

**SREDNJA ŠKOLA ZVANE ČRNJE**  
**SCUOLA MEDIA SUPERIORE ZVANE ČRNJE ROVINJGO**  
**Carduccijska 20**  
**52210 Rovinj**

## **ZAVRŠNI RAD**

Tema:

***Programiranje skripte za Unity igru***

Mentor:

Dražen Domitrović, struč. spec. ing. techn. inf.

Učenik:

Juraj Crljenko 4.d

Smjer: Tehničar za računalstvo

Rovinj, ljetni rok  
šk. god.  
2021./2022.

# Sadržaj

<b>1</b>	<b>UVOD.....</b>	<b>3</b>
<b>2</b>	<b>ŠTO JE TO UNITY ENGINE .....</b>	<b>4</b>
2.1	RAZVOJ UNITY-A KROZ POVIJEST.....	4
2.1.1	Unity 2.0 .....	4
2.1.2	Unity 3.0 .....	5
2.1.3	Unity 4.0 .....	5
2.1.4	Unity 5.0 .....	5
2.1.5	Unity 2017- danas.....	5
2.2	POZNATE IGRE UNITY-A.....	6
<b>3</b>	<b>INVENTORY SYSTEM (SUSTAV PRTLJAGE) .....</b>	<b>7</b>
3.1	APSTRAKTNI DIO .....	7
3.2	DODIJELIVI DIO .....	9
3.2.1	<i>InventorySystem skripta</i> .....	9
3.2.1.1	Metode „InventorySystem“ skripte .....	10
3.2.1.1.1	Add() metoda (metoda za dodavanje) .....	10
3.2.1.1.2	Remove() metoda (metoda za micanje).....	11
3.2.1.1.3	GetKeyFromValue() metoda (metoda za pristup ključu pomoću vrijednosti).....	12
3.2.1.1.4	PrintInventory() metoda (metoda za ispis prtljage).....	12
3.2.2	<i>InventorySave skripta i podatkovni dio</i> .....	14
3.2.2.1	Metode InventorySave skripte .....	15
3.2.2.1.1	Save() metoda .....	15
3.2.2.1.2	Load() metoda.....	20
3.2.2.1.3	Clear() metoda .....	21
3.2.2.1.4	ShowExplorer() metoda .....	21
3.3	DIO ZA UREĐENJE .