LEGO UNITY TANFOLYAM ÖSSZEFOGLALÓ

1. C# alapok

```
//Gyerek az osztály neve, amely a Szülő osztályból származik
//Származik = megörökli az attribútumait, függvényeit,
//de bővítheti, módosíthatja viselkedését
public class Gyerek : Szülő {
    //Privát attribútumot csak saját osztály látja
    private int szam = 1;
    //protected attribútumot saját osztály és gyerekei látják
    //C#-ban string is alapértelmezett típus
    protected string szoveg;
    //public mindenhonnan elérhető
    //Függvény felépítése: láthatóság visszatérési_érték név(paraméterlista)
    public void Fgv(int a, string b){
        double a = 1.0;
        //Egy osztályból példány létrehozása: new
        //Nincsenek pointerek, nem kell felszabadítanunk, megcsinálja magának
        Gyerek gyerek = new Gyerek();
    }
}
```

2. Unity alapfogalmak

- Scene: egy pálya, vagy akár a főmenü, GameObject-eket tartalmaz.
- GameObject: A játék és a világ építőelemei, a játékostól és ellenségektől a pályai falain át még a UI elemei is GameObject-ek. Egy GameObject egy vagy több Component-et tartalmazhat.
- Component/Script: A világunk vezérlő programjai, saját életciklussal, ezek segítségével tudjuk programozni a játékunkat. Script a saját készítésű Component. Nagyon sok hasznos Component előre el van nekünk készítve.
    - Minden GameObject-nek egy alap komponense van, ez a Transform. Ez tartalmazza a pozícióját, forgását, valamint méretét, amik mind 3 dimenziós vektorok.

3. Saját Scriptek írása

```
//Ha GameObject-re szeretnénk helyezni szkriptünket, akkor a MonoBehaviour-ből
//kell örökölnünk
public class CustomScript : MonoBehaviour {
    //Ha Inspector-ban szeretnénk egy értéket állítani, akkor vagy publikussá
    //tesszük
    public int x;
    //Vagy eléírjuk a [SerializeField] jelölést (ez szebb, de public is jó)
    [SerializeField] private bool y;

    void Start(){
        //Minden scriptből elérhető:
        //A jelenlegi GameObject, amin fut a Script
        GameObject g = this.gameObject;
        //A jelenlegi GameObject transformja
        Transform t = this.gameObject.transform;
        //Bármelyik komponens (most Komponens nevű) lekérhető a szkriptel
        //azonos GameObject-ről
        Komponens komp = this.GetComponent<Komponens>();
        //Másik GameObjectről ez ugyanúgy megtehető:
        GameObject other; //Most tegyük fel, hogy valahonnan megkaptuk
        Transform ot = other.transform;
        Komponens ok = other.GetComponent<Komponens>();
    }
}
```

Ha lekérjük egy GameObject egy komponensét, ott nagyon sok beállítást láthatunk és módosíthatunk, valamint egyéb, Inspectorban nem elérhető függvényeit is meghívhatjuk. A dokumentum tartalmazza a vett komponensek fő beállításait és függvényeit.

Ha saját szkriptünket kérjük le GetComponent-el, akkor annak is elérhetjük a public változóit és függvényeit, így tudunk saját szkriptjeink közt kommunikálni.

Transform egy speciális komponens, minden GameObject kötelezően tartalmazza, ezért ez elérhető GetComponent nélkül is (lásd: az előző példakód). Transform fő értékei (legyen Transform t;):

- t.position: a pozíciója, Vector3
- t.rotation: a forgása, Quaternion
- t.localScale: a mérete, Vector3
- t.right: a jobbra mutató egységmátrix (X tengely +1)
- t.up: a felfelé mutató egységmátrix (Y tengely +1)
- t.forward: az előre mutató egységmátrix (Z tengely +1)
- t.parent: a jelenlegi GameObject szülő GameObject-jének Transformja
- t.Translate(Vector3 merre): megadott irányba mozgatja a GameObject-et

4. Életciklus

A MonoBehaviour még egy dolgot tesz a szkriptjeinkkel; lehetővé teszi, hogy életciklus függvényeket használjunk. Ezek egyszerű, privát, void visszatérésű függvények, amelyeket a Unity automatikusan meghív a játék megfelelő szakaszán.

```
//Főbb életciklus függvények:
void Awake(){
    //Az első dolog ami lefut, inicializáláshoz használatos
}
void Start() {
    //A játék elindításakor, vagy az új elem létrejöttekor
}
void Update(){
    //Minden képkockán lefut, FPS függő
    Time.deltaTime; //Ez visszaadja az előző képkocka óta eltelt időt
    //Ezzel beszorozva a mozgásokat FPS függetlenné tudjuk tenni
}
void FixedUpdate() {
    //FPS független Update, futási gyakorisága a fizika motorhoz van kötve
    //Másodpercenkénti hívása projekt szinten állítható
    //Főként fizikához szokás használni
}
void OnDestroy(){
    //Az elem megsemmisülése előtti utolsó lefutó függvény
}
```

