו למצב שלא נשאר לשחקן חיים. חשוב לא לשכוח 'ליידע' את השחקן על האובייקט בן שנוסף אליו, כדי שיוכל להפעיל אותו. כדי שהשחקן לא יזוז תוך כדי שהוא מפעיל את האנימציה, כי אחרת זה נראה מוזר, פשוט נגדיר את `_speed` להיות 0 כשמפעילים את האנימציה, כך השחקן לא יוכל לנוע בזמן הפיצוץ. לאחר שהפעלנו את האנימציה נוכל לקרוא לפונק' `destroy()`. הערה: יכול להיות שהפונק' `destroy` תקרא לפני שהאנימציה פעלה, או באמצע הפעולה, במקרה כזה כדי שנוסיף מתודת `coroutine` או שנכנס את כל ה"פרוטוקול הרס" לתוך מתודה כזו, ונקרא ל-`yield return` כאורך האנימציה.

```
public void Damage()
{
    life--;
    if(life<1)
    {
        StartCoroutine(Explosion());
    }
}
```

```
IEnumerator Explosion()
{
    _explosion.SetActive(true);
    _speed = 0;
    yield return new WaitForSeconds(0.7f);
    Destroy(this.gameObject);
}
```

דבר דומה ניתן לעשות גם עם ה-`prefab` של האויב, רק לשים לב שהאויב נפגע גם מפגיעה ע"י לייזר וגם ממגע בשחקן.

מכונת מצבים-

לפעמים יש צורך ביותר מאנימציה אחד לאובייקט, או בתזמון בין האנימציות של אובייקט שיקרה בזמן קבוע או ע"י טריגר. לכן נעדיף להשתמש במכונת מצבים (`animator`). מכונת מצבים כשמה כן היא אוטומט שמתזמן את האנימציות של האובייקט בהתאם לטריגר (או פרמטר אחר) שמעביר אותו ממצב למצב. למשל אוטומט שמעביר מצב אם הוא קיבל אחד או אפס ניתן לדמות אותו למעבר מצב של דמות מזמן בטלה (`idle`) לזמן ריצה ולהפך- אם קיבלנו כקלט 1 אז הדמות תציג את האנימציה של הריצה, אם קיבלנו אפס (או יותר נכון לא קיבלנו קלט) אז היא תציג את האנימציה של הבטלה. המכונה בנויה ממצבים (או אנימציות), `transitions` – מעברים בין מצב למצב, `parameters`- הטריגרים שדרכם המכונה יודעת



לאיזה מצב לפנות. מצב שמוגדר כברירת מחדל יהיה בצבע כתום.

בשביל להפעיל את המכונת מצבים דבר ראשון נצטרך להוסיף את חלון ה-animator למסך הראשי, ע"י window > animator

נבחר את האובייקט עליו נרצה לעבוד. בעדיפות על אובייקט עם כמה אנימציות, אבל גם אובייקט עם אנימציה אחת מספיק לנו ובלבד שיהיה לנו מעבר בין מצב אנימציה למצב בלי אנימציה.

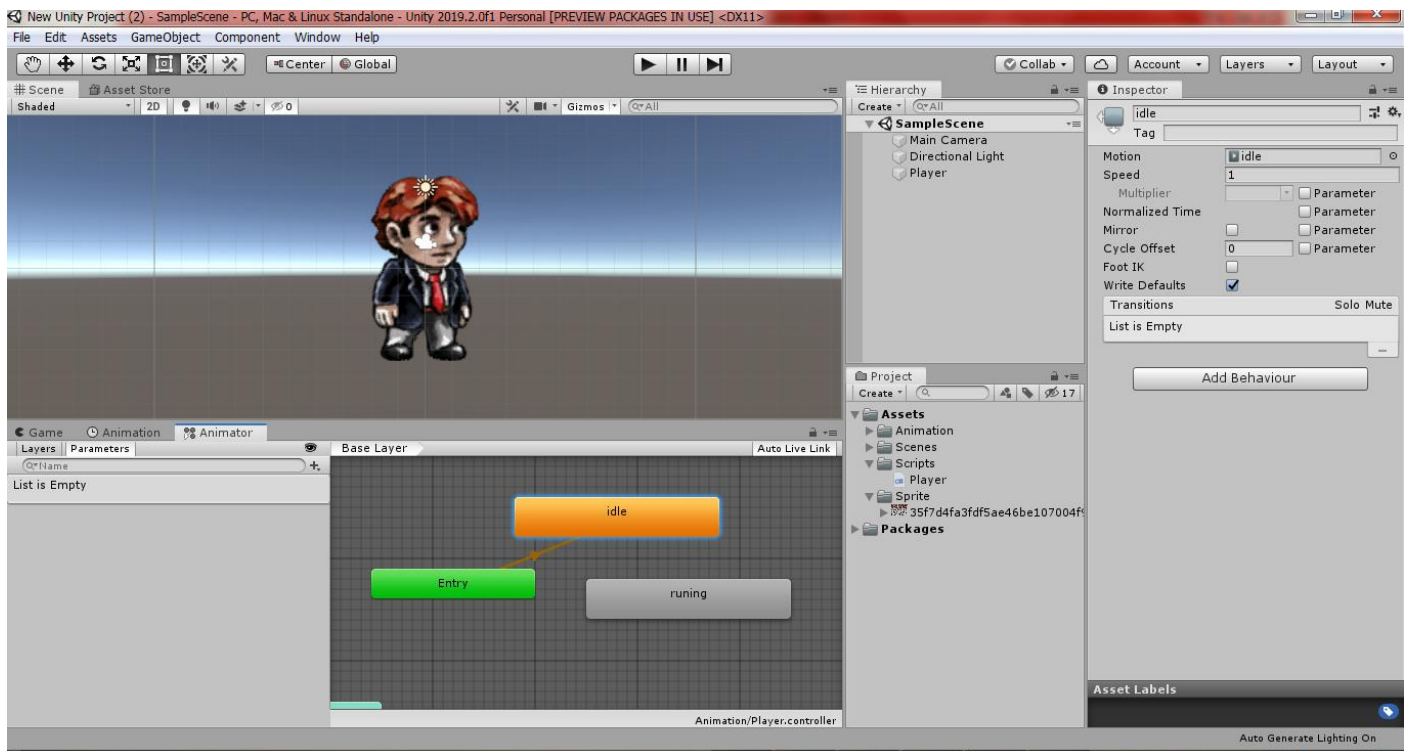
דוגמא מדמות שעשינו למשחק אחר: דמות של עורך דין שיש לו שני מצבים (1) מצב עומד או בטל (idle), הדמות נשארת במקום ויש לה אנימציה מהספרייטים הבאים:



(2) מצב רץ (running), הדמות נעה:



לכל רצף ספרייטים יצרנו אנימציה נפרדת. ניתן ליצור כמה אנימציות לאובייקט בחלון ה-animation וליד ה-samples (מצד שמאל) יש לשונית עם שם האנימציה, אם פותחים אותה יש למטה את האפשרות Create new clip. כשפתחנו את animator קיבלנו את החלון הבא:



ניתן לראות מהתמונה שהמצב המוגדר כברירת מחדל הוא המצב idle, ובינתיים אין לנו שום קשרים בין מצבים למעט הקשר היחיד שיש לנו שהוא מעבר מ-entry שהוא הפעם ה- q_0 שלנו, ומשתמשים בו פעם אחת באתחול הדמות ויותר לא חוזרים אליו, ל-idle שהוא האנימציה הראשית.

כדי להוסיף קשרים (transition) נלחץ מקש ימני על המצב שממנו יוצא הקשר-<make transition ונגרור אותו למצב אליו הוא

