



תכנות ב-#c לרכיבים פיזיקליים של UNITY

דיברנו בגדול על המערכת הפיזיקלית של unity ועל רכיבים שימושיים שיש לה. אך עדיין לא דיברנו על איך ניתן לנתב את המערכת ע"י קוד: איך לגרום לאובייקט לנוע בצורה אקטיבית, כיצד להפעיל כוח על אובייקט מאובייקט אחר מבלי להשתמש ברכיבים חיצוניים. כמו כן בכלל לא דיברנו על מה הוא כוח פיזיקלי בטבע ואיך ניתן לחשב אותו. עתה נתמקד בכל הדברים האמורים לעיל, ובסוף נדגים כיצד ניתן ליישם אותם ע"י הצגת מיני משחק זריקות לסל.

-Collision detection

ל-unity יש שתי קבוצות של פונקציות לזיהוי התנגשות. הראשונה היא "Collisions" ומכילה שלוש מתודות: `onCollisionEnter` - המתודה נקראת כאשר הקולידר או הגוף קשיח של האובייקט בדיוק נוגע בקולידר או גוף קשיח אחר.

`OnCollisionStay` - נקראת כמו הפונקציה `update` אחת לכל פריים, כל זמן שהאובייקט נוגע בגוף קשיח או קולידר אחר.

`OnCollisionExit` - נקראת כאשר גוף קשיח או קולידר מפסיק לגעת באובייקט שלו יש את המתודה. כל אחת משלושת הפונקציות הנ"ל מקבלת אובייקט מסוג `Collision` כפרמטר. המחלקה `Collision` מכילה מידע על נקודות קשר, מהירות השפעה וכדו'.

אם לא משתמשים בפרמטר שמועבר בפונקציה, אין צורך לכתוב אותו בחתימת הפונקציה כדי למנוע חישובים מיותרים.

את הקבוצה השנייה כבר יצא לנו לראות בשיעורים הקודמים והיא "Triggers" שמכילה את המתודות `OnTriggerEnter`, `OnTriggerExit` ו-`OnTriggerStay` בדומה ל-"Collisions".

לעומת הקבוצה הראשונה, הפרמטר שמועבר לפונקציה בקבוצה זו הוא מסוג `Collider` ולא `Collision`. קולידר כפי שאנחנו יודעים מכיל שדות כמו `material`, `is trigger` וכדו', פחות שדות חישוביים.

ההבדל המהותי ביניהם הוא השימוש שלהן. בעוד הקבוצה הראשונה היא יותר חישובית: שימוש באלמנטים של ה-collision שנכנס לקולידר כדי לבצע פעולות חישוביות כגון הגדרת הדף מהמכה, חישוב נזק כתוצאה מהתנגשות וכדו'. הקבוצה השנייה יותר משמשת למימוש התנהגות לאחר התנגשות מבלי להתייחס לגורמים הפיזיקליים העוטפים את התרחיש. למשל לקחת מטבע או `power-up` נעדיף להשתמש ב-`trigger`, כי אין כאן באמת צורך חישובי ובכל זאת אנחנו צריכים את הפרמטר המועבר לפונקציה כדי לבצע שינויים באובייקט. הערה: בשתי הקבוצות יש צורך שבשני האובייקטים המתנגשים יהיו קולידרים. כמו כן לפחות אחד מהאובייקטים המתנגשים חייב להכיל `Rigidbody` אם לשניהם עובדים עם `Trigger`.

כמובן שבדו-ממד חתימת הפונקציה שונה `OnTriggerEnter` הופך ל-`OnTriggerEnter2D` והפרמטר הופך ל-`Collider2D` וכו'.

-Update & FixedUpdate

Update היא אחת מהפונקציות היותר משומשות ב-unity. היא נקראת אחד לפריים בכל סקריפט שמשתמשים בה. כמעט כל מחלקה או אובייקט שצריך להשתנות או להיות מותאם לסצנה בצורה רגילה נעשה דרך המתודה: תנועה של אובייקטים לא פיזיקליים, טיימרים פשוטים או זיהוי קלט הם רק חלק מהדוגמאות לדברים שניתן לעשות



עם המתודה.

Update לא נקראת בזמן סדור- אם פריים אחד לוקח יותר זמן לעבד מפריים אחר, אז הזמן בין קריאות למתודה יהיה שונה.

FixedUpdate דומה מאוד ל-update אך יש לה כמה שינויים מהותיים:

בעוד ש-update נקראת בזמן לא סדור, FixedUpdate דווקא נקראת בזמן מדויק, ולכן הפער בין קריאה לקריאה של המתודה לא תלוי בזמן עיבוד של הפריים.

בדיוק לאחר שהמתודה נקראת נעשים החישובים הפיזיים במנוע.

בתוך זה כל דבר שמכיל rigidbody, כלומר אובייקט פיזיקלי, צריך להתבצע ב-FixedUpdate ולא ב-Update.

זאת אחת הסיבות שהמנוע שומר על חישובים מדויקים של פעולות פיזיקליות.

הסקריפט הבא מקרין למסך את הזמנים בהם נקראת המתודה FixedUpdate לעומת Update, הצמידו אותם לאיזשהו אובייקט ריק כלשהו והריצו:

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

// GameObject.FixedUpdate example.
//
// Measure frame rate comparing FixedUpdate against Update.
// Show the rates every second.

public class ExampleScript : MonoBehaviour
{
    private float updateCount = 0;
    private float fixedUpdateCount = 0;
    private float updateUpdateCountPerSecond;
    private float updateFixedUpdateCountPerSecond;

    void Awake()
    {
        // Uncommenting this will cause framerate to drop to 10 frames per second.
        // This will mean that FixedUpdate is called more often than Update.
        //Application.targetFrameRate = 10;
        StartCoroutine(Loop());
    }

    // Increase the number of calls to Update.
    void Update()
    {
        updateCount += 1;
    }

    // Increase the number of calls to FixedUpdate.
    void FixedUpdate()
    {
        fixedUpdateCount += 1;
    }

    // Show the number of calls to both messages.
    void OnGUI()
    {
        GUIStyle fontSize = new GUIStyle(GUI.skin.GetStyle("label"));
        fontSize.fontSize = 24;
        fontSize.normal.textColor = Color.black;
        GUI.Label(new Rect(100, 0, 200, 50), "Update: " + updateUpdateCountPerSecond.ToString(), fontSize);
        GUI.Label(new Rect(100, 50, 200, 50), "FixedUpdate: " + updateFixedUpdateCountPerSecond.ToString(), fontSize);
    }

    // Update both CountsPerSecond values every second.
    IEnumerator Loop()
    {
        while (true)
        {
            yield return new WaitForSeconds(1);
            updateUpdateCountPerSecond = updateCount;
            updateFixedUpdateCountPerSecond = fixedUpdateCount;

            updateCount = 0;
        }
    }
}
```



```

    fixedUpdateCount = 0;
  }
}

```

– Forces

עתה נדבר קצת על כוחות (forces). כשמדברים על כוחות ב-unity מתכוונים בד"כ לדחיפה או משיכה של אובייקט. לדוגמה אדם ואבן גדולה. כאשר האדם דוחף את האבן הוא בעצם מפעיל כוח שגורם לה לנוע. כמה רחוק האבן תנוע תלוי בכמה חזק אותו אדם דחף את האבן, או ב-unity: מה ה-magnitude של הכוח, וכמה המסה (mass) של האבן (יש הבדל בין מסה לסתם משקל, מסה בהגדרה היא המידה בה הגוף מתנגד לשינוי במהירות). למעשה הראשון שהגדיר את זה בצורה פורמלית היה אייזיק ניוטון. ניוטון הגדיר שלושה [חוקי תנועה](#) המהווים הבסיס למכניקה הקלאסית. החוק הראשון קובע שכל גוף יתמיד במצבו, כל עוד אין כוחות חיצוניים שפועלים עליו. כלומר בהיעדר כוחות חיצוניים, גוף השרוי במנוחה יישאר במנוחה, וגוף נע יתמיד בתנועתו במהירות קבועה בקו ישר. החוק השני של ניוטון קובע שהאצה של אובייקט הוא תולדה של יחס הכוח מופעל עליו ומשקלו, ככל שהאובייקט כבד יותר ככה צריך להפעיל יותר כוח כדי להזיז אותו, או בניסוח פורמלי: $a = \frac{F}{M}$, כאשר a זה acceleration, F force, M mass. מייצג את המסה.

כמובן שהמשוואה גם לכיוון השני, אם נרצה לגלות כמה כוח הופעל על האובייקט נצטרך לגלות כמה רחוק הוא הגיע ולהכפיל במסה שלו. למשל נניח שהאבן נעה במהירות של $8 \frac{m}{s^2}$ (ה-m כאן מייצג מטר) כתוצאה מהדחיפה של

האדם, ונניח שמשקלה של האבן הוא 50kg, כמה כוח הופעל על האבן? התשובה $40kg \frac{m}{s^2} = (8 * 50)kg \frac{m}{s^2}$ נהוג לכתוב את הביטוי $kg \frac{m}{s^2}$ באות N או כ- f_n , השם המקצועי של הכוח הוא כוח ניוטון.

אם אותו אדם ימשיך לדחוף את האבן באותו הכוח, היא תעלה את מהירות שלה בהדרגה בהתאם לכוח המופעל עליה, המהירות היא בעצם המהירות ממנה התחיל האובייקט לנוע ועוד התאוצה של האובייקט כפול הזמן של התאוצה, או בניסוח פורמלי יותר: $v = v_0 + a * t$. במקרה שלנו האדם התחיל לדחוף את האבן כאשר היא נחה במקום, לכן המהירות ההתחלתית שלה היא 0, כלומר אם אותו אדם דוחף את האבן במשך שניה (באותה תאוצה), יוצא שהאבן תנוע במהירות: $0 + 8 \frac{m}{s^2} * 1s = 8 \frac{m}{s}$.

בשביל למצוא את המרחק אותו עברה האבן נצטרך להיזכר בנוסחה שלמדנו בתיכון: מהירות כפול זמן שווה לדרך, היות והמהירות של האובייקט משתנה מ- v_0 ל- $a*t$ (המהירות המקסימלית) נחפש את המהירות הממוצעת שעבר האובייקט בתהליך וכך נקבל שהמרחק אותו עבר האובייקט הוא מכפלת ממוצע המהירויות של האובייקט כפול הזמן,

כאשר האובייקט עומד במקום בתחילת הדרך נקבל שהדרך היא: $d = \frac{at+0}{2} * t = \frac{at^2}{2}$. ניקח את המקרה שלנו כדוגמה ונקבל: $\frac{1}{2} * 8 \frac{m}{s^2} * s^2 = 4m$.

לא קשה לנחש מה יקרה במקרה של שתי כוחות נגדיים שפועלים על אותו אובייקט, למשל נניח שכנגד האדם שדוחף את האבן מהדוגמה הקודמת יש אדם שדוחף אותה בכוח של 600N, הכוח שיופעל על האבן יהיה ההפרש בין שני

הכוחות: $400N - 600N = -200N$ והתאוצה של האובייקט תהיה $-4 \frac{m}{s^2} = \frac{F}{M} = -\frac{200N}{50kg}$.

מה שמוביל אותנו לחוק השלישי של ניוטון שקובע שכל גוף המפעיל כוח על גוף אחר, הגוף האחר יחזיר כוח שווה בעוצמתו לגוף הראשון. זה מסביר מדוע כאשר אנחנו דוחפים קיר או קופצים במקום אנחנו נהדפים אחורה- מיצרים כוח שחוזר אלינו, למרות שהגוף שאותו אנחנו דוחפים הוא סטטי ולא יכול להחזיר לנו כוח.

אם כך כיצד בכלל אותו אדם מצליח להזיז את האבן אם היא מחזירה לו את אותו הכוח? התשובה לכך היא משום שמופעלים כוחות נוספים על אותו אובייקט חוץ מהכוח שאותו אדם מפעיל על אבן (כמו חיכוך למשל).

לעומת זאת אם הוא היה דוחף את האבן בחלל כנראה שההדף כתוצאה מהדחיפה היה שווה, והוא היה עף אחורה.

>אפשר להתעמק יותר בכל הנוסחאות האחרות ($d = v_0 t + \frac{at^2}{2}$; $d = vt - \frac{at^2}{2}$; $v^2 = u^2 + 2ad$) שהן נגזרות

של הנוסחאות שלמדנו ולהראות איך הם בהם לידי ביטוי אח"כ בתוכנה <



–Rigidbody properties

תנועה של גופים-

עכשיו שאנחנו יודעים קצת מהתיאוריה שמאחורי הפיזיקה של תנועת גופים במרחב, נראה כיצד ניתן ליישם את זה בקוד. כפי שראינו בשיעורים הקודמים, כאשר אנחנו רוצים לגרום לאובייקט רגיל לנוע אנחנו משתמשים ברכיב translate ובמחלקה transform כדי לשנות את ווקטור הכיוון של האובייקט:

```
transform.Translate(new Vector3(x, y, z));
```

את העדכון בתזוזה נעשה במתודה Update, ואם נרצה שהתנועה תהיה מבוקרת נְתַנֶּה את התנועה בקלטים. הבעיה בתנועה כזאת שהיא לא פיזיקלית, האובייקט אומנם ינוע אבל לא יפעלו עליו חוקי הפיזיקה שלמדנו לעיל, זה מתבטא בכך שהוא לא יוכל לדחוף אובייקט אחר עם תאוצה וכוח. אם האובייקט שייצג את האדם מנסה לדחוף אובייקט אחר המייצג את האבן, אם האבן שוקלת הרבה יותר מהאדם אז לא משנה כמה חזק השחקן ילחץ על המקש שמזיז את הדמות היא לא תצליח להזיז את האבן. לעומת זאת אם הכוחות פיזיקליים, כלומר האדם ינוע תחת אילוצים פיזיקליים, יכול להיות שאם הדמות תיקח תאוצה מספיק גדולה הוא יוכל להניע את האבן במידה מסוימת. תנועה פיזיקלית בקוד מעט שונה אך עם זאת מאוד דומה ברעיון. בתנועה פיזיקלית אנחנו נפעיל כוח דרך הרכיב rigidbody שנע בכיוון ווקטור מוגדר כלשהו. את העדכון בתנועה נבצע במתודה הייעודית לעדכונים פיזיקליים-FixedUpdate, ואם נרצה שהתנועה תהיה מבוקרת נשתמש בקלט כדי לנתב את וקטור התנועה. נסתכל על תנועה פיזיקלית בunity. אם נרצה לשנות את הפוזיציה של האובייקט שנשנה את השדה הנקרא velocity, נוכל להשתמש במתודות כמו AddForce, AddExplosionForce ועוד. אם נרצה לשנות את הסיבוביות של האובייקט, נצטרך לשנות שדה הנקרא AngularVelocity, ונוכל להשתמש במתודות כמו AddTorque או AddRelativeTorque. המתודות המשותמשות ביותר הן AddForce (ו-AddTorque למקרה של סיבוב האובייקט). נראה הדגמה: פתחו פרויקט חדש ב-unity בגרסת התלת ממד. צרו אובייקט קובייה והוסיפו לו Rigidbody. הוסיפו לאובייקט סקריפט חדש באיזה שם שתבחרו. בשביל שנוכל להפעיל את המתודות לעיל נצטרך להשתמש ב-rigidbody של האובייקט לשם כך נשמור אובייקט עצם במחלקה מסוג rigidbody ובמתודה start נבקש את אותו רכיב באמצעות המתודה GetComponent. במתודה Update שנו את הפונקציה כפי שהיא מופיעה בחתימת הפונקציה ל-FixedUpdate. כדי לגרום לאובייקט לנוע בכיוון אחד נשתמש בפונקציה של rigidbody: AddForce. לפונקציה יש פרמטר אחד שנדרש ואחד אופציונלי. הפרמטר הראשון והנדרש הוא ווקטור הכיוון שאליו הכוח מכוון. לצורך הנוחות נניח שהאובייקט נע קדימה(רק על ציר ה-z) ולכן נשתמש ב-Vector3.forward, אם נרצה שהאובייקט גם ינוע במהירות מסוימת נכפיל את הווקטור בסקלר שייצג את המהירות, מומלץ לעשות משתנה מסוג float שייצג את המהירות ולהגדיר אותו כ-SerializedField, כדי שאח"כ נוכל לשנות את המהירות בזמן ריצת הסצנה. הפרמטר השני והאופציונלי הוא להגדיר את סוג הכוח שמפעילים, בשביל כך נצטרך להגדיר איזה ForceMode מפעילים. ForceMode היא מחלקה שמכילה כמה שדות: Acceleration- משמש לשינויים מתמשכים בפונקציונליות, אך לא מושפעים מהמסה של הגוף הקשיח. Force- דומה ל-Acceleration רק שהשינויים מושפעים מהמסה של האובייקט. Impulse- שינוי מידי במהירות הגוף(הגוף לא צובר תאוצה). כמו ב-Force השינוי מושפע מהמסה VelocityChange- כמו Impulse רק שהמסה לא משפיעה על התנועה. כברירת מחדל, אם לא הוספנו ForceMode כשלהו, המתודה תתבצע כ-ForceMode.Force. אם נרצה שהאובייקט גם יסתובב נשתמש באותה הדרך בדיוק במתודה AddTorque כמו שהשתמשנו ב-AddForce מבחינת קוד זה יראה כך:



```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AddForceScript : MonoBehaviour
{
    Rigidbody rb;
    [SerializeField]
    private float _speed = 10f;
    [SerializeField]
    private float _angularSpeed = 10f;

    private float _Velocity;
    private float AngularVelocity;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    private void FixedUpdate()
    {
        _Velocity = rb.velocity.magnitude;
        AngularVelocity = rb.angularVelocity.magnitude;
        rb.AddForce(Vector3.forward * _speed, ForceMode.Force);
        rb.AddTorque(Vector3.forward*_angularSpeed,ForceMode.Force);
    }

    void OnGUI()
    {
        GUIStyle fontSize = new GUIStyle(GUI.skin.GetStyle("label"));
        fontSize.fontSize = 22;
        fontSize.normal.textColor = Color.black;
        GUI.Label(new Rect(100, 0, 200, 50), "Speed: " + _Velocity, fontSize);
        GUI.Label(new Rect(100, 50, 200, 50), "AngularSpeed: " +AngularVelocity.ToString("F2"), fontSize);
    }
}

```

הפונקציה OnGUI היא פונקציה שמדפיסה ערכים שהיא מקבלת על המסך, ברעיון היא חוסכת את הצורך ליצור UI במיוחד בשביל הסצנה, במקרה שלנו היא נועדה כדי להציג לנו על המסך את ההבדלים בין המהירות הסיבובית של האובייקט למהירות התנועה שלו.

שמרו וחזרו לunity, היכנסו לאינספקטור שך הקובייה ובטלו את ה-Gravity כדי שהקובייה לא תיפול בזמן ריצת הסצנה, והריצו את המשחק.

שחקו באובייקט- שנו את המהירות של הקובייה, המסה שלה ה- Drag וכו', כמו כן שחקו ב-ForceMode בקוד, וראו כיצד השינויים הפיזיקליים מתבטאים בהרצת המשחק.

הערה: לדו ממד יש ForceMode אחר: ForceMode2D ולו יש רק שני מצבים אפשריים: Force ו-Impulse

עוד מאפיינים של גוף קשיח

אם נסתכל על המאפיינים של הגוף הקשיח נראה את המרכיבים הבאים:

use gravity, is Kinematic ... אלו שדות ב-rigidbody שניתן שלנות אותן אינטראקטיבית תוך כדי הרצת הסצנה, לעומתם ישנם שדות (או מתודות) שניתן לגשת אליהם רק ע"י קוד למשל IsSleeping ו-Velocity.

נסתכל על **RigidBody.IsKinematic**. כפי שכבר למדנו, כאשר גוף הוא קינמאטי לא משפעים עליו כוחות חיצוניים.

במילים אחרות, אם נגדיר את הגוף כקינמאטי אזי שהמנוע יתעלם מ-collision, joints וכוחות פיזיקאליים שמופעלים על אותו הגוף והוא ינוהל ע"י קוד במקום על ידי המערכת הפיזיקלית של unity.

לפעמים נרצה לשנות אובייקט שהתחיל כאובייקט פיזיקלי לאובייקט קינמאטי, למשל אויב שנפגע מלייזר ועכשיו הוא



"מת", יכול להיות שנרצה שאויב המחוסל יישאר על המסך אך לא יפריע למהלך המשחק, לכן ונעביר אותו למצב `isKinematic`.

(Rigidbody.IsSleeping) - היא מתודה שמחזירה ערך Boolean אם הגוף הקשיח במצב שינה.

אז מה זה בעצם מצב שינה? כאשר גוף קשיח נע מאוד לאט המנוע הפיזיקלי מניח שהוא עומד לעצור, ואז מצב "שינה" מופעל. האופטימיזציה הזאת חוסכת זמן עיבוד.

בד"כ נשתמש בזה כדי לזהות מתי אובייקט משחק הפסיק לנוע. למשל נניח שאנחנו רוצים שדמות תזרוק כדורים, אבל שהכדור השני לא יזרק לפני שהכדור הראשון עצר. נשתמש ב `isSleeping` של `rigidbody` של הכדור הראשון, ואם הוא `true` נזרוק את השני.

Rigidbody.Velocity - ווקטור המייצג מהירות וכיוון תנועה. ברוב המקרים לא מומלץ לשנות את הווקטור כי זה עלול להביא לתוצאות לא ריאליסטיות.

אך יש גם מקרים יוצאי דופן, למשל במשחקי יריות מגוף ראשון כאשר הדמות קופצת נרצה שינוי חד במהירות.

-Raycast

Raycast הוא תהליך של יריית קרן בלתי נראית מנקודה מסוימת בכיוון מוגדר, כדי לבדוק האם איזשהו קולידר במסלול של הקרן. בד"כ הקרן משמשת לירי במשחקי fps (first personal shooter), אך לא רק (בהמשך נראה דוגמא). הסינטקס של raycast הוא זה:

```
Physics.Raycast(Vector3 origin, Vector3 direction, RaycastHit hitInfo, float distance, int Layermask);
```

נראה מבלבל בהתחלה, אבל ברגע שמבינים מה כל חלק עושה הכל נראה הרבה יותר הגיוני.

תחילה `origin` - וקטור 3 שמייצג נקודה במרחב שממנה מתחיל הירי של הקרן, אך היות וקורדינטות הכיוון לא בהכרח עומדות מול היריה נצטרך גם וקטור שיכוון אותנו מה הכיוון של בקרן לכן צריך גם את `direction`. שני המשתנים הראשונים (הוקטורים) יוצרים את הקרן. למעשה יש אובייקט ב `unity` שיכול לקצר לנו את הסינטקס ולקבל את שני הוקטורים כפרמטר וליצור קרן:

```
Ray myRay = new Ray(Vector3 origin, Vector3 direction);
Physics.Raycast(myRay, RaycastHit hitInfo, float distance, int Layermask);
```

הפרמטר הבא בפונקצייה הוא משתנה מסוג `RaycastHit` ששומר את המידע על הקולידר שהתנגש עם הקרן. כשאנחנו שולחים את האובייקט לפונקציה, אנחנו בעצם דורשים ממנה שתחזיר לנו ערך לתוך אותו אובייקט, לכן נשלח אותו לפונקציה עם המילה שמורה `out`:

```
RaycastHit hitInfo;
Ray myRay = new Ray(Vector3 origin, Vector3 direction);
Physics.Raycast(myRay, out hitInfo, float distance, int Layermask);
```

אז מה זה בעצם `out`? כשאנו שולחים משתנים לפונקציה הם יכולים להיות משלושה סוגים:

`input/output` - בד"כ אנחנו שולחים מידע כ-`input` ואז לא נדרשת איזושהי מילה שמורה למשתנה. אם אנחנו משתמים במילה השמורה `out` אז אנחנו מחייבים את הפונקציה לתת ערך למשתנה, כלומר הפלט של הפונקציה נכנס למשתנה. אם אנחנו משתמשים במילה השמורה `ref` אז הפונקציה יכולה להחליף את הערך שהוכנס למשתנה לדוגמא:

```
class StaticFunction
{
    static bool DoMath(float a, ref float b, out float c)
    {
        c = a + b; //we initialize a new value to c using a and b, this value will be assigned to the variable.
        b -= a; //we change the value of b, the variable will be updated
        if (b >= 0.0f) return true;
    }
}
```



```

    else return false;
}
static void Main()
{
    float a = 1.0f;
    float b = 2.0f;
    float c; //This variable has no value assigned to it...
    if (DoMath(a, ref b, out c)) Console.WriteLine(a.ToString() + " " + b.ToString() + " " + c.ToString());
    else Console.WriteLine("B Is below zero!");
}
}

```

המשתנה $c = \text{input} - 1$, $b = \text{input/output}$, $a = \text{output}$

הערה: ההסבר על המילים השמורות לקוח מהדיון הבא בפורום של unity:

<https://forum.unity.com/threads/what-is-out-syntax-of-c-and-what-does-it-actually-do.404585>

חזרה לנושא המקורי: השני פרמטרים האחרונים בפונקציה אופציונליים: float distance מגדיר את אורך הקרן, כבירת מחדל הקרן תמשיך עד אינסוף. ו-layermask אמור להגדיר מאילו שכבות הקרן מתעלמת. אם נרצה שהקרן תתעלם משכבה ספציפית נוכל להשתמש בטריק המוכר: נניח השחקן נמצא בשכבה 8-

```

// bit shift the index of the layer to get a bit mask
var layerMask = 1 << 8;
// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast (transform.position, Vector3.forward, Mathf.Infinity, layerMask))
print("The ray hit the player");

```

מי שזוכר במ ++c האופרטור "<<" מייצג הזזה ביטית, למשל כשתכתבו $1 << 8$ בעצם הזזנו את הביט 1 מהמקום הראשון שמונה מקומות שמאלה: $...000000001 <- ...100000000$

המנוע מקציב לנו להשתמש ב-32 שכבות, כל שכבה מיוצגת ע"י ביט אחד מתוך 32 של משתנה מסוג int, לכן השכבה השמינית היא בעצם 1 במקום התשיעי (השכבה ה-0 היא ביט 1 במקום הראשון). בזמן ריצת המשחק לא נוכל לראות את הקרן, נרצה שתהיה לנו לפחות האפשרות לראות את הגיזמו שלה, גיזמו הם הסימונים הנלווים לאובייקטים כפי שהם מופיעים בחלון הסצנה, למשל אנחנו יכולים לראות גיזמו של קולידר של אובייקט (בצבע ירוק) בחלון הסצנה כאשר אנחנו בוחרים באותו אובייקט, ואנחנו לא נראה את הסימון בחלון המשחק. כדי לראות את הגיזמו נשתמש בפונקציה הבאה:

```
Debug.DrawRay(Vector3 origin, Vector3 direction*float distance, Color.red);
```

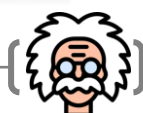
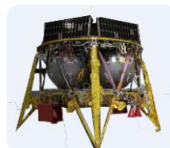
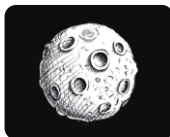
אנחנו בעצם מציירים קרן בחלון הסצנה מנקודת המקור דרך ווקטור כיוון מנומל (אם נרצה לראות את המרחק נכפיל במרחק), וניתן להגדיר את הצבע של הקרן, אך זה אופציונלי.

בדו ממד הסינטקס קצת שונה. הפונקציה תהיה במבנה דומה יחסית רק שבמקום Physics נשתמש במחלקה Physics2D והווקטורים יהיו ווקטור 2 ולא 3, כמו כן המשתנה RaycastHit יהפוך ל-RaycastHit2D, ולא נכניס אותו כפרמטר לפונקציה, אלא נכניס את הערך חזרה מהפונקציה אליו:

```
RaycastHit2D hit = Physics2D.Raycast(Vector2 origin, Vector2 direction, float distance);
```

אז עם המידע החדש בואו נעזור לחלילית [בראשית](#) לנחות על הירח (כמו שצריך הפעם) ניצור פרוייקט חדש ב-unity2D.

נמצא תמונה שתייצג את הרקע (חלל), תמונה שתייצג את הירח ותמונה שתייצג את החלילית בראשית. לאחר שיטות קל בגוגל מצאנו תמונה לייצוג הרקע והירח, עם החלילית היינו צריכים לעבוד קצת ב-krita כדי להסיר ממנה את הרקע.



כל אחת מהתמונות הגדרנו כ- `sprite(2D and UI)` והתאמנו אותן למסך. כמובן גם הגדרנו כל תמונה בשכבה מתאימה (ובמיקום נכון בשכבה). הוספנו לירח קולידר, ולחללית הוספנו קולידר `(boxCollider2D)` וגוף קשיח `2D` המטרה שלנו לנו לדמות נחיתה של החללית על הירח לכן נוסף גרביטציה לגוף הקשיח (אפשר להשתמש ב- `pointEffector2D` גם, לנוחות השתמשנו בגרביטציה). כמו כן נרצה שמתי החללית תנחת מהר מידי היא תתפוצץ, נגיד ומהר זה כאשר הכוח כתוצאה מהנחיתה הוא $1 < \text{magnitude}$). כדי להציג את זה בצורה ויזואלית יותר חיפשנו ב- `unityStore` איזשהי אנימציה של פיצוץ, גם כן לאחר שיטות קל מצאנו את האנימציה `ToonExplosion` אנימציה בחינם של פיצוץ. קוד: יצרנו סקריפט חדש לחללית. כדי לשלוט על הנחיתה נרצה שמתי שהחללית תתקרב לנחיתה על הירח (במרחק שאנחנו נגדיר) היא תפעיל קרן שתבדוק שהיא במרחק המוגדר, אם היא במרחק המוגדר נרצה שיפעל כוח כנגד הגרביטציה, לכן נגביר את ה- `linear drag` של הגוף הקשיח באותו זמן. בשביל שנוכל לבחון את המרחק ממנו יופעל ה- `drag` וכמות ה- `drag` נגדיר מראש שני משתנים שנוכל לשנות אותם מהאינספקטור, וכמובן את הגוף הקשיח שאותו נבקש בפונקציה `:start`:

```
[SerializeField]
private float _distance;
[SerializeField]
private float _addDrag = 4;
private Rigidbody2D rb;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
}
```

כדי ליצור את הקרן נשתמש בפונקציה `FixUpdate` כי אנחנו משתמשים בחישובים פיזיקליים. ניצור את הקרן שלנו ונבדוק עם המשתנה מסוג `RaycastHit2D` האם הקרן התנגשה עם אובייקט כלשהו המהלך הנחיתה (יש רק שני אובייקטים עם קולידר במשחק, החללית והירח), במידה והייתה התנגשות נגדיר את ה- `drag` על הגוף הקשיח כ- `_addDrag`. אנחנו גם רוצים לראות את הקרן אז בחלון הסצנה כדי לוודא שהיא באמת עובדת כמו שצריך, אז נשתמש בפונקציה שלמדנו `Debug.DrawRay()`:

```
private void FixedUpdate()
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, -Vector2.up, _distance);
    if (hit.collider != null)
    {
        rb.drag = _addDrag;
    }
    //To see the Gizmo of the ray
    Debug.DrawRay(transform.position, -Vector2.up * _distance, Color.red);
}
```

לסיום נפעיל את האנימציה במקרה שההתנגשות היתה חזקה מידי. לשם כך נצטרך להוסיף את האנימטור של הפיצוץ כמשתנה עצם של המחלקה. נגרור את האנימטור לאובייקט של בראשית, שיהיה אובייקט בן לחללית ונגדיר אותו כלא פעיל. נוסיף אובייקט עצם מסוג אובייקט משחק לקוד של החללית, ונגרור את האנימטור של הפיצוץ (שהוא האובייקט בן של החללית) למשתנה שמופיע באינספקטור של החללית בראשית:

```
[SerializeField]
private GameObject _explosion;
```

נשאר לנו רק לבדוק בהתנגשות האם התנגשות היתה בעוצמה גדולה מ-1, במידה וכן נצטרך להפעיל את האנימציה של הפיצוץ ולהרוס את האובייקט משחק של החללית:




```
private void OnCollisionEnter2D(Collision2D collision)
{
    print("The magnitude is" + rb.velocity.magnitude);

    if (rb.velocity.magnitude > 1)
    {
        StartCoroutine(Explosion());
    }
}

IEnumerator Explosion()
{
    _explosion.SetActive(true);
    yield return new WaitForSeconds(0.68f);
    Destroy(this.gameObject);
}
```

נריץ את הסצנה כמה פעמים ונשחק ב-drag וב-distance.

