Belépő a tudás közösségébe

Informatika szakköri segédanyag

Játékfejlesztés Unity-ben

Bende Imre

Szerkesztő: Zsakó László

A kiadvány „A felsőoktatásba bekerülést elősegítő készs
lesztő és kommunikációs programok megvalósítása, val
az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-
16-2017-006) című pályázat keretében készült 2018-ban.

Eötvös Loránd Tudományegyetem
Informatikai Kar

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

A szakköri anyag segítségével szeretném bemutatni a Unity-t, egy videójáték-motort. A szoftver ismertetésén, főbb funkcióinak bemutatásán kívül egy konkrét Labirintus játék fejlesztése során lesz lehetőség elsajátítani a modern játékfejlesztés alapjait. A leírás nem teljeskörű, nem tartalmaz minden feature, szolgáltatás részletes leírását. A célom kifejezetten ennek a konkrét játék készítésének a bemutatása úgy, hogy a felmerülő elemeket elmagyarázzam, részletezzem (a felhasznált függvények dokumentációját linkekkel jelöltem, így mélyebb betekintésre is szert lehet tenni). Az anyag végeztével aztán az olvasó/diák elsajátítja a szoftver alapjait, motivációt szerez, ami alapján a későbbiekben aztán maga is készíthet hasonló játékokat, valamint további tutorialok[1], dokumentációk[2] felhasználásával bővítheti tudását a témakörben (amik értelmezése egyszerűbbé válik az első játék megalkotása után). A projekt egésze tömörítve feltöltve megtalálható, így első áttekintésnél, vagy probléma esetén érdemes rátekinteni.

Unity

A Unity egy videójáték-motor, mely környezetet ad kétdimenziós, illetve háromdimenziós játékok, animációk, szimulációk készítéséhez. Az eszköz segítségével többfajta platformra is fejleszthetünk, többek között: Microsoft Windows, Mac OS, Xbox, Playstation, Wii, iPhone, Android. Az elsődleges nyelvnek a C#-ot tekinti, így mi is a scripteket abban fogjuk írni (régebbi verziókban más nyelvek is szerepeltek, de ezek a 2017-es verzióval kikerültek). Ezekhez bármilyen fejlesztői eszközt lehet használni alapértelmezetten a Visual Studiot ajánlott, de akár egy egyszerű Notepad++ is megfelel a célnak. A Unity legfőbb erőssége a támogatottság: részletes, naprakész dokumentáció, rengeteg elérhető tutorial, videóanyag, valamint folyamatosan jönnek ki újabb frissítések, amelyekhez bővülő szolgálatatások is társulnak (például: Ads, Analytics, Cloud Build, Multiplayer, Performance Reporting). A szoftver mindenki számára ingyenesen elérhető[3] és átlátható felülettel rendelkezik, így a kezdők is használhatják. Érdekességként: Unity-vel készült több közismert számítógépes játék is, mint például a Cuphead, a Hollow Knight és az Escape from Tarkov is. Háromdimenziós modelleket azonban nem tudunk létrehozni benne, így ahhoz egy másik szoftver használata szükséges[4] (az általunk elkészített játékhoz erre nem lesz szükség, alapmodelleket használunk, valamint az Asset Store-ból is letölthető rengeteg ingyenes modell).

Telepítés

A telepítés előtt regisztrálnunk kell a Unity honlapján (ingyenes). Ezután az operációs rendszerünknek megfelelő installert letöltjük, amely az elindítása után tölti le, majd telepíti az alkalmazás egészét (ez internetkapcsolattól függően hosszabb folyamat is lehet).

A fejlesztői felület

Miután sikerült feltelepíteni a szoftvert fontos, hogy megismerkedjünk annak felületével [1. ábra], főbb komponenseivel, amiket aztán a későbbiekben használni is fogunk.

Többféle layoutot is be lehet állítani, én a „2 by 3”-at fogom bemutatni, illetve a fejlesztés során ezt fogom használni (ezt a Window → Layouts menüpontban, vagy a jobb felső sarokban lehet módosítani).