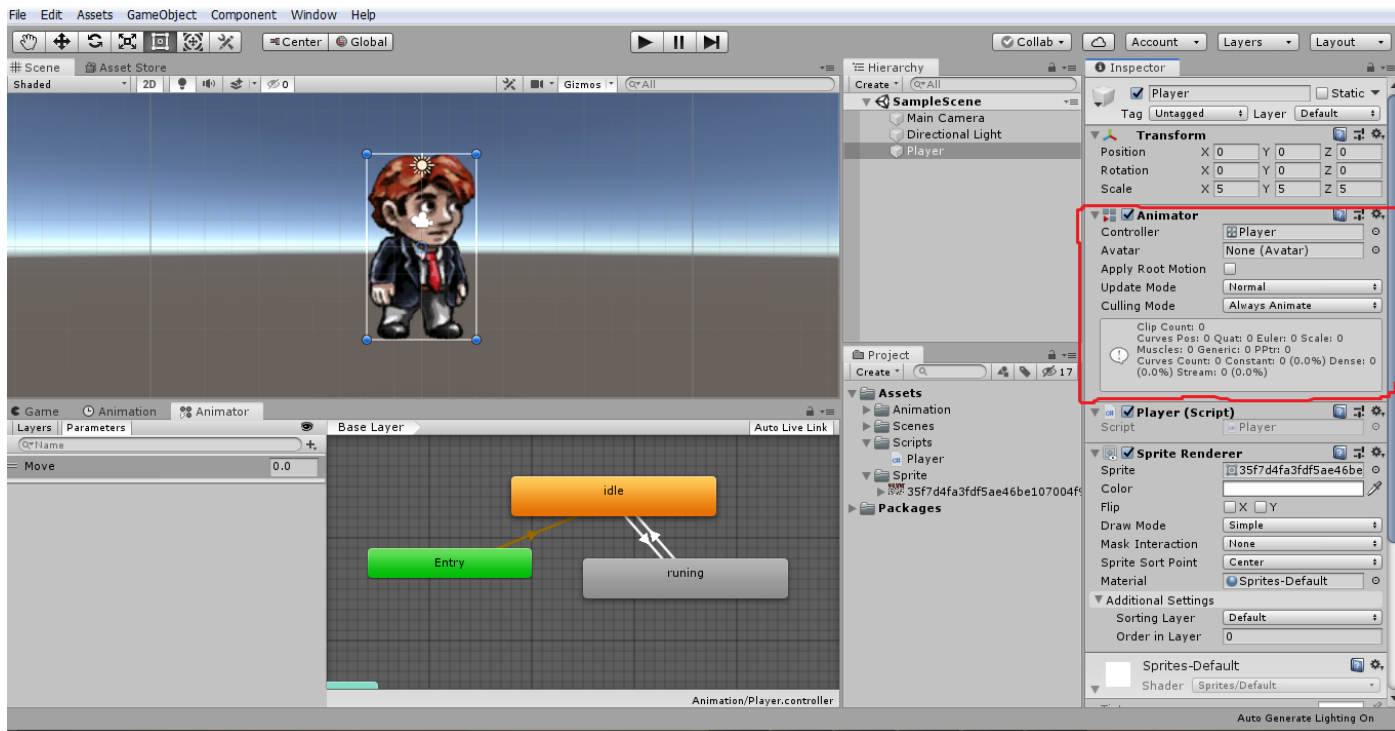


אמור להגיע, וכנ"ל בכיוון ההפוך.

בהרצה של המשחק הדמות מחליפה בין אנימציות לאחר כמה שניות, זאת משום שה-transition עדיין מוגדרים ב-inspector כ-
has exit time. אנחנו רוצים להוסיף לדמות פרמטרים כדי שהמעבר יהיה לפי דרישה.

אם נשים לב מהתמונה למעלה בצד שמאל של המכונת מצבים יש לנו layers ו-Parameters, כדי להוסיף פרמטר חדש נצטרך לעמוד על חלון הפרמטרים וללחוץ על הפלוס הקטן מצד ימין לכפתור החיפוש. יש לנו אפשרות לבחור את הסוג של הפרמטר (int, float, bool, trigger). משום שאנחנו מתעסקים בתזוזה כרגע נבחר float. ניתן שם לפרמטר (אצלנו קוראים לו Move), נלחץ על אחד הקשרים ונסתכל על האינספקטור. נשים לב שבאינספקטור מתחת ל-setting של ה-exit time יש לנו רכיב שקוראים לו Conditions הרכיב אחראי להוסיף לקשר פרמטר שלפיו הוא עובד. נוסיף את הפרמטר שיצרנו לקשר ע"י לחיצה על הפלוס הקטן מצד ימין (add to list). התנאי שהוספנו כרגע בנוי משלושה חלקים- שם התנאי, במקרה שלנו Move, לשונית שמגדירה האם גדול מ (Greater) או קטן מ (less) וחלון להזין את המספר שממנו הוא אמור להיות גדול/קטן. היות וההתנהלות שלנו היא בוקטורים, אז אם אנחנו במצב של vector2d.right כלומר אם קיבלנו בפלט לפנות ימינה ואנחנו מתקדמים אז אנחנו כבר לא עומדים במקום אלא נעים, כלומר אם קיבלנו קלט מהמשתמש נעבור ממצב של אנימציה עומדת לאנימציה רצה, לכן עדיף כאן שהתנאי יהיה אם Move גדול מ-0 (או 0.1 אם אנחנו עוברים בין מצבים שיש להם כמה אפשרויות), וכנ"ל לכיוון ההפוך רק עם less.

ועכשיו לחלק התכנותי. בהסתכלות על ה-inspector של הדמות ניתן לראות שיש לנו כבר משתנה מסוג animator:



לכן מספיק לנו לקרוא לרכיב במתודת start() של הדמות בכדי לאתחל אותה. ניצור משתנה עצם מסוג אנימציה שאותו נאתחל להיות הרכיב אנימציה:

```
private Animator _anim;
void Start()
{
    _anim = GetComponent<Animator>();
}
```

המטרה היא שהדמות תעבור למצב אנימציה "רץ" במידה וקיבלנו ערך שהוא גדול מאפס, לשם כך נצטרך לעדכן את הפרמטר שיצרנו. ב-unity יש מתודה במיוחד לכך: <Animation name>.Set<Parameter type>(<Parameter name>,<var>): למשל אצלנו אנחנו רוצים "לערוך" את הפרמטר של _anim (כך קראנו לאנימציה) והוא מסוג Float לכן נשתמש במתודה.SetFloat (ואם זה היה int אז היינו משתמשים ב-SetInt,



(וכו'), שמקבלת את שם הפרמטר (התנאי) כמחרוזת, ולפי מי התנאי מתקיים. נניח אנחנו רוצים משתנה שהערך שלו הוא פלט מהפונקציה input אם לחצנו על איזשהו חץ אופקי ("horizontal"), נקרא לו direction לצורך הדוגמא, אזי המתודה תראה כך:

```
void Update()
{
    float direction = Input.GetAxis("Horizontal");
    _anim.SetFloat("Move", direction);
    ...
}
```

כשמריצים את המשחק רואים שהשחקן באמת עובר בין המצבים אבל עדיין יכול להיות שהוא לא עובר ישר בין אנימציות אלא מחכה לסוף האנימציה האחת בכדי לעבור לאחרת. כנראה שהסיבה שזה קורה היא כי ה-Has Exit Time עדיין מסומן בשני ה-transitions, אם נבטל אותם המעבר יהיה חד יותר בהתאם לקלט אותו הוא מקבל, ואם נרצה מעבר חד אפילו יותר, בתוך ה-setting (מתחת ל-has exit time) יש transition duration הוא אחראי לדיילי בין המעברים, ואם נשווה אותו ל-0 המעבר יהיה חד הרבה יותר, אך לפעמים נעדיף דווקא שהמעבר לא יהיה חד מידי, כי אחרת זה לא נראה "ראליסטי" מספיק. אומנם הצגנו דוגמא פשוטה יחסית לשימוש במכונת המצבים, ואם ננסה ליישם את זה במשחק שלנו יהיה לנו קצת יותר מסובך כי הקוד קצת יותר עמוס, אך הבסיס הוא אותו בסיס בשניהם.

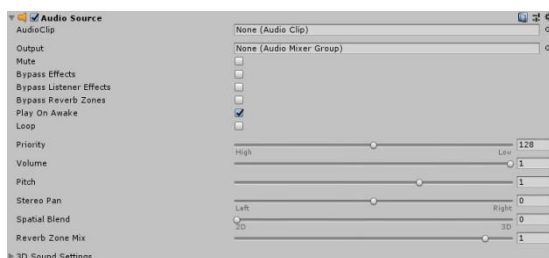
במשחק שלנו אפשר ליישם את מכונת המצבים בהרבה מקרים: בחללית או בדמות הראשית כשהיא יורה או כשהיא זזה; אם נרצה להוסיף "בוסים" למשחק, כלומר אויבים יותר גדולים מהאויבים הקטנים שמתים אחרי ירייה אחת, אפשר להוסיף להם אנימציות ביניים כל פעם שפוגעים בהם עד שהם מתפוצצים לחלוטין, ויש עוד אינספור דוגמאות.

הערה: בדוגמא לעיל הראנו רק שימוש בפרמטר מסוג float הדבר דומה מאוד במקריים של bool ו-int, אבל במקרה של פרמטר מסוג trigger הוא קצת שונה. בניגוד לשלושת הקודמים, במצב טריגר אנחנו מחכים שיקרה איזשהו אירוע, ולא דווקא קלט, למשל אם היינו עושים אויב "בוס" היינו משתמשים בטריגר כאשר הדמות הראשית פגעה באויב כמה פעמים ועכשיו הוא להתפוצץ ("להיות מושמד").

פרמטר מסוג טריגר אנחנו מפעילים כאשר אנחנו רוצים שיהיה מעבר בין האנימציות במקרה קיצוני ולא משהו נשלט לגמרי ע"י השחקן, נגיד לאויב "בוס" לפני שנעשה עליו destroy() נפעיל את הטריגר(ולרוב נשהה את האובייקט כמה שניות בשביל שתרוץ האנימציה) ואז נשמיד את הדמות סופית.