5. Debug

Debug segítségével egyszerű üzeneteket írhatunk ki a konzolra, amely csak szerkesztőben (vagy direkt ezt tartalmazó build-ben) látható. 3 szintje van:

```
Debug.Log("információ");
Debug.LogWarning("figyelmeztetés");
Debug.LogError("hiba"); //Ez esetén beállítható, hogy leálljon
```

6. Prefabok

Ha egy GameObject-et sokszor akarunk a Scene-en elhelyezni (pl. azonos építőelemekből álló pálya), vagy esetleg játékmenet közben akarunk létrehozni (pl. lövedék), akkor érdemes azt Prefabbá alakítani. Ez egyszerűen megtehető, ha lehúzzuk a GameObject-et a Project nézetbe. Ekkor az összes komponensével, azok jelenlegi beállításaival, valamint gyerekeivel elmentődik Assetként, innen visszahúzva a Scene-re vagy Hierarchy-ba annak pontos mását kapjuk. Prefab használatának két nagy előnye van:

1. A Prefab beállításait a Scene-re lehelyezett példányok követik, így elég egy helyen módosítani
2. Prefab átadható GameObject típusú [SerializeField]-nek, így bármelyik szkriptnek át tudjuk adni mint változó, ami később tud példányt létrehozni belőle futási időben (pl. lövésnél)

```
//Létrehozás futási időben:
//mit GameObject típusú, általában [SerializeField]-ben kapjuk
//hol egy Vector3 rotation, általában transform.position
//forgas: egy Quaternion (nem néztük), általában transform.rotation
//Visszaadja GameObject-ként a létrehozott elemet
GameObject uj = Instantiate(mit, hol, forgas);
//Pozíció és forgás elhagyható, ekkor a másolt GameObject helyén hozza létre
GameObject uj = Instantiate(mit);
//Mindkét változatnál megatható, hogy hierarchiában ki legyen felette
GameObject uj = Instantiate(mit, szulo);
```

7. Scene váltás

Ahhoz, hogy egy Scene megnyitható legyen kódból, ahhoz azt hozzá kell adni a Build Scene listájához, ezt Scene listájához, ezt File→Build Settings menüben tehetjük meg. Ekkor a Scene megnyitása vagy név, vagy itteni id alapján működik:

```
//Név alapján
SceneManager.LoadScene("név");
//Azonosító alapján
SceneManager.LoadScene(1);
```

8. Input

a) Egyszerű input

Egyszerű Inputnak neveztük azt, amikor egyszerű logikai értékek szintjén tudjuk lekérdezni a gombnyomás egy-egy állapotát. Az ezt lekérdező függvények a következő formátumot követik:

Input.Get[Mit][Szakasz]([gomb]);

A Mit rész arra utal, hogy a beviteli eszközök közül miről szeretnénk lekérdezni. Két változatát néztük:

1. Mouse: egér gombjai
2. Key: billentyűzet gombjai

A Szakasz rész arra utal, hogy a gombnyomás melyik szakaszát szeretnénk lekérdezni

1. Down: akkor igaz, ha éppen lenyomtuk a gombot, egy Update loop idejéig
2. [üres]: akkor igaz, ameddig nyomva tartjuk a gombot

3. Up: akkor igaz, ha éppen felengedtük a gombot, egy Update loop idejéig

Gomb:

1. Mouse esetén: számmal jelöljük, 0 = bal gomb, 1 = jobb gomb, 2 = görgő lenyomása
2. Key esetén: KeyCode.[gomb] formátum, ez általánosságban:
    a. Betű gombok: nagy betű, pl. KeyCode.A
    b. Szám gombok: Alpha[szám], pl. Alpha3
    c. Speciális gombok: minden szava nagy betűvel kezdve, egybe, pl. KeyCode.LeftControl

Például:

```
if (Input.GetKey(KeyCode.W)){
    //Ameddig le van nyomva a W
}
if (Input.GetKeyUp(KeyCode.AltGr)){
    //Amint felengedjük az AltGr-t
}
if (Input.GetMouseButtonDown(1)){
    //Amint lenyomjuk a jobb egérgombot
}
```

b) Input Rendszer

A Unity Input Rendszer az egyszerű input újragondolása, pár újítást hoz be az eddigihez képest:

- A játékban használt gombokat elnevezhetjük, alternatív gombot adhatunk ugyanahhoz az akcióhoz, ekkor név alapján az Input.GetButton…(név) formában olvashatjuk a lenyomást
- Ugyanazt az akciót, csak különböző irányba végző (pl. előre-hátra) gombokat összerendelhetünk egy Axis-ra (tengelyre), ekkor egyik lesz a pozitív, egyik a negatív irány, ennek ugyanúgy nevet és mindkét irányhoz alternatív gombot adhatunk. Ennek kódból kiolvasása nem igaz/hamis értéket ad vissza, hanem [-1;1] tartománybelit attól függően, hogy melyik irányhoz tartozó gomb van lenyomva. Sok irány előre el van nekünk készítve:
    - Vertical: előre-hátra mozgás, W vagy Felfele Nyíl = +1, S vagy Lefele nyíl = -1
    - Horizontal: jobbra-balra mozgás, D vagy Jobbra Nyíl = +1, A vagy Balra nyíl = -1
    - Mouse X: egér vízszintes mozgása
    - Mouse Y: egér függőleges mozgása
    - Mouse ScrollWheel: egér görgő görgetése

Ezek módosítása: Edit→Project Settings→Input Manager

Például:

```
if (Input.GetButtonDown("Jump")){
    //"Jump" nevű gomb lenyomásakor, alapértelmezetten szóköz
}
float elore = Input.GetAxis("Vertical"); //Előre/hátra mozgás
```

9. Fizika

A fizikai szimulációhoz két dologra van szükségünk:

a) Rigidbody

Ez az elem szükséges ahhoz, hogy a fizikai kölcsönhatások mozgathassák egy GameObject-ünket. Inspector beállításai a fizikai attribútumait módosítják:

- Mass: tömeg
- (Angular) Drag: milyen (forgási) lassítás hasson rá
- Use Gravity: gravitáció hasson-e rá, ennek mértéke és iránya projekt szinten állítható
- IsKinematic: ezt bekapcsolva fizikai kölcsönhatások nem mozgathatják a testet, de másokra hathat
- Constraints: letiltható az X, Y, Z tengely menti mozgás, valamint ezek körüli forgás a fizikai kölcsönhatásokból (egyesével)

Egyetlen fontos függvénye az erőhatás alkalmazása. Erőt 4 féle módon lehet hozzáadni:

ForceMode típusai	Időtől függő (dt)	Időtől független
Tömegtől függő (m)	Force V += f * dt / m	Impulse V += f / m
Tömegtől független	Acceleration V += f * dt	VelocityChange V += f

Időfüggő: egy erőt folyamatosan adunk hozzá, pl. rakéta kilövése

Időfüggetlen: egyszeri, hirtelen erő, pl. lövedék kilövése

Tömegfüggő: különböző tömegű Rigidbody-kra is szeretnénk használni, dupla tömegűre fele olyan hatásos legyen, pl. különböző méretű lövedékek

Tömegfüggetlen: egységesen hat különböző tömegű Rigidbody-kra, pl gravitáció

Példa használat:

```
//ero: Vector3, ilyen irányú és mértékű erő
rigidbody.AddForce(ero, ForceMode.?);
//hol: Vector3, ebből a pozícióból érkező erőt szimuláljon
rigidbody.AddForceAtPosition(er, hol, ForceMode.?);
//ForceMode megadása nélkül ForceMode.Force-ot használ
```

b) Collider, Trigger

Ezek tulajdonképpen a testek „fizikai” felületét adják meg („hitbox”), ezen keresztül tudnak egymással ütközni, és az ütközést fizikailag szimulálni. Egy ütközéshez szükséges mindkét elemen Collider, valamint az egyiken Rigidbody. A Trigger rendszer ennek az alternatív felhasználása, ha az ütközéshez szükséges elemek közül az egyik, és csak egyik Collider-én az IsTrigger be van kapcsolva. Ekkor nem történik ütközés, helyette a Trigger egy „zónát” jelöl ki, amelybe belépést tudjuk érzékelni. Ekkor ugyanúgy szükséges legalább az egyik elemen Rigidbody. Mind a Collider, mind a Trigger különböző alakú lehet (a Trigger-hez Collidert használunk, amin bekapcsoljuk az IsTrigger-t):

- BoxCollider – téglatest alakú
- SphereCollider – gömb alakú
- CapsuleCollider – kapszula alakú
- MeshCollider – tetszőleges modell alakú (általában importált modellek saját modellével egyező)

A Collider és Trigger kevés meghívható függvényt tartalmaz, inkább az életciklus függvényeihez hasonlóan „függvény-alakokat” határoz meg, amelyeket ha megírunk a szkriptünkben, akkor azokat a Unity automatikusan meghívja. Ezek közül fő függvények:

```
void OnCollisionEnter(Collision other) {
    //Ez a Collider ütközése egy másikkal
    //A kapott paraméter tárolja az ütkézés adatait, és a másik elemet
    GameObject o = other.gameObject;
}
void OnTriggerEnter(Collider other) {
    //Ezen Trigger-be belép egy másik Collider
    //A kapott paraméter maga a belépő Collider komponens
    //Ennek GameObject-je:
    GameObject o = other.gameObject;
}
//Enter mindig a belépést, ütközés elejét kezeli le
//Ugyanezen függvények Exit változata a kilépést, ütközés végét
```

10. GameObject-ek csoportosítása

A GameObject-ek két féle módon csoportosíthatóak:

a) Tag

A Tag a GameObject célját, csoportját tudja meghatározni (pl. Enemy az ellenségeket, Bullet a játékos által kilőtt lövedékeken), egy GameObject csak egybe tartozhat, de egy Tag bármennyin rajta lehet. Bármennyi tetszőleges Tag-et hozzáadhatunk a projekthez. Kódból két fő dolgot tudunk velük csinálni:

```
if (gameObject.tag == "Keresett Tag"){
    //Lekérjük, ellenőrizzük egy elem Tag-jét
}
//Keresünk egy adott Tag-gel rendelkező elemet
GameObject go = GameObject.FindWithTag("Keresett Tag");
//Ha több is van ugyanazzal a Tag-gel, véletlen, melyiket adja!
```

b) Layer

A Tag-gel ellentétben a Layer nem csak logikai rendszerezését segíti az elemeknek, hanem meghatározza, hogy melyik Layer-ű fizikai elemek melyikekkel tudnak ütközni. Például beállíthatjuk, hogy egy „Labda” Layer-ű elem mindennel tudjon ütközni, kivéve másik „Labda” Layer-űekkel. Ezek beállítása egy (fél) mátrixban történik, ez a Edit→Project Settings→Physics→Layer Collision Matrix helyen található. Ezen kívül a kamera is beállítható úgy, hogy csak adott rétegeket jelenítsen meg. A Layer-ek száma korlátozott, összesen 32 létezik (0-31 idexelve), ebből Unity előre meghatározott 5 darabot (0-2, 4-5), valamint a 31-est maga az Editor használ(hat)ja, így ezt érdemes csak végső esetben használni.

11. Raycast

Raycast segítségével sugarat tudunk vetni egy adott pontból egy adott irányba, és vizsgálni, hogy melyik elem van ezen a „sugáron”. 4 dologra van hozzá szükségünk:

1. Ray: egy sugár, kezdőpontból és irányból hozzuk létre (mindkettő Vector3)
2. out RayCastHit: a találati információkat ebbe a változóba fogja kiírni (visszatérési értéke az, hogy volt-e találat), az out-ot elé kell írni, mert ez az a C# nyelvi elem, amely engedi a függvénynek, hogy módosítsa a változót (mint C/C++-ban a mutató szerinti átadás)
3. float MaxDistance: maximum távolság, amennyit mehet a sugár, mielőtt nem találtnak vesszük, alapértelmezett értéke a végtelen
4. int LayerMask: azokat a rétegeket határozza meg, amelyeket eltalálhatja a sugár, alapértelmezetten mindegyik

A függvény visszatérési értéke, hogy volt-e találat, például:

```
//Gyakori feladat: képernyőre kattintáskor egér alatti ele, ehhez ray:
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
//Különben tetszőleges kiindulással és iránnyal is létrehozható:
Ray ray2 = new Ray(kezdes, irany);
RaycastHit hit;
float distance = 100f;
LayerMask reteg = LayerMask.GetMask("Labda");
if (Physics.Raycast(ray, out hit, distance, reteg)){
    //Ha volt találat, akkor igazzal tér vissza
    //Ekkor eltalált GameObject:
    GameObject eltalalt = hit.collider.gameObject;
}
```

12. ProBuilder

Egy Unity Plugin, amely segítségével modellezni, pályákat építeni tudunk egyszerűen egyből Unity-ben. Lényege, hogy alakzatokat tudunk létrehozni (téglatest, henger, … valamint tetszőleges sokszög), majd ezek módosíthatóak a csúcsok, élek, oldalak mozgatásával, forgatásával és méretezésével. Ehhez nem néztünk kódokat (egyébként kódból is módosítható valamennyire egy alakzat), helyette az elérhető Tool-okat kipróbálni, valamint szükség esetén videókat nézni a használatáról.