תכנות ב-c# לרכיבים פיזיקליים של UNITY

תכונות של גוף קשיח - Rigidbody properties

תנועה של גופים

עכשיו, כשאנחנו 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 transform ובשיטה Translate כדי לשנות את ווקטור המיקום (position) של האובייקט:

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

את העדכון בתזוזה נעשה במתודה Update, ואם נרצה שהתנועה תהיה מבוקרת - נָתנֶה 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 מאד 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 מהירות האובייקט נשנה את השדה הנקרא velocity, נוכל להשתמש במתודות כמו AddForce, AddExplosionForce ועוד.
אם נרצה לשנות את המהירות הסיבובית של האובייקט, נצטרך לשנות שדה הנקרא AngularVelocity, ונוכל להשתמש במתודות כמו AddTorque או AddRelativeTorque (כוח זויתי).
המתודות המשומשות 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 הrigidbody: **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- SerializeField, 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- תאוצה קבועה (שינוי קבוע במהירות לשניה), לא מושפעת מהמסה של הגוף הקשיח.

Force- דומה לAcceleration, רק שהשינויים מושפעים מהמסה של הגוף.
Impulse- שינוי חד-פעמי 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 כלשהו, המתודה תתבצע כ-ForceMode.Force .

אם נרצה שהאובייקט גם יסתובב, נשתמש באותה הדרך בדיוק במתודה AddTorque כמו שהשתמשנו בAddForce .
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 של 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 בForceMode בקוד, וראו כיצד השינויים הפיזיקליים מתבטאים בהרצת המשחק.

שימו לב – המהירות הזויתית המירבית של העצם היא 7 – ניתן לשנות מספר זה ע"י השדה maxAngularVelocity.

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 בrigidbody שניתן לשנות 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 המנוע 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 שהאויב המחוסל יישאר על המסך אך לא יפריע למהלך המשחק, לכן נעביר אותו למצב isKinematic.

מצב נוסף שבו נרצה שגוף יהיה קינמאטי הוא כשאנחנו גוררים אותו בעזרת העכבר.

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 ייזרק לפני שהכדור הראשון עצר. נשתמש בisSleeping של הrigidbody 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.

יריית קרניים - Raycasting

Raycast הוא תהליך של יריית קרן בלתי נראית מנקודה מסוימת בכיוון מוגדר, כדי לבדוק האם יש איזשהו קוליידר במסלול של הקרן. בדר"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-** וקטור3 המייצג נקודה במרחב התלת-ממדי, שממנה מתחיל הירי של הקרן. אך היות וקורדינטות הכיוון לא בהכרח עומדות מול היריה, נצטרך גם וקטור שיכוון אותנו מה הכיוון של הקרן, לכן צריך גם את **direction**.
שני המשתנים הראשונים(הוקטורים) יוצרים את הקרן. למעשה יש אובייקט בunity שיכול לקצר לנו את הסינטקס ולקבל את שני הוקטורים כפרמטר וליצור קרן:

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

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

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

אז מה זה בעצם out? כשאנו שולחים משתנים לפונקציה הם יכולים להיות משלושה סוגים:
input, output ו- 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.Write(a.ToString() + " " + b.ToString() + " " + c.ToString());
        else Console.Write("B Is below zero!");
    }
}
```

המשתנה a=output , b=input/output ו- c=input

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: הפונקציה RayCast מחשבת אם הקרן מתנגשת במשהו או לא, ואם כן – היא מחזירה לנו את נתוני ההתנגשות בפרמטר הפלט `hitInfo`. בין השאר, אפשר להסתכל בשדה collider של hitInfo כדי לראות במה בדיוק הקרן התנגשה.

שני הפרמטרים האחרונים בפונקציה אופציונליים: float distance מגדיר את אורך הקרן, כברירת 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, out hitInfo, Mathf.Infinity, layerMask))
print("The ray hit the player");
```

מי שזוכר ב- c++ האופרטור "<<" מייצג הזזה ביטית, למשל כשתכתבנו 1<<8 בעצם הזזנו את הביט 1 מהמקום הראשון שמונה מקומות שמאלה: 00000001...-> 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 ה**גיזמו** שלה. גיזמו (gizmo) הם הסימונים הנלווים לאובייקטים כפי שהם מופיעים בחלון הסצנה, למשל אנחנו יכולים לראות גיזמו של קוליידר של אובייקט (בצבע ירוק) בחלון הסצנה כאשר אנחנו בוחרים באותו אובייקט, אבל אנחנו לא נראה את הסימון בחלון המשחק.

כדי לראות את הגיזמו נשתמש בפונקציה הבאה:

```
Debug.DrawRay(Vector3 origin, Vector3 direction_distane, Color.red);
```

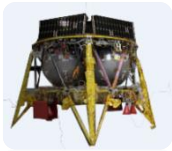
אנחנו בעצם מציירים קרן בחלון הסצנה מנקודת המקור דרך ווקטור כיוון מנורמל (אם נרצה לראות את המרחק נכפיל את וקטור הכיוון במרחק - distance), וניתן להגדיר את הצבע של הקרן, אך זה אופציונלי.

בדו ממד הסינטקס קצת שונה. הפונקציה תיהיה במבנה דומה יחסית רק שבמקום Physics נשתמש במחלקה Physics2D והווקטורים יהיו ווקטור2 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 לחללית 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 החללית בראשית.
לאחר שיטוט 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 קוליידר, ולחללית הוספנו קוליידר(boxCollider2D) וגוף קשיח 2D.
המטרה שלנו לנו לדמות נחיתה של החללית על הירח לכן נוסיף גרביטציה לגוף הקשיח (אפשר להשתמש ב-pointEffector2D גם, לנוחות השתמשנו בגרביטציה).

כמו כן נרצה שמתי שהחללית תנחת מהר מידי היא תתפוצץ, נגיד ומהר זה כאשר מהירות הנחיתה היא מעל 1 (magnitude>1). כדי להציג את זה בצורה ויזואלית יותר חיפשנו ב-unityStore איזשהי אנימציה של פיצוץ, גם כן לאחר שיטוט 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 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 **FixedUpdate** 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 קוליידר במשחק, החללית והירח), במידה והייתה התנגשות נגדיר את drag על הגוף הקשיח כ- _addDrag .

אנחנו גם רוצים לראות 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 האובייקט 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.