קבצי שמע

אפקטי קול הם חלק אינטגרלי בכל משחק שמכבד אותו(למעט משחקים שהקהל יעד שלהם הוא חרשים). החשיבות באפקטים קוליים ושימוש במנגינות רקע שהם יכולים לשלוט ברגשות השחקנים ולהכתיב את הטון של הסיפור שמלווה את המשחק. יש משחקים שהסאונד-טראק שלהם נהיה כל כך מהותי במשחק שאנשים זוכרים בעיקר אותו. קחו לדוגמא משחק כמו סופר-מריו, אם נעשה סקר שוק מה אנשים זוכרים יותר- את הסיפור רקע של המשחק או את הסאונד-טראק שלו, סביר להניח שיותר אנשים יזכרו את המוזיקה ולא את הסיפור על אף שהוא מהות המשחק (לכאורה). השימוש בסאונד ב-unity מופעל ע"י שני דברים: AudioSource - שאחראי לנגן את הקבצים בזמן המשחק. במשחקי תלת-ממד הצליל יכול להתכוון לפי המרחק, נניח דמות רחוקה ממך את תשמע אותה פחות מאשר אם תהיה לידה; ו-AudioClip – הקובץ שמע (למשל mp3) שאותו מנגנים.



ניתן לראות מהתמונה לעיל כי ל- AudioSource יש את האפשרויות הבאות:

output - להיכן יוצא הקובץ. כברירת מחדל הקובץ שלנו יוצא לאיזשהו Audio Listener שהוא כמין מכשיר מיקרופון. לרוב הוא מחובר למצלמה. אפשרות נוספת היא להוציא את הסאונד ב-Audio Mixer בשביל להפיק יותר אפקטים קוליים

Mute - משתנה בוליאני שמגדיר האם הסאונד נשמע כעת או לא. הוא לא עוצר את נגינת הסאונד אלא רק "מנמיך" את עוצמת הקול ל-0.

Bypass effect - דרך מהירה להפעיל/לכבות את כל האפקטי קול של ה-AudioSource.

Bypass listener effect - אותו דבר רק על ה-Audio Listener.

Play on Awake - משתנה בוליאני שמאפשר להפעיל את הקובץ ישירות כאשר הסצנה מוטענת למסך. אם לא נסמן את זה נצטרך להפעיל את האודיו-קליפ בקוד דרך המודה Play().

Loop - מנגן את הצליל בלופים עד שיבטלו את הצליל את המשתנה loop.

Priority - קובע את חשיבות הצליל יחסית לצלילים אחרים בסצנה. 0- החשיבות הכי גבוהה ו-256- החשיבות הכי נמוכה(הברירת מחדל היא 128). משתמשים ב-0 לרצועות מוזיקה בדר"כ.

Volume - עוצמת הסאונד. מוגדרת להיות כמה המוזיקה "רחוקה" מאתנו (מה-Audio Listener) כל יחידה שווה בערך מטר.

Pitch - מהירות הסאונד. 1-מהירות נורמלית, מתחת לזה המהירות איטית יותר ומעל זה המהירות גבוה יותר, המהירויות מוצגות ככפולות של המהירות המקורית, כך למשל מהירות 1 היא כאילו פי 1 מהמהירות המקורית שזאת אותה מהירות.

Stereo pan - קובע להיכן הצליל קרוב יותר- לאזור הימני או השמאלי.

Spatial Blend - קובע כמה השפעה יש למנוע התלת-ממדי על ה-AudioSource.

עצם היות ה-AudioSource רכיב של unity, כלומר יורש מ-monobehavior, הוא יכול להיצמד ישירות לכל GameObject של unity.

חלק מרכזי מהפעלת אפקטים קוליים הוא השימוש באיזשהו טריגר שיפעיל אותם, איזשהו תנאי שיתקיים שלפיו יופעל הקובץ(למשל פגיעה ע"י לייזר, תחילת משחק וכו').

מוזיקת רקע-

נתחיל בדבר הפשוט יותר לעשות- מנגינת רקע.

בשביל מוזיקה שמתנגנת ברקע נצטרך איזשהו אובייקט שיכיל בתוכו את האודיו-קליפ. ניצור אובייקט ריק חדש ונוסיף לו את הרכיב AudioSource. למי שלא זוכר, כדי להוסיף רכיב חדש צריך ללכת ל- Add Component באינספקטור, ולחפש בשורת חיפוש את שם האובייקט אותו אנחנו רוצים.

לתוך הרכיב שהוספנו נגרור את קובץ שמע לשורה Audio Clip, ונוודא ש-play on awake מסומן(שברגע שמופעל האובייקט משחק מופעל גם האפקט קול), ושגם loop מסומן, כי אחרי הכל אנחנו רוצים שהשיר יתנגן במשך כל הסצנה. וברמת העיקרון זהו, לא צריך אפילו להתעסק בקוד. נשמור את הסצנה ונריץ את המשחק לוודא שהכל עובד. ולפני שנמשיך ניתן למצוא ולהוריד מוזיקת רקע נהדרת למשחק דרך האתר YouTube. למי שלא מכיר, יש מלא אתרים שמאפשרים להוריד את התוכן של הסרטון או לפחות את הפס-קול שלו בפורמט mp3. דוגמא לאתר:

<https://2conv.com/en4/youtube-mp3>

אפקטים קוליים-

יש כמה גישות באשר איך לעשות את האפקטים הקוליים. לפי הגישה הראשונה שצורכת מינימום של קוד היא להוסיף לכל אנימציה ייעודית (או אובייקט ייעודי) את הצליל שמתאים להם שיופעל ברגע שהם נוצרים. למשל נצמיד לאנימציה של הפיצוץ סאונד של פיצוץ כאשר הוא נוצר, ואפילו לא נצטרך להיכנס לקוד היות ויש לנו את המשתנה Play on awake באינספקטור, כך שהצליל יופעל עם רגע היווצרות האובייקט. במקרים בהם יש לנו כמה אפקטים קוליים לאובייקט מסוים, לדוגמא דמות שפעם בעשר שניות מזכירה שצריך למהר, ופעם בחצי דקה מזכירה לו מה המשימה. נוכל להשתמש במערך של אודיו-קליפים וכל פעם יוטען AudioSource קליפ אחר. החיסרון כאן ברור- אין אפשרות לערוך כל קליפ בנפרד, יוצא שלכל הקליפים יהיו את אותם מאפיינים, מה שלא יעזור לנו אם נרצה שלכל קליפ יהיה את הערכים שלו. אפשרות אחרת היא לעשות מערך של AudioSource, ואז נטעין למאורע את האחד הרלוונטי לו מהמערך.



גישה שניה, צורכת קצת יותר מאמץ אך שווה את זה לטווח הארוך, היא ליצור איזשהו Audio Manager שינהל לנו את הסאונד המתאים לכל מאורע. כך באירוע מסוים נבקש ממנהל הסאונד שיפעיל לנו צליל שמתאים לאותו אירוע. זה יעזור לנו לשמור על קוד יותר גנרי-יהיה לנו מקום מסודר לכל קבצי הסאונד שלנו ולא נצטרך לקפוץ בין אינספקטורים של כמה אובייקטים כדי לשנות צליל מסוים. כמו כן זה מאפשר לשמור על אותם צלילים בכמה סצנות. כפי שנראה בהמשך במעבר בין סצנה לסצנה (או במעבר בין שלבים) לא נשמרים לנו אותם נתונים אלא אם הגדרנו מראש את האובייקט כ"לא נמחק בין סצנות" (כמין אובייקט סטטי). אם לא נשתמש במנהל, יהיה לנו קשה יותר לשמר כמה צלילים שיתנגנו לאורך כמה סצנות. נצטרך לעבור אובייקט אובייקט ולהגדיר אותו במיוחד. ובמנהל סאונד פשוט נגדיר אותו ככה וזה יספיק לנו. אבל איך ניצור את ה-Audio Manager?

-Audio Manager

תחילה ניצור אובייקט ריק חדש עם השם Audio Manager ונחבר לו סקריפט. אנחנו רוצים לשלוט בקבצי האודיו ולכן נצטרך להשתמש במרחב שם ייעודי של unity שמטפלת בקבצי אודי:

Using UnityEngine.Audio;

הרעיון במנהל האודיו הוא שנוכל להוסיף ולהפחית צלילים בקלות לרשימה תוך כדי שימוש. ואז כאשר נצטרך להשתמש באחד הצלילים נחפש את אותו הצליל שמתאים לאירוע מתוך הרשימה שהכנו מראש. בשביל שנוכל לשלוט במידע שנכנס לנו לתוך הרשימה בקלות ובלי סרבול של שימוש ב-AudioSource, ניצור מחלקה חדשה שתהווה מעטפת לאודיו קליפים. ניצור סקריפט חדש ונקרא לו Sound לצורך הפשטות. צריך להכליל במחלקת Sound שלנו גם את הספרייה הייעודית של unity לקבצי אודיו כפי שעשינו ל-Audio Manager. אין צורך שהמחלקה תירש מ-monobehavior אז נמחק את שורת הירושה, ונ"ל המתודות start() ו-update(). למחלה הייעודית שאנחנו בונים נצטרך כמה משתנים:

1. AudioClip - שאותו אנחנו מנגנים
2. משתנה float שאחראי על עוצמת הקול (volume)
3. משתנה float שאחראי על מהירות הסאונד (pitch)
4. משתנה bool שיגדיר לנו האם להשמיע את הצליל בלופים (loop)
5. מחרוזת שמייצגת את שם האובייקט עליו אנחנו עובדים (name)

בשביל הפשטות נגדיר את המשתנים כ-public שיהיה לנו קל לעבוד איתם (ניתן להגדיר כ-private ולהוסיף להם getter ו-setter). כדי להשתמש ב-unity במחלקה חיצונית (שלא מחוברת לשום אובייקט ולא יורשת מ-monobehavior) אנחנו צריכים לסנכרן את המחלקה עם המערכת. אפשר לסנכרן מחלקות ע"י ה-casting הייחודי: [System.Serializable] בראש ההצהרה של המחלקה.

בשביל להקל אף יותר, ל-unity יש את האפשרות להגדיר למשתנים כפתורים מיוחדים באינספקטור עוד בקוד ע"י casting מסוים, למשל כדי להפוך את המשתנה volume באינספקטור לכפתור הזזה עם טווח מסוים למשל בין 0 ל-1 נוכל להשתמש ב-casting: [Range(0f,1f)] כך נוסיף הגבלה למשתנה מבלי להיכנס ל-if מיותר, אותו דבר אפשר לעשות ל-pitch בערכים בין 0.1f ל-3f (כך הערכים ב-AudioSource).

בשביל להפעיל כל אודיו-קליפ אנחנו צריכים משתנה מסוג AudioSource, נגדיר אותו כ-public אך כדי לא לראות אותו באינספקטור אם צורך בכך (כי זה סתם תופס מקום מיותר), נצמיד את ה-casting: [HideInInspector]. מבחינת סינטקס זה יראה כך:

```
using UnityEngine;
using UnityEngine.Audio;
using System.Collections;
using System.Collections.Generic;
```

```
[System.Serializable]
public class Sound
{
    public string name;
    [Range(0f,1f)]
    public float volume;
    [Range(0.1f,3f)]
    public float pitch;
    public bool loop;
```



מבוא לפיתוח משחקי מחשב
ד"ר סגל הלוי דוד אראל

```
public AudioClip clip;
[HideInInspector]
public AudioSource source;
}
```

נחזור ל-Audio Manager: נצטרך משתנה שיאחסן בתוכו את כל משתני ה-sound שנשמור. ניתן להשתמש בכל דבר שישירש collection בעדיפות על dictionary (ה-Hash Table של c#) ששם פשוט נצמיד את השם של הקובץ כמפתח לאובייקט מסוג Sound, אך הבעיה בשימוש בdictionary היא שלא לא ניתן לראות את האובייקטים שנכנסים אליו דרך האינספקטור, או יותר נכון זה דורש הרבה יותר עבודה. לעומת מערך פשוט, שאומנם פחות יעיל אך הרבה יותר קל לעבוד איתו. אנחנו צרכים לעבור על כל משתני הסאונד ברשימה שלנו ולאתחל להם את ה-AudioSource כדי שנוכל לנגן את האודיו-קליפים ביתר קלות. נוכל לאתחל אותם באמצעות לולאת for each שתעבור עליהם אחד אחד ולהתאים את הערכים של ה-AudioSource של משתני סאונד שלנו, לערכים שהגדרנו להם מראש. את האתחול נבצע דווקא לא במתודה start(), אלא במתודה Awake(), שהיא דומה מאוד ל-start() אך שהיא מתחילה עוד קודם לכן (מתחילת הסצנה ולא מיצירת האובייקט). מבחינת סינטקס זה יראה כך:

```
public class AudioManager : MonoBehaviour
{
    public Sound[] sounds;

    void Awake()
    {
        foreach(Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.clip = s.clip;
            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.loop = s.loop;
        }
    }
    . . .
}
```

כמובן שבמהלך המשחק אנחנו צריך להריץ את האודיו-קליפ הרלוונטי למאורע, בשביל זה ניצור מתודה מיוחדת שתחפש את הצליל מתוך המערך ובמידה וקיים אובייקט כזה גם תנגן אותו, אחרת תעדיכן אותנו (המתכנתים) שמנסים להגיע לאובייקט שלא קיים ותחזיר ערך ריק. ל-c# יש פונקציות מיוחדות לאוספים, בניהם מתודות ייחודיות למערכים. נוכל להשתמש במתודה `Array.find(array, function)` שמקבלת מערך ומצביע לפונקציה שלפיו היא תחפש. אפשר ליצור פונקציה במיוחד או להשתמש בפונקציה למבדה שמבחינת סינטקס זהה כמעט לחלוטין לג'אווה. נשתמש במרחב השם `using System;` לשם כך. במקרה שלנו נדרוש שהמתודה תקבל כפרמטר את קובץ האודיו שאותו אנחנו רוצים למצוא מתוך המערך ובפונקציית למדה נחפש משתנה סאונד שהשם שלו זהה לשם המחרוזת אותה הביאו לנו:

```
public void Play(string name)
{
    Sound s = Array.Find(sounds, sound => sound.name == name);
    if (s == null)
    {
        Debug.LogWarning("sound" + name + "not found");
        return;
    }
    s.source.Play();
}
```

בשביל להשתמש ב-Audio Manager שיצרנו באובייקט משחק אחרים נוכל: א. במקרים בהם אנחנו אמורים להפעיל כמה אודיו-קליפים דרך אותו אובייקט נשמור משתנה עצם מסוג AudioManager ונפעיל את הפונקציה play שלו במידת הצורך. (באפשר גם להשתמש בפונקציה של unity שמביאה לנו אובייקט שמוצאת לנו אובייקט מסוים ואז נפעיל את play: `FindObjectOfType<AudioManager>().Play(...)`;



במדריך על Audio Manager השתמשנו בסרטון הבא:

<https://www.youtube.com/watch?v=60T43pvUyfy&t=679s>

של הערוץ Brackeys. הערוץ הוא ערוץ מעולה למפתחי משחקים ובייחוד לאנשים שיש להם ניסיון בתכנות. מומלץ בחום להעשרה בין בן התכנותי, ובין בן העיצובי של המשחקים. לפני שנעבור לפרק הבא אני רוצה להמליץ על האתר:

<https://freesound.org/>

האתר מספק מאגר עצום של צלילים ברישיון CC שניתן להוריד בחינם.

מבוא ל-UI:

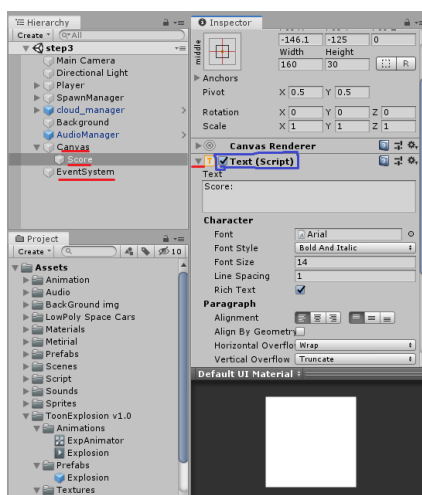
ממשק משתמש או User Interface הוא המרחב שבו מתנהלת התקשורת בין החלק התכנותי של המכונה לחלק הנשלט בידי אדם. בכלליות המטרה של עיצוב ופיתוח ממשק משתמש היא לייצר תקשורת קלה, יעילה ומהנה (User friendly) שמפעילה את המכונה בדרך הרצויה. בפיתוח משחקים המושג בד"כ מתקשר עם כל התצוגה על המסך שלא קשורה ישירות לאובייקטים הפועלים במשחק, למשל כפתורים- הכפתורים מתחברים למשחק במין שכבה מעל שמפעילה את המנגנון בפנים, לעומת זאת inputs מהמקלדת לא נחשבים ממשק משתמש כי המשתמש לא רואה אותם על המסך (רק את התוצאה שלהם). גם דברים שלא ניתן לשלוט עליהם אך מקלים על המשתמש כגון תצוגה של נקודות בצד המסך וכדו' נחשבים ל-UI כי הם משפרים את חווית המשתמש.


בunity השימוש ב-UI לא מסובך, למעשה יש כאלה שבונים אפליקציות שלמות דרך unity משום ש-unity פשוט מהרבה תוכנות עיצוביות ייעודיות לאפליקציות (אין צורך בשימוש בxml), מתאים את עצמו לכמה פלטפורמות בקלות, ונח מאוד לתכנת בו.

-Score text

בחלק הזה נתחיל לבנות את ה-UI הבסיסי. נרצה לבנות איזושהי גרפיקה שתציג את הנקודות שלנו במשחק, כמה חיים נשאר לנו ותפריט ראשי.

כדי שתהיה לנו אפשרות בכלל לבנות UI נצטרך קנבס (canvas) שעליו נציג. אם נלחץ על מקש ימני בחלון ההיררכיה ניתן לראות שיש את התת-אפשרות UI, שם ניתן לראות את כל האפשרויות שנכללות בתוך UI - טקסט, כפתורים וכדו'. כבסיס ניצור משתנה שיוציג לנו את כמות הניקוד שיש לנו. כל פעם שחיסלנו אויב נקבל נקודה. נלחץ מקש ימני-UI <-TEXT. כאשר הוספנו את האובייקט למסך נוסף לנו עוד אובייקט אב לחלון ההיררכיה שקוראים לו canvas, הוא למעשה כמין מסך ריק שעליו "מדביקים" את אובייקטי ה-UI שניצור. דבר נוסף שמופיע לנו במסך ההיררכיה הוא אובייקט מסוג Event System, או בקיצור ES, הוא אובייקט שמאפשר לנו לתקשר עם ה-UI באמצעות "מאורעות" (דוגמה למאורע היא לחיצה על כפתור). נשנה באינספקטור של הטקסט את שם האובייקט ל-score, ונפתח את הלשונית ברכיב טקסט (text(script) נראה שיש לנו אפשרות לשנות את הצבע הדיפולטיבי של הטקסט, את הפונט, הגודל וכו'.



ניתן לשנות את מיקום הטקסט על הקנבס באמצעות חצי ההזזה שנמצאים מצד שמאל למעלה. נשים לב שאם נקטין את גודל המסך של המשחק הטקסט לא יתאים לגבולות החדשים, זה משום שלא התאמנו את האובייקט למסך הראשי. אם נסתכל על האינספקטור של הטקסט נבחין כי יש לנו רכיב מסוג Rect Transform שהוא אחראי על התצוגה של האובייקט יחסית למסך המשחק, ברכיב שי לנו אייקון של ריבוע עם צלב באמצע:  הוא מסמן לנו היכן ממוקם כעת הטקסט יחסית למסך. אנחנו רוצים לשנות אותו ככה שיתאים למיקום שהצבנו לו- אם למשל הצבנו את הטקסט בפניה הימנית למעלה למטה נצטרך לשנות למצב המתאים לכך, בשביל לשנות את המצב הדיפולטיבי נלחץ על האייקון יפתח לנו חלון ה-Anchor Preset, ונבחר מתוכו את המצב שנראה לנו הכי מתאים. לצורך הדוגמא, אם אנחנו רוצים למקם את הטקסט בפניה הימנית העליונה נבחר ב- top ובתוכו נבחר ב-Right.

הצלחנו למקם את הטקסט באזור מסוים אך עדיין גודל הטקסט נשאר קבוע כשאנחנו מגדילים ומקטינים את המסך. כדי לתקן את זה נלך לאובייקט קנבס, בחלון האינספקטור שלו מופיע רכיב Canvas Scaler, כברירת מחדל הוא מסומן כ-Constant Pixel Size, אנחנו צריכים לשנות אתו ל- **Scale With Screen Size**, ועכשיו הוא מתאים את עצמו לגודל המסך של המשחק.

בינתיים הטקסט שלנו מוגדר להיות "score" בלי שינוי מהותי, כדי שנוכל לשנות אותו תוך כדי משחק נצטרך לתכנן כמה דברים: (1) לדאות שהשחקן ישמור ניקוד עבור כל אויב שהוא חיסל. (2) שהקנבס יעדכן את הטקסט לניקוד הנוכחי של השחקן. תחילה נוסיף לשחקן מתודה חדשה void AddScore() שתוסיף ניקוד למשתנה עצם _score מסוג int כפי שראינו כבר כמה פעמים במהלך המדריך. ניכנס לקוד של האויב: האויב צריך לקרוא למתודה AddScore() של השחקן בכל פעם שמחסלים אותו בין במגע עם השחקן ובין ע"י הלייזר. כבר ראינו כיצד לבקש אובייקט משחק ספציפי, אך אם נבקש בכל פעם שנפגע אויב את האובייקט player נבצע פעולה יקרה, לכן עדיף לנו לשמר אובייקט מסוג Player כמשתנה עצם ולא לחל אותו במתודה start() ואז מתי שהאויב מתנגש עם הלייזר הוא משתמש במתודה של האובייקט עצם שלו. מבחינת סינטקס זה יראה כך:

```
[SerializeField]
private Player _player;

void Start()
{
    _player = GameObject.Find("Player").GetComponent<Player>();
    . . .
}

. . .

private void OnTriggerEnter2D(Collider2D other)
{
    if(other.tag=="Player")
    {
        Destroy(this.gameObject);
        Player player= other.transform.GetComponent<Player>();
        player.Damage();
        player.AddScore();
    }
    if(other.tag=="Laser")
    {
        _player.AddScore();
        Destroy(this.gameObject);
    }
}
```

חזרה לקנבס, בכדי לעדכן אותו בכל פעם שעלה הניקוד נצטרך שיהיה לו איזשהו סקריפט שינחה אותו, נוסיף סקריפט UIManager וניכנס אליו. אם ננסה להוסיף משתנה מסוג טקסט כנראה שלא נמצא אותו המאגר של visual studio, וזאת משום שכדי להשתמש באובייקטים של UI נצטרך להשתמש במרחב שם מיוחד של unity לשם כך- UnityEngine.UI. אחר שהוספנו את מרחב השם אנחנו יכולים סוף סוף להוסיף את האובייקט טקסט שנקרא לו _score. נגרוור בunity את האובייקט טקסט score למקום המתאים באינספקטור של ה-canvas. נרצה לאתחל את האובייקט שלנו כך שהניקוד ההתחלתי שלנו יהיה אפס לכן במתודה start() נגדיר את המשתנה כך:



_score.text = "score" +0;

ובשביל שנוכל להתעדכן בזמן אמת עם ה-player נצטרך מתודה שתשנה את אותו משתנה עצם בזמן אמת:

```
public void ScoreView(int Point)
{
    _score.text = "score: " + Point;
}
```

נעבור לקוד של השחקן. נצטרך משתנה עצם מסוג UIManager כדי להפעיל את המתודה ScoreView. יש לשים לב שהרכיב UIManager נמצא בתוך האובייקט Canvas, לכן אם אנחנו רוצים לאתחל אותו נצרך למצוא את האובייקט Canvas וממנו לבקש את הרכיב UIManager:

```
[SerializeField]
private UIManager _UImanager;
...

void Start()
{
    . . .
    _score = 0;
    _UImanager = GameObject.Find("Canvas").GetComponent<UIManager>();
}
```

ועכשיו כל מה שנצטרך זה להפעיל את המתודה ScoreViewer ב-AddScore():

```
public void AddScore()
{
    _score+=10;
    _UImanager.ScoreView(_score);
}
```

-Live sprite

בדיוק כמו שעשינו עם הניקוד, אנחנו שואפים שיהיה לנו איזשהו אינדיקטור לכמה חיים נשארו לנו. אבל בניגוד לניקוד שהיה עשוי מטקסט, עכשיו אנחנו עובדים על אובייקט תמונה. נסיף אובייקט תמונה לקנבס במקש ימני -> UI -> image, נקרא לו בשם מתאים ונגרור את התמונה לתוך ה-image source באינספקטור של האובייקט שיצרנו הרגע. נמקם את הספריט במקום שנראה לנו מתאים בקנבס. אם הצורה של התמונה לא תואמת לאספקט של התמונה המקורית ניתן לשנות את זה ע"י בחירה ב-Preserve Aspect ברכיב image. כמובן שנשכפל את התמונה לפי כמות החיים של השחקן וכל תמונה נמקם אחת ליד התמונה האחרת.

נעברו לקוד: אנחנו צריכים ב-UI Manager שיהיה לנו מערך של התמונות (אובייקט מסוג Image), כך שלפי מצב החיים של השחקן המערך "יפעיל" את התמונות, למשל אם לשחקן שני חיים נשנה את ה-active של אחת מהתמונות הקיצוניות ל-false, לשם כך ניצור מתודה חדשה שאחראית לבטל את הספרייטים לפי החיים של השחקן. בתמונות של הקנבס, בניגוד לאובייקט-משחק אחרים, כדי לבטל את הפעולה של אותו רכיב משתמשים במשתנה enable של אותו אובייקט. מבחינת סינטקס זה יראה כך:

```
[SerializeField]
private Image[] _lives;
...

public void LifeScore(int lives)
{
    if (lives < 3)
    {
        _lives[lives].enabled = false;
    }
}
```



ובקוד של השחקן, היות וכבר יש לנו משתנה מסוג UI manager נוכל להשתמש בו במתודה damage() כך שמת'י שירידו לו נקודות הוא יעדכן את הקנבס להוריד תמונה אחת מתוך המערך:

```
public void Damage()
{
    life--;
    _UImanager.LifeScore(life);
    if(life<1)
    {
        StartCoroutine(Explosion());
    }
}
```

הערה: המתודה שיצרנו לוקחת בחשבון שלא תהיה תוספת חיים במהלך המשחק- אם ירדו לך חיים אז אתה נשאר עם אותם נקודות עד לפגיעה הבאה או סוף המשחק. במידה ואתם רוצים להוסיף power-up יש לממש את המתודה אחרת.

-Game over Text

הפסדנו את כל נקודות החיים שלנו ונגמר המשחק, לא יהיה יותר נחמד אילו יכולנו להוסיף את הטקסט הכל כך "כייפי" Game Over ? לאחר שכבר ראינו איך ליצור טקסט לניקוד יהיה לנו פשוט לפענח איך לעשות את זה גם בטקסט Game over- ניצור אובייקט טקסטGameOver חדש למסך ונכתוב את הטקסט הרצוי, נוסיף ל-UI Manager משתנה מסוג טקסט ונגרור אליו את האובייקטGameOver שיצרנו. ניצור מתודה שתציג את האובייקט על המסך, אם נרצה לעשות את זה בסטייל נדאג שהמתודה תפעיל מתודת coroutine שתפעיל את הטקסט כל שניה ותכבה אותו אחרי שניה, זה יצור אפקט RETRO. בקוד של השחקן נוסיף במתודה damage() להפעיל את אותה מתודה של ה-UI Manager שמפעילה את האפקט הפליקנינג:

```
public void Damage()
{
    life--;
    _UImanager.LifeScore(life);
    if(life<1)
    {
        StartCoroutine(Explosion());
        _UImanager.GameOverEnable();
    }
}
```

טעינה מחדש של סצנות-

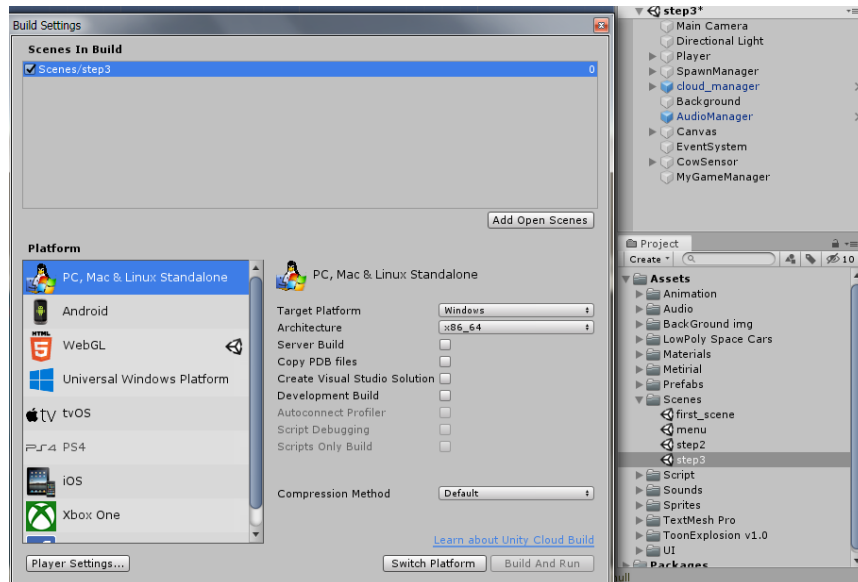
עד עכשיו המשחק שלנו אמור לרוץ כמו שצריך, הבעיה שהוא עדיין משחק חד פעמי- אחרי שהגענו לסוף המשחק אי אפשר לשחק בו שוב, אלא אם נצא ממנו ונתחיל אותו מחדש. אם נתייחס לסצנה כאל שלב במשחק, מה שבעצם נרצה שיקרה זה שנוכל להטעין מחדש את הסצנה למסך כך שכל ה"שלב" שהרגע שחקנו יתחיל מחדש אם הפסדנו. ראשית נדאג שעם סוף המשחק יופיע לנו טקסט שמודיע לנו שכדי להתחיל מחדש יש להקיש על אחד המקשים במקלדת, כבר ראינו מספיק פעמים כיצד לעשות את זה- נצטרך להוסיף אובייקט טקסט ל-UI Manager, ולמתודה שמפעילה את הטקסט GAME OVER שתפעיל גם את הכיתוב הנ"ל. עכשיו בשביל להטעין את הסצנה נצטרך איזשהו אובייקט ריק שינהל לנו את המעבר בין הסצנות.

ניצור אובייקט כזה עם השם My Game Manager או משהו בסגנון, נוסיף לו סקריפט עם שם זהה וניכנס אליו. אנחנו צריכים דבר ראשון איזשהו משתנה בוליאני עם ערך false שיהווה אינדיקטור שהמשחק נגמר, בהמשך נשנה את ערך המשתנה כאשר המשחק באמת נגמר ב-UI Manager. עכשיו במתודה update() נצטרך לבדוק שני דברים: (1) האם לחצנו על הכפתור או על המקש שאמור להתחיל את הסצנה מההתחלה. (2) האם למשתנה הבוליאני שמסמן את סוף המשחק יש ערך true, ואם אנחנו מקיימים את שני התנאים נרצה שתטען הסצנה מחדש. למזלנו ל-unity יש אובייקט מיוחד שאחראי לטעינת assets מהמשחק- Scene Manager וע"י המתודה



LoadScene(int SceneName) נוכל לטעון סצנות . לשם כך נצטרך להשתמש במרחב שם ייחודי לכך :
using UnityEngine.SceneManagement.

המתודה שצינו קודם מקבלת כפרמטר מספר אינטג'רי שמסמן את המספר של הסצנה, כדי לבדוק איזו סצנה יש לנו נצטרך להוסיף את הסצנה שלנו ל-Build Setting. כדי להגיע ל-Build Setting <-File נבחר Build Setting , ולבחור "Add open scene" אמורות להופיע לנו כל הסצנות שיצרנו עד עכשיו במשחק, נוכל לבחור מבניהן מה אנחנו רוצים שיבנו ואילו לא, וליד כל סצנה מופיע מצד ימין המספר שמייצג אותה, כמו כן ניתן פשוט לגרור את הסצנה מחלון הפרויקט לחלון ה-build setting וגם כאן יופיע מצד שמאל בקטן המספר שמייצג את הסצנה:



אופציה נוספת היא להשתמש בשם הסצנה דרך ה-scene manager : `SceneManager.GetActiveScene().buildIndex` . ייתן לנו את המספר סצנה שאנחנו עכשיו נמצאים בה.
מבחינת סינטקס במתודה `update()` של `MyGameManager` נרצה שזה יראה כך:

```
[SerializeField]
private bool _GameOver = false;
. . .
public void SetGameOver()
{
    _GameOver = true;
}
void Update()
{
    if (Input.GetKeyDown(KeyCode.R) && _GameOver)
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

במקרה שלנו הסצנה מוגדרת להיות המספר 0, אבל היא יכולה להשתנות בהתאם למספר הסצנות שאנחנו בונים למשחק שלנו. נשאר לנו עדיין להתאים בין ה-UI Manager לבין My Game Manager כדי שיעדכן אותו להפעיל את המתודה `SetGameOver()` כאשר המשחק נגמר. בעיקרון ניתן לשמור אובייקט עצם מסוג `MyGameManager` אבל בנתיים זה דיי מיותר לכן נסתפק בלמצוא אותו ואת המתודה עם `FindObjectOfType<MyGameManager>().SetGameOver();` . אותה נבצע ב-UI Manager איפה שמפעילים את הכיתוב `Game Over` .
נריץ את המשחק ונראה שכאשר באמת נפסלנו ואנחנו לוחצים על אותו כפתור הוא מאפס לנו את כל הערכים שהיו לנו לפני כן, זה משום שהטענה מחדש של הסצנה גם מאתחל את כל הערכים של האובייקטים של אותה הסצנה, יותר נכון לומר שכל האובייקטים שקשורים לסצנה נמחקים מהזיכרון ומאותחלים מחדש. בהמשך נראה כיצד ניתן לשמור על אובייקטים מסוימים במעבר בין הסצנות.



שמירה על Data במעבר בין סצנות-

כפי שכבר נזכרנו לראות- כאשר אנחנו מטענים את הסצנה מחדש כל המידע אודות האובייקטי משחק של הסצנה מתאפס. כדי שנוכל לשמר מידע בין הסצנות נצטרך איזשהו אובייקט ריק שיספוג לתוכו את כל המידע הרצוי על כל אובייקט שנרצה לשמור על נתוניו לסצנה הבאה. כמין מסד נתונים קטן שילווה את כל המשחק.

אנחנו צריכים שהאובייקט:

(1) יהיה נגיש לכל סקריפט במשחק.

(2) מאותחל רק פעם אחת

(3) ישמור מידע של כמה אובייקטים.

בחירה לוגית תהיה לממש אותו בתבנית ייצוב סינגלטון כדי שלא יוכלו ליצור עוד אינסטנסים שלו במהלך המשחק(רק אובייקט אחד שימש אותנו לאגור את data).

דבר ראשון שנעשה- ניצור אובייקט ריק חדש ונשנה את שמו למשהו שמתאים לתפקיד שלו, בסגנון של "Global Object".

נצמיד לאובייקט סקריפט עם שם זהה ונפתח אותו.

לפני שנמשיך, תזכורת לתבנית סינגלטון- בסינגלטון אנחנו יוצרים במחלקה משתנה עצם סטטי מסוג המחלקה ובודקים האם הוא כבר מאותחל, כלומר האם הוא null. במידה והוא לא מאותחל נאתחל אותו להיות this כלומר להיות המחלקה, אחרת, אם הוא כבר מאותחל, נשמיד את אותו אינסטנס. זה יבטיח לנו שיהיה רק אובייקט אחד כזה לאורך כל התוכנית(במקרה שלנו המשחק). בנוסף, כדי להגדיר את האובייקט כאחד שנשמר בין הסצנות נצטרך להשתמש בפונקציה: DontDestroyOnLoad(...). שכשמה כן היא- מקבלת כפרמטר איזשהו אובייקט ומגדירה אותו ככזה שנשמר בין סצנות.

היות ואנחנו רוצים שהאובייקט יאותחל לפני כולם (כי הוא מעדכן את שאר האובייקטים) נשתמש במתודה awake() במקום ב-start() כדי לאתחל אותו. ומעתה כאשר נרצה להשתמש באותו "Global Object", נשתמש באותו משתנה סטטי של המחלקה ששאר האובייקטים יקבלו ממנו מידע ויתעדכנו ממנו (במתודה או בדרך אחרת) כל תחילת סצנה או סוף סצנה. מבחינת סינטקס זה יראה כך:

```
public class GlobalObject : MonoBehaviour
{
    public static GlobalObject Instance;

    void Awake()
    {
        if (Instance == null)
        {
            DontDestroyOnLoad(gameObject);
            Instance = this;
        }
        else if (Instance != this)
        {
            Destroy(gameObject);
        }
    }
}
```

שאלה תכנותית: אנחנו שכבר מכירים דבר או שניים בתכנות בטח חושבים למה לא ניתן פשוט להשתמש באובייקט סטטי או משתנים סטטיים של מחלקה, הרי אובייקטים סטטיים משותפים לכל האינסטנסים של אותה סוג מחלקה. בניגוד למה שאנחנו עלולים לחשוב אינטואיטיבית, אובייקטים סטטיים לא "מתמידים" לאורך כל המשחק. היות וכל סקריפט (כל מחלקה) מחובר לאיזשהו אובייקט משחק, היא תיהרס ביחד עם האובייקט כאשר אנחנו נטעין סצנה חדשה. אפילו אם בסצנה החדשה אנחנו מייצרים את אותו אובייקט עם שדות ומשתנים סטטיים, משום שהרסנו את האובייקט הקודם כאילו לא נוצר אף פעם אובייקט כזה(כאילו זאת תוכנית חדשה). לכן, כדי לשמר נתונים בין סצנות, אנחנו חייבים להשתמש בתבנית סינגלטון ובמתודה DontDestroyOnLoad(...).

עכשיו כשכבר ראינו כיצד לדאוג שאובייקט ישמר במעבר בין הסצנות, נראה דוגמא על ה-AudioManager. אנחנו רוצים שבכל פעם שנאתחל את הסצנה המוזיקה שלנו לא תתאפס אלא תמשיך מאותו מקום. לשם כך נצטרך להגדיר אותו כ-סינגלטון להצמיד לו את מתודה DontDestroyOnLoad(...) כמו שכבר ראינו:



מבוא לפיתוח משחקי מחשב
ד"ר סגל הלוי דוד אראל

```
public class AudioManager : MonoBehaviour
{
    public Sound[] sounds;

    void Awake()
    {
        public static AudioManager Instance;

        void Awake()
        {
            if (Instance == null)
            {
                DontDestroyOnLoad(gameObject);
                Instance = this;
            }
            else
            {
                Destroy(gameObject);
                return; // So it won't do unnecessary commands
            }
        }

        foreach(Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.clip = s.clip;
            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.loop = s.loop;
        }
    }
}
```

-Main Menu

כשמדברים על תפריט ראשי מתכוונים לחלון הראשון או הסצנה הראשונה שאנחנו רואים כאשר אנחנו מפעילים את המשחק. שלב ראשון ניצור סצנה חדשה במשחק, נקרא לה Menu או משהו בסגנון וניכנס אליה(כמובן לשמור לפני את שאר הסצנות). כדי ליצור סצנה חדשה נעמוד על חלון הפרויקט(על התיקייה של הסצנות) <-Scene<-Create<- מקש ימני <-Scene<-Create<- כדי להוסיף רקע לסצנה נוכל לגרור תמונת רקע ואז נצטרך להתאים אותה למסך, או שנוכל להוסיף אובייקט פאנל למסך ולגרור אליו את התמונה. בחלון ההיררכיה נלחץ מקש ימני <-UI<-Panel, ונגרור ל-Source Image של הפאנל את התמונה הרצויה (יש לוודא שהתמונה היא מסוג (Sprite(2D and UI)). בשביל לשפר את הנראות אפשר לשנות את הצבע שלה. אם עדיין לא התאמנו את הקנבס כדי שיתאים את עצמו לגודל המסך זה הזמן לעשות את זה, **לתזכורת**. בתפריט נרצה שיהיו לנו כפתורים עם טקסט, וכפי שכבר נוכחנו לראות האובייקט טקסט של הקנבס לא נותן לנו יותר מידי אפשרויות למניפולציות, לשמחתנו החל מ-Unity2016 יש asset חדש שמאפשר משחק בטקסטים כמו צביעה הצללה ושיפור איכות אוטומטית-Text Mesh Pro. כדי ליצור אובייקט text-mesh נבחר מקש ימני מעל הקנבס<-UI<-Text Mesh Pro. Unity אמור להוריד לנו את ה-asset אוטומטית, במידה ולא אפשר להוריד אותו מה-Asset Store של unity בחינם. אחרי שכבר יש לנו text-mesh נתאים את גודל הטקסט למסך ונשנה את שמו לפונקציה אותה הוא אמור למלא, למשל לאופציה שמאתחלת את המשחק נקראה "PLAY" (נהוג לכתוב את התפריט באותיות גדולות), ליציאה-"QUIT" וכו'. בינתיים הטקסט עדיין צבוע לבן, אם נרצה להוסיף מעט הצללה נוכל ללכת לרכיב- underlay "נאפשר אותו"(enable) ונשנה את כפתורי ההזזה כאוות נפשנו. להוספת צבעים נלך לרכיב-color Gradient ובנחר את ארבעת הצבעים שיראו לנו הכי מתאימים לטקסט. אם נרצה שלמור על הגוונים שעשינו לטקסט שישמשו אותנו בהמשך לעוד אובייקטים כאלה נוכל ליצור asset של Color Gradient ולגרור אותו לכל אובייקט text-mesh חדש שניצור (ב-Gradient(Preset)). בשביל ליצור asset כזה נלחץ החלון הפרויקט מקש ימני<-Create<-Text Mesh Pro<-Color Gradient. כמובן שה-text-mesh אמור להיכנס לאיזשהו כפתור. ניצור כפתור חדש לקנבס, מקש ימני על הקנבס<-UI<-Button. בשביל להתאים את הכפתור נשנה את הצבע שלו לשחור(או צבע אחר שיתאים למסך) ולא נאפשר את הרכיב image שלו. נשים לב שלאובייקט כפתור יש אובייקט בן-טקסט, ברמת העיקרון אנחנו רוצים להחליף בין האובייקט טקסט הזה לבין ה-text-



mesh שייצרנו. נמחק את האובייקט טקסט ונגרור לכפתור את האובייקט שיצרנו. כדי להקל עלינו נשנה את שם הכפתור למילה של ה-text-mesh מייצג ואת שם ה-text-mesh נשנה לטקסט. נשנה את המיקום של הטקסט ביחס לכפתור באייקון ה-Rect Transform של ה-text-mesh, נלחץ Alt ונבחר באייקון הימני למטה, זה ידאג שהטקסט יותאם לכפתור. אם נריץ את הסצנה נראה שבאמת ניתן ללחוץ על האובייקט אך שום דבר לא קורה, אפילו אינדיקציה שלחצנו אין. נחזור לכפתור ונאפשר שוב את הרכיב image. נשנה ברכיב button את המשתנים הבאים:

- ב-Normal Color נשנה את הצבע אלפא (האות A בחלון שנפתח, אמור להיות האופציה האחרונה) ל-0, כלומר לצבע שחור.
- ה-normal color הוא המשתנה שמייצג את הצבע רקע של הכפתור השיגרה.
- ב-Highlighted Color נשנה את האלפא גם צבע כהה אך לא לגמרי שחור. ה-Highlighted מייצג את הכפתור כאשר עומדים עליו אבל לא לוחצים עליו.
- ב-Pressed Color נשנה את הגוון של האלפא להיות כהה, אך אפילו יותר בהיר מהצבע של ה-Highlighted. ה-Pressed, כשמו כן הוא, מייצג את האובייקט כשלוחצים עליו.

אם נריץ נראה שוב נראה שהתוצאה די יפה – כשלא לוחצים על הכפתור נראה שאין רקע לטקסט; כשעוברים מעל הכפתור אך לא לוחצים נראה שיש רקע כהה מסביב הטקסט, אך עדיין רקע שקוף; וכשלוחצים על הכפתור יש רקע אפילו יותר כהה מסביב. נשכפל את הכפתור ונשים את הכפתור השני מתחת לראשון. נניח הכפתור הראשון שעשינו הוא PLAY, אז נשנה את השם של הכפתור השני ל-QUIT למשל, ונשנה גם את הטקסט שלו בהתאם. בדוגמא הקרובה אנחנו נציג רק את שני הכפתורים האלו (QUIT ו-PLAY).

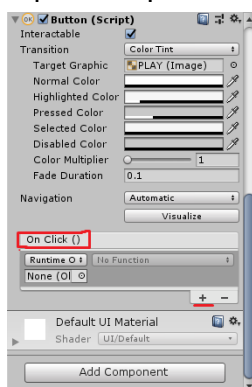
לפני שניתן פונקציונליות לכפתורים, כדי שנשים אותם תחת אובייקט ריק אחד מכמה סיבות:

- * אנחנו רוצים את האופציה להזיז אותם יחד, מבלי להצטרך לגרור אותם אחד אחרי השני.
- * יהיה לנו יותר קל אם יהיה לנו סקריפט אחד לשני הכפתורים, כפי שנראה בהמשך.
- * נוכל לשכפל את האובייקט הנ"ל וכך, אם נצטרך עוד סוגים שונים של תפריטים, פשוט להעתיק את אותו אובייקט ולשנות אותו בהתאם לתפריט החדש (נהפוך אותו ל-prefab ונדרוס אותו).

לכן ניצור אובייקט ריק חדש בתוך הקנבס, נקרא לו בשם מתאים למשל Game Menu ונגרור לתוכו את הכפתורים. נצמיד סקריפט לאובייקט עם שם זהה וניכנס אליו. ברמת העיקרון האובייקט Game Menu הוא אובייקט זמני במשחק שלא מתעדכן כל פריים, וכנראה לא ישמור בתוכו אובייקטים, אפוא ניתן למחוק לו את שתי המתודות הדיפולטיות שמגיעות איתו: Start() ו-Update(). אנחנו רוצים שתהיה מתודה מיוחדת לכל כפתור באובייקט: לכפתור PLAY אנחנו צריכים שתהיה מתודה שמטעינה את הסצנה הבאה (כלומר תחילת המשחק ממש). לפני שנכנס לקוד נחזור ל-unity וניכנס build setting, הסצנה הראשונה במשחק אמורה להיות הסצנה של ה-menu דווקא ולא הסצנה שעבדנו עליה עד עכשיו, לכן פשוט נמחק את הסצנות מה-build setting ונגרור את הסצנה של התפריט הראשי קודם ולאחר מכן את הסצנה של המשחק. נחזור לקוד. תזכורת: בכדי להשתמש בטעינת סצנות צריכים להשתמש במרחב שם מיוחד : using UnityEngine.SceneManagement, נבנה מתודה מיוחדת שטוענת את הסצנה של תחילת המשחק:

```
public void PlayGame()
{
    SceneManager.LoadScene(1);
}
```

ונחזור ל-unity שוב. אם נסתכל באינספקטור של הכפתור PLAY נראה שם את הרכיב on click(), הרכיב אחראי למאורע שלחצנו על הכפתור, הוא יכול להוסיף פונקציונליות לכפתור. נלחץ על הפלוס מצד ימין למטה ונוסיף עוד פונקציה לכפתור:



עם הוספת הפונקציה קפצו לנו שלושה מלבנים: runtime, no function, None . המלבן none מסמן האם יש אובייקט שאפשר להשתמש באחת(או יותר) המתודות שלו כאשר לוחצים על הכפתור. נגרור לשם את ה-Game Menu שלנו, כי אנחנו הולכים להשתמש במתודות שלו. במקום ה- no function נבחר -> Game Menu (שנוסף לאחר שגררנו אותו פנימה) -> PlayGame() . אם נריץ את המשחק נראה שאכן מתי שלוחצים על PLAY נטענת הסצנה הבאה כמתוכנן. נחזור שוב לסקריפט של ה-Game Menu, ונוסיף מתודה ליציאה מן המשחק. ניתן להשתמש בפונקציה: Application.Quit() אבל שימו לב שהיא לא תעבוד לנו כל זמן שאנחנו מריצים את המשחק דרך unity, אלא רק לאחר שנבנה את המשחק ממש כפי שנראה בהמשך. מבחינת סינטקס זה אמור להיראות כך:

```
public void QuitGame()
{
    Application.Quit();
}
```

אותו דבר שעשינו עם הכפתור PLAY, נעשה גם עם הכפתור QUIT, רק שנטעין את המתודה QuitGame() במקום. אם הפכנו את ה-Audio Manager לאובייקט don't destroy on load נוכל להכניס אותו לתפריט, כך נשמע את מנגינת הרקע עוד מהתפריט והיא תשמר לנו גם כשנתחיל את המשחק. לעוד מידע על האופציות שאפשר להוסיף ל-menu main:

<https://www.youtube.com/watch?v=Y0aYQrN1oYQ>

–Post processing

בחלק זה נעסוק בשימוש ב-post processing.

Post processing מאפשר לנו לשנות את האיות הצבעים של התמונות המשחק, להוסיף פילטרים וכדו'. כמין Photoshop למשחק. בשביל להשתמש ב post processing נצטרך להתקין אותו תחילה. ניכנס ל-windows package management ונחפש בשורת החיפוש ב post processing .

לאחר שהוספנו אותו לפרויקט שלנו, נצטרך להוסיף שכבה חדשה למצלמה שעליה נלביש את כל הפוסט פרוססינג: נלך לאינספקטור של אובייקט המצלמה -> Add Component -> Post-processor Layer. נצטרך לקבוע שכבה חדשה שעליה יעבוד הפוסט פרוססור, עדיפות על שכבה גבוהה יחסית. נלך מעל לאינספקטור, היכן שכתוב layer, ונוסיף שכבה חדשה. נקרא לה בשם post processing, ונגדיר את השכבה של ה-post processor layer להיות אותה שכבה שכרגע הגדרנו. ניצור אובייקט ריק חדש ונקרא לו post processor או משהו בסגנון. נגדיר את השכבה שלו באינספקטור להיות ה-post processing. נוסיף לו את הרכיב Post processing Volume ונגדיר אותו כ-Is Global, הגדרנו אותו כגלובלי כדי שההשפעה של האובייקט תהיה על כלל התמונות במשחק ולא על אובייקט ספציפי. לאחר שהגדרנו את הרכיב הפוסט פרוססור של המצלמה והאובייקט הפוסט פרוססור שלנו להיות באותה שכבה נוכל להוסיף אפקטים חזותיים. בהתחלה נצטרך ליצור פרופיל חדש לאובייקט הפוסט פרוססור. נבחר ב-new היכן שכתוב profile באינספקטור שך האובייקט. וכדי להוסיף אפקטים נבחר add effect. הדרך היעילה ביותר לעבוד עם הפוסט פרוססור היא ע"י ניסוי וטעייה, נסו את האפקטים השונים ונוכחו לדעת מה הכי מדבר אליכם. להלן סקירה כללית:

***Anti-aliasing**: מביא לגרפיקה מראה חלק יותר. טוב כאשר הקווים נראים משוננים או בעלי מראה מדורג (staircase appearance) זה קורה בד"כ כאשר למרנדר הגרפי אין רזולוציה גבוהה מספיק כדי ליצור קו ישר.

***Bloom**: יוצר שוליים או התרחבות של האור. מגביל או מוסיף אורות לתמונות. תורם לתאורה הכללית ולבהירות המצלמה.

***Chromatic Aberration**: מחקה את אפקט המצלמה בעולם האמיתי-כשהמצלמה שלה לא מצליחה לצרף את כל הצבעים לאותה נקודה. התוצאה- "שוליים" של צבעים לאורך גבולות התמונה שמפרידים חלקים כהים ובהירים שלה. נותן תחושה של שכירות לתמונה.

***color grading**: משנה או מתקן את הזוהר שunity מספק. דומה ליישום filter באינסטגרם.



ל-color grading יש שלשה מצבים:

low definition range - אידיאלי לפלטפורמות עם איכות ירודה.

High definition range - אידיאלי לפלטפורמות התומכות HDR rendering .

External - מאפשר לספק טקסטורות מתוכנות חיצוניות.

***Deferred fog:** מתאים למשחקי תלת-ממד שיש בהם עומק. מייצר כמין אפקט של ערפל- נותן לאובייקטים צבע אפרפר יותר בהתאם למרחק שלהם מהמצלמה.

***Depth of field:** אפרט לאחר עיבוד שמדמה את הפוקוס של עדשת המצלמה על אובייקט מסוים.

***Auto Exposure:** מדמה איך העיניים האנושיות מסתגלות לרמות שונות של חושך.

***Grain:** מחקה את האפקט של מצלמות בעולם האמתי מייצרות כאשר חלקיקים קטנטנים במסך גורמים לחספוס התמונה.

בדר"כ משתמשים באפקט במשחקי אימה, שמנסים שהתמונה לא תהיה "מושלמת".

***Motion blur:** מטשטשים אובייקטים מסוימים שנעים מהר יותר מהזמן חשיפה של המצלמה.

***Screen space reflection:** מיצר השתקפויות עדינות שמדמות משטח רטוב או שלולית.

***Vignette:** משאיר רק את מרכז התמונה מואר ומחשיך את הפינות.