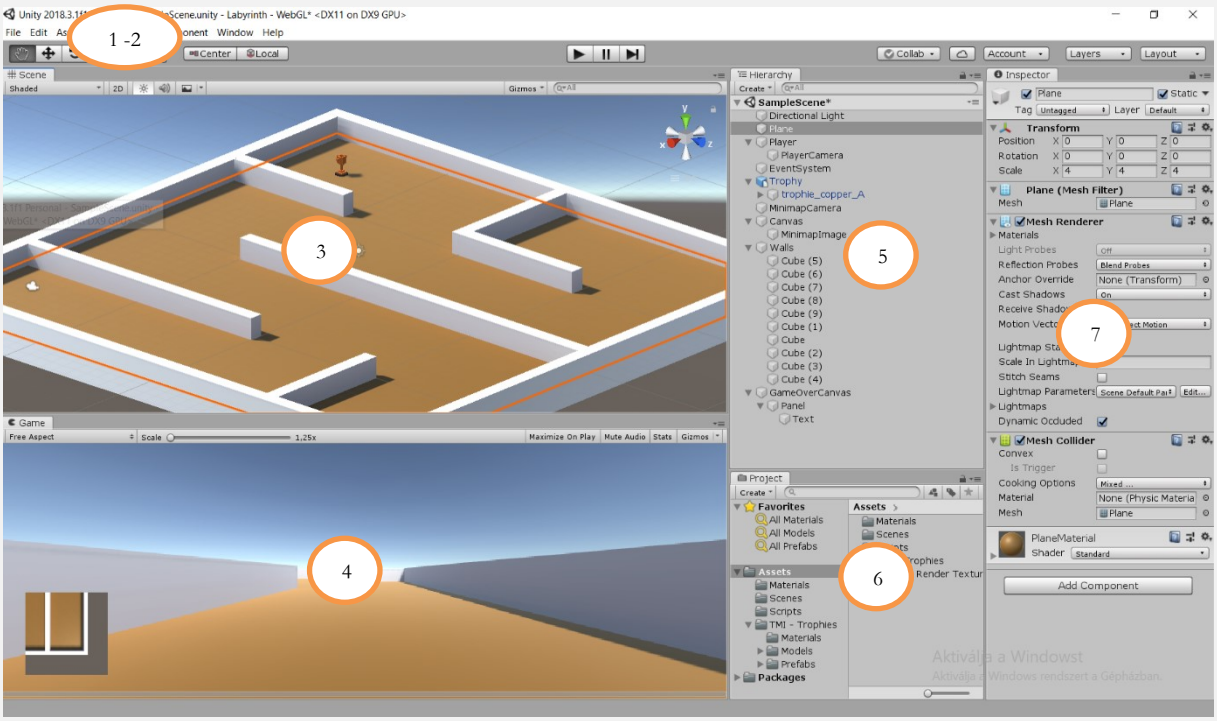
---

[1] Unity honlapján elérhető tutorialok (videók, elérhető anyagokkal): https://unity3d.com/learn/tutorials

[2] Unity dokumentációja: https://docs.unity3d.com/Manual/index.html

[3] Az ingyenes (Personal) verziót évi $100.000 bevétel alatt használhatjuk.

[4] Például: Adobe Director, Blender



1. ábra Unity felülete

1. Az összes funkció a menüsorból elérhető.
2. Alatta vannak a főbb, gyakran használt funkciók: tárgyak mozgatása, méretének módosítása kézzel, játék/animáció lejátszása, megállítása.
3. A színpad, ahol térben láthatjuk a játékunkat, valamint a benne lévő tárgyakat.
4. A játék előnézete, valamint indításkor ezen keresztül próbálhatjuk ki azt.
5. Az objektumok hierarchiája. Itt adhatunk hozzá, illetve láthatjuk a már létrehozott komponenseket, objektumokat, amiknek tulajdonságait aztán a későbbiekben módosíthatjuk.
6. A projekt mappastruktúrája, valamint annak tartalma.
7. A komponensek tulajdonságai, attribútumai találhatók meg, illetve módosíthatók itt.

A játék

Tervezés

Egy játék készítésének elkezdése előtt fontos, hogy megtervezzük milyen főbb komponenseket, motívumokat szeretnénk megjelentetni abban, mi legyen a játék célja, valamint hogyan lehessen ezeket elérni, megvalósítani.

A szakköri anyag során egy labirintus játék készítését, fejlesztését szeretném bemutatni, így felsorolásszerűen ebben az esetben a következők lennének az előző kérdésekre adott válaszok:

- Legyen a labirintusnak egy pályája: egy alappal, valamint falakkal (amiken keresztül nem tudunk átmenni, illetve nem látunk át rajtuk)!
- Legyen egy játékos, akit irányítunk, valamint az ő szemszögéből láthatjuk a dolgokat!
    - Kurzorral irányíthatjuk a mozgását.
    - Az egérrel irányíthatjuk merre nézzen.

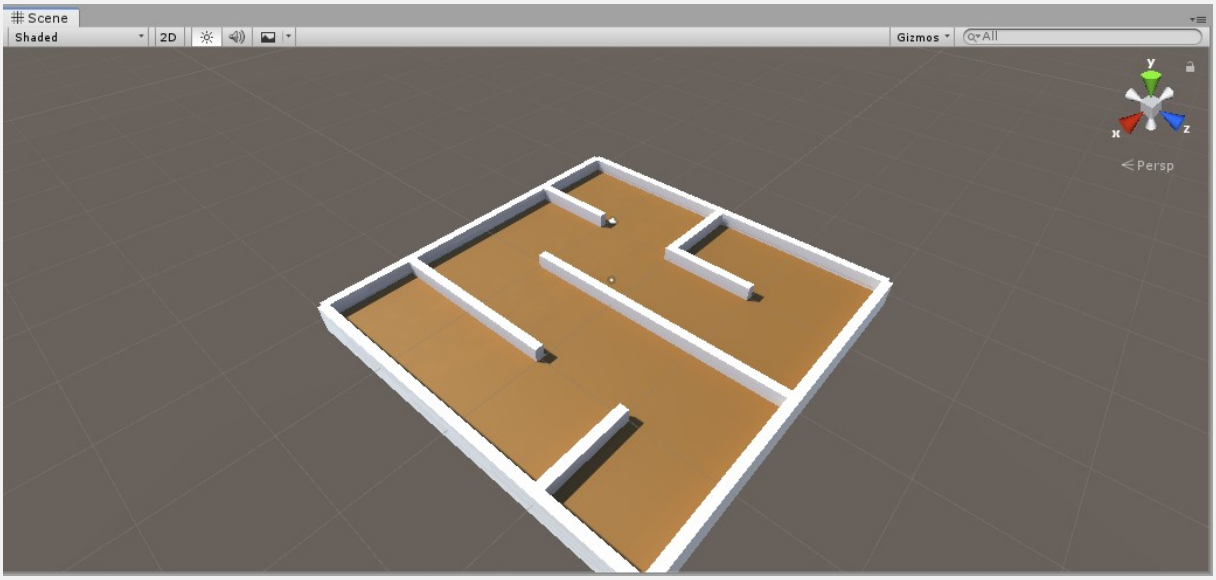
- A cél a labirintusban elhelyezett trófea megtalálása. Ha ezt megtaláltuk, akkor vége a játéknak és ezt a felhasználó felé is jelezük.
    - Legyen lehetőség ebben az esetben a játék újrakezdésére, vagy befejezésére!
- A cél eléréséhez egy minimap (jelen esetben felülnézeti kamera) adjon útmutatást, mely jelzi a játékos helyzetét, a trófeát, valamint a falakat! Ez a funkció csak egy betű (például: M betű) lenyomásával legyen elérhető (ne minden esetben látszódjon, így nehezebbé téve a játékot)!

A pálya

Legelső lépés az üres projekt létrejötte[5] után, hogy megalkossuk az alapot és a falakat (kezdetben egy kameránk és egy fényforrásunk van a vásznon). A GameObject → 3D Object-ben találhatjuk meg azokat az alap objektumokat, amikkel ezeket felépíthetjük. Az alap egy Plane lesz amire aztán építhetjük a falakat, illetve ezen tud a játékos is majd úgy mozogni, hogy ne essen le. Ennek elhelyezkedését és méretét lehet egérrel a bal felső sarokban található ikonokkal, illetve az Inspectorban kézzel is módosítani. A további ilyen jellegű módosításoknál igyekszem leírni, hogy én pontosan milyen paramétereket is használtam útmutatást adva ezáltal, de érdemes minél szabadabban, kreatívabban önállóan megoldani, hogy az eredmény egyedibb legyen és a diákok is többet fejlődhessenek. Én a Plane méretét az X és Z tengely megnégyszerezésével növeltem meg (Scale).

A falakat Cube-okkal hoztam létre. A szélső falakat a Plane síkjában 40-szeresére nagyítottam (X/Z tengelyen, attól függ melyik oldalnál), hogy befogják a pálya szélét, magasságukat pedig 3-szorosára (Y tengely), hogy ne lehessen átlátni majd felettük. A köztes falakat saját igény (méret, pozíció) szerint lehet beállítani, a magasságot érdemes mindenhol ugyanakkorára állítani.

Ezek után a vászon hasonlóképpen fog kinézni:



2. ábra Vászon kinézete az alap és a falak elkészülte után

[5] A projekt neve után már csak a 3D-s template-t kell kiválasztani a kezdőablakban, hogy ezt létrehozhassuk.

A játékos

Következő lépésben a játék elkészítéséhez szükségünk van egy irányítható játékosra. Jelen esetben a játékos FPS-szerűen (First Person Shooter) lesz irányítható, ez azt jelenti, hogy az ő szemszögéből láthatunk mindent.

Az első lépés a játékos létrehozása, mely két elemből áll: egy játék objektumból, illetve azon belül egy kamerából (ezen keresztül látjuk aztán majd a játékot). Az elsőhöz létrehozunk egy üres GameObject-et (*Player*), amihez hozzáadjuk a jobb oldali menüben (Add Component) a Character Controllert (ezzel kapcsoljuk hozzá az alap fizikai tulajdonságokat: mozgathatóság, ütközés). A kamerát az előzőhez hasonlóan egy Camera hozzáadásával tehetjük meg (*PlayerCamera*), amit az előbb létrehozott objektum alá rendelünk (szimplán ráhúzzuk az objektumok hierarchiájában a kamerát a játékosra). Mivel az alapból generált kamerát a későbbiekben nem fogjuk használni, így akár azt törölhetjük is.

Második lépés a játékos irányításának implementálása. A játékos a kurzor gombokkal lesz irányítható, így a már létrehozott „*Player*" objektumunkhoz létre kell hozni egy C# scriptet (mappastruktúrában érdemes létrehozni egy Scripts mappát, majd azon belül jobb egérkattintással létre tudunk hozni egy új scriptet, ezután a script objektumhoz húzásával tudjuk hozzárendelni azt) (*PlayerMove.cs*). Az Awake függvény hasonlóan működik, mint egy konstruktor, tehát amikor a script betöltődik, akkor fut le ez az eljárás majd. Betöltéskor ezen függvényen belül a komponenst változóként kezeljük, amelyet aztán a későbbiekben így módosíthatunk (ebben az esetben mozgathatunk). Public változó esetén a komponensben (Inspectorban) állíthatjuk a változó értékét, referenciáját (persze a scriptben adhatunk neki kezdőértéket is, így a komponensnél azt nem kell kötelezően beállítani). Private esetén csak a scripten belül érhető el a változó.

```
private CharacterController charController;

private void Awake()
{
  charController = GetComponent<CharacterController>();
}
```

A mozgás két összetevőből áll: melyik irányban, mekkora sebességgel hajtjuk végre azt. Az Input.getAxis függvénnyel a lenyomott irányt kapjuk vissza egy [-1,1] intervallumban, a sebességet pedig paraméterként fogjuk meghatározni a komponensben (jelen esetben 6-os kezdőértéket adunk neki). Így a két vektor összegének megfelelően a játékos mozgatását a SimpleMove függvénnyel tesszük meg. Alapértelmezetten a kurzor gombok, valamint a WASD gombok vannak beállítva a horizontális, vertikális tengelyek inputjaira, azonban ezek az Edit → Project Settings → Input menüpontban módosíthatók.

```
public float movementSpeed = 6;

private void PlayerMovement()
{
  float horizInput = Input.GetAxis(horizontalInputName) * movement-
Speed;
  float vertInput = Input.GetAxis(verticalInputName) * movement-
Speed;

  Vector3 forwardMovement = transform.forward * vertInput;
  Vector3 rightMovement = transform.right * horizInput;

  charController.SimpleMove(forwardMovement + rightMovement);
}
```

Ezt az előbb létrehozott eljárást az Update-be tesszük, mely minden frame generálásakor le fog futni.

```
private void Update()
{
  PlayerMovement();
}
```

A harmadik lépés, hogy hűek maradjuk az FPS-es irányításhoz az az, hogy a kamerát egérrel lehessen mozgatni. Ehhez az előzőhez hasonló módon, a kamerához (*PlayerCamera*) hozzáadott scriptre lesz szükségünk (*PlayerLook.cs*). Elsőként inicializáljuk a változókat (játékos hozzáadása, egér érzékenység):

```
public float mouseSensitivity = 150;
public Transform playerBody;
private float xAxisClamp;

private void Awake()
{
  Cursor.lockState = CursorLockMode.Locked;
  xAxisClamp = 0.0f;
}
```

Az Update-ben pedig a CameraRotation függvénnyel folyamatosan állítjuk a kamerát és a játékost:

```
private void CameraRotation()
{
  float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity *
Time.deltaTime;
  float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity *
Time.deltaTime;

  xAxisClamp += mouseY;
  if(xAxisClamp > 90.0f)
  {
    xAxisClamp = 90.0f;
    mouseY = 0.0f;
    ClampXAxisRotationToValue(270.0f);
  }
  else if (xAxisClamp < -90.0f)
  {
    xAxisClamp = -90.0f;
    mouseY = 0.0f;
    ClampXAxisRotationToValue(90.0f);
  }

  transform.Rotate(Vector3.left * mouseY);
  playerBody.Rotate(Vector3.up * mouseX);
}

private void ClampXAxisRotationToValue(float value)
{
  Vector3 eulerRotation = transform.eulerAngles;
  eulerRotation.x = value;
  transform.eulerAngles = eulerRotation;
}
```

A végén egy dolog maradt már csak hátra, hogy a Player Look scriptjének Player Body tagjához hozzárendeljük a *Player* objektumot (az objektumot behúzzuk az Inspectorba található mezőbe). Így elkészült a játékos, akit mi magunk tudunk irányítani majd a játék során.

Minimap

Hogy a játékosnak legyen segítsége (és ne csak vakon próbálja megtalálni a trófeát) létrehozunk egy minitérképet, amely a bal alsó sarokban mutatja a labirintust, a trófeát, valamint a játékos helyzetét felülnézetben.

Egy kamerán (*MinimapCamera*) keresztül tesszük meg mindezt, amit majd a későbbiekben a képernyőre (Canvas-re) kiteszünk (GameObject → Camera). A kamerát a (15, 20, -15)-ös pozícióra helyeztem, majd az X tengely körül elforgattam 90°-kal, így kezdőhelyzetben fentről nézz le a játékosra. Canvas-t a GameObject → UI menüponton belül fogunk tudni létrehozni, ez ad majd keretet a felületen megjelenő UI elemeknek (jelen esetben minimapnek). Ezen belül (ugyanabban a menüpontban elérhető) egy Raw Image-en fogjuk majd megjeleníteni a kamera képét (*MinimapImage*). Ahhoz, hogy a kamera és a kép össze legyen kapcsolva szükségünk lesz egy Render Texture-re. Ezt a projekt struktúrájában tudjuk létrehozni (hasonlóan a scriptekhez) jobb egér Create → Render Texture. Ezután a kamera Target Texture-jébe behúzzuk az előbb létrehozott objektumot, majd a Raw Image Texture-jébe is. Így már a felületen is látszik a kamera képe. Ahhoz, hogy a képernyő megfelelő helyén legyen, ahhoz a kép Rect Transform attribútumain kell állítani, amelyben bal alsó sarokba tesszük azt, majd finomításokkal helyére igazítjuk (Pos X, Pos Y: 80, mérete 100x100).

A kamera képe látszik, azonban a játékos mozgatása után az nem fog a minimap kamera látószögében maradni, így érdemes lenne egy scripttel megoldani, hogy kövesse a kamera a játékost (*PlayerFollow.cs*):

```
public Transform target;
Vector3 offset;

void Start()
{
  offset = transform.position - target.position;
}

void Update()
{
  Vector3 targetCamPos = target.position + offset;
  transform.position = targetCamPos;
}
```

Az Inspectorban a script Target inputjához behúzva a *Player* objektumot, már azt is tudni fogja, hogy minek a mozgását kell majd követni.

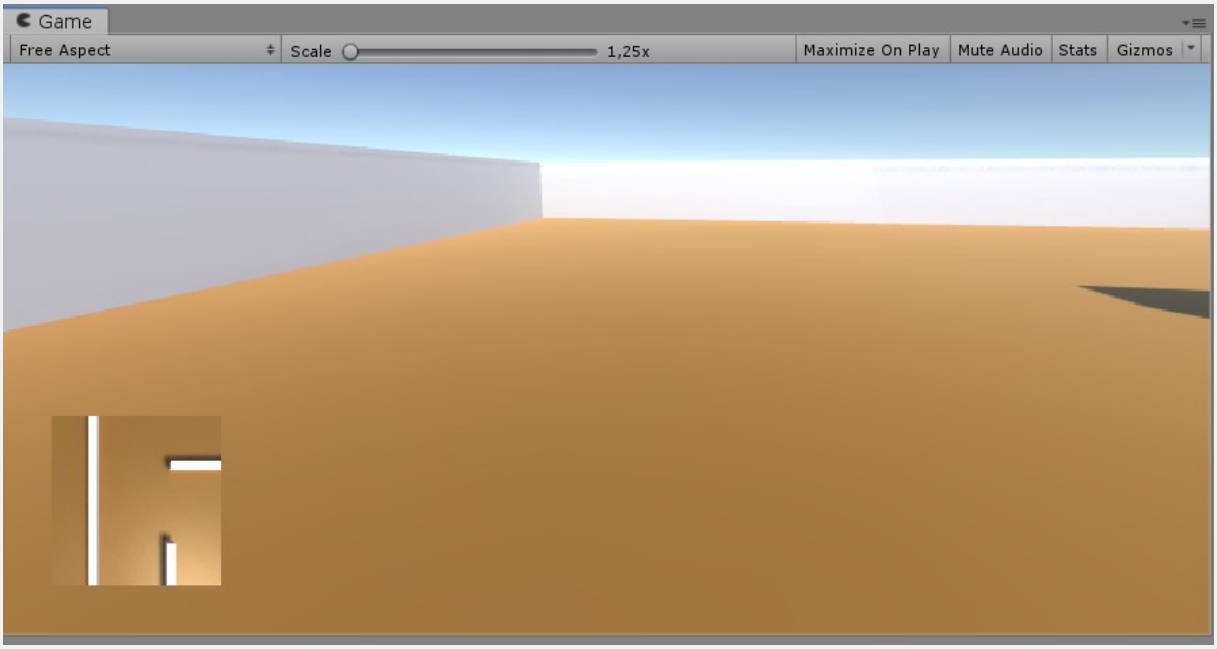
Alapvetően nem szeretnénk, hogy a térkép látható legyen, hanem csak akkor, ha a játékos ezt külön jelzi, így annak megjelenését az „M” betű megnyomásához kötjük. Ezt egy script hozzáfűzésével (*Minimap.cs*) tesszük meg (hasonlóan, ahogyan a játékos mozgatásánál is tettük), mely figyeli az „M” betű megnyomását az Update eljárásban és ha ez megtörtént, akkor a Canvas objektum enabled státuszát negálja (induláskor a GetComponent-tel az objektumot egy változóban eltároltuk):

```
private Canvas CanvasObject;

void Start()
{
  CanvasObject = GetComponent<Canvas> ();
}

void Update()
{
  if (Input.GetKeyUp(KeyCode.M))
  {
    CanvasObject.enabled = !CanvasObject.enabled;
  }
}
```

Utolsó lépés, hogy amikor elindul a játék, akkor ne látszódjon a minimap, ezt a Canvas objektumunk Canvas komponense előtti pipa kiszedésével tehetjük meg. Ezek után a minimap így fog megjelenni a játék során, ha az M billentyű lenyomásra került:



3. ábra Minimap megjelenése a játékban

Trófea, játék vége

A játék célja a trófea megtalálása, ehhez kell egy trófea elem. A Unity Asset Store-ját fogjuk használni a megfelelő trófea megtalálásához, itt rengeteg tárgyat, komponenst, scriptet találhatunk, melyeket aztán behúzhatunk a saját projektünkbe is (vannak ingyenes, illetve fizetős tartalmak is). Én most a Too Many Items: Trophies packját húztam be a projektembe. Unity-ben az Assets menüpont alatt tudjuk beimportálni ezt. Majd elég a Prefabs mappából a már előkészített trófeák közül kiválasztani, behúzni egyet, ezután már a megfelelő helyre tudjuk tenni azt az ideális méretben (én 0,1-szeresére kicsinyítettem a modellt).

A játék utolsó lépése, hogy amennyiben a játékos eléri az előbb beemelt trófeát, akkor a játék jelezze ezt egy üzenet formájában. Ezt többféle módon megtehetjük én a következő módszert használtam: létrehoztam egy Canvas-t (hasonló módon, ahogyan a kamera képénél is tettem), ez fog egy felületet adni majd a felugró „ablaknak”, abban van még egy Canvas, ami egy hátteret ad annak ezzel elválasztva grafikusan is a játék elemeitől, interface-étől, majd a végén ezen belül is szerepel egy Text,

amivel a szöveget adom hozzá a felülethez (jelen esetben, hogy y megnyomásakor újra kezdődjen a játék, míg n hatására kilépjünk a játékból, ennek kódját a következő részben tesszük bele a játékos scriptjébe). A *PlayerMove.cs* Update függvényében két újabb eljárást hívok meg:

```
public Canvas GameOverCanvasObject;

private void WinCheck()
{
  if (Vector3.Distance(transform.position, GameOb-
ject.Find("Trophy").transform.position) < 1.5) {
    GameOverCanvasObject.enabled = true;
  }
}

private void GameOverInputCheck()
{
  if (GameOverCanvasObject.enabled)
  {
    if (Input.GetKeyUp(KeyCode.Y))
    {
      Application.LoadLevel(0);
    }
    else if (Input.GetKeyUp(KeyCode.N))
    {
      Application.Quit();
    }
  }
}
```

Megjegyzés: a WinCheck-en belül a Distance függvénnyel nézzük meg, hogy eléggé közel van-e a játékos poziciója a trófeáéhoz (ha igen, akkor megjelenítjük az előre létrehozott felugró ablakot). Míg a GameOverInputCheck-nél a beviteli gombot vizsgáljuk akkor, ha a felugró üzenet enabled státuszban van (y = új játék, n = játék bezárása). Fontos, hogy a script módosítása után a *Player* Game Over Canvas Object-jébe behúzzuk az előre létrehozott Canvas-ünket.

Miután a scriptek a helyükre kerültek a trófea megtalálása után hasonló üzenet fog megjelenni:

4. ábra A trófea megtalálása és a felugró üzenet

Tesztelés, visszatekintés

Érdemes folyamatosan tesztelni, kipróbálni az egyes elemeket, funkciókat, amiket megcsináltunk, majd a program elkészítésénél átfogó ellenőrzést is végezni. Mik azok a dolgok, amiket meg kell vizsgálnunk, le kell tesztelnünk?

- A játékos szemszögéből látjuk a játékot, annak nézetét az egérrel tudjuk irányítani, a falakon átlátni nem tud.
- A játékos mozgatható (WASD/kurzor gombok), a falakon átmenni nem tud.
- M gomb megnyomására a minimap eltűnik/megjelenik.
- A minimap megfelelően jeleníti meg a falakat, illetve a trófeát.
- A minimap követi a játékos mozgását.
- A trófea megtalálása után a játék véget ér. Y/N gomb megnyomására újrakezdi a játékot/kilép a játékból.

Ha bármelyik ponttal gond adódik érdemes visszatekinteni az anyag megfelelő részeire.

Továbbfejlesztési lehetőségek

A játék elkészítésének, tesztelésének végeztével aztán érdemes további igényeket, lehetőségeket felvetni (megfelelő időkerettel aztán meg is valósítani). Pár ötlet/feladat erre vonatkozólag:

- TPS nézet megvalósítása (a játékost a játék során hátulról, kissé fentebbről látjuk).
- Gyorsfutás (shift + irány), ugrás (space) implementálása.
- Ellenfelek létrehozása (pl.: ha elkapnak vesztesz).
- Egy nyíl, ami mutatja merre van a trófea.
- Az előző pont tovább gondolása: egy „szellem”, aki mutatja az utat a kincshez (előre beprogramozott útvonalon).
- Gyűjthető dolgok kreálása (például érmék), pontszám megjelenítése a felületen.
- Idő használata:
    - Limit meghatározása, aktuális idő mutatása.
    - Játékon belüli legjobb idők mutatása (toplista). (adatbázis, internetkapcsolat)
- Különböző zenék (aláfestés), hangok (lépéshang, kincs megtalálása) betétele.

További lehetséges témakörök

Ezen a példán keresztül (amennyiben van rá időkeret) érdemes tovább építeni, fejleszteni a tudást újabb játékok kreálásával. Ehhez példákat láthatunk a Unity tutorial weboldalán, illetve érdemes lehet a kreativitásra is hagyatkozni, így mikor felmerülnek újabb ötletek, igények, akkor kutakodhatunk ezek megvalósításának mikéntje után. Itt segítségünkre lehet az előbb említett tutorial oldal, a többször linkelt dokumentáció, de akár különböző fórumok, YouTube videók is, hiszen egyre elterjedtebb a technológia és egyre többen osztják meg valamilyen formában saját műveiket, illetve azok elkészítésének mikéntjét.

Órai felbontás

Ha szeretnénk az anyagot tanórákra felbontani, akkor én a következőt javasolnám:

- 0. óra: Habár nem szükséges magas fokú C# ismeret a labirintus létrehozásához, adott esetben érdemes lehet egy bevezető órával megismertetni a diákokat a programozási nyelvvel.
- 1. óra: Unity-vel és annak kezelőfelületével való ismerkedés.
- 2. óra: A pálya és a játékos létrehozása, irányítás implementálása.
- 3. óra: Minimap, trófea. Játék lezárásának implementálása. Végtermék kipróbálása, tesztelése.
- 4-5. óra: Opcionálisan újabb igények, lehetőségek felmérése, ehhez kapcsolódóan egyedi kutatás, implementálás, megvalósítás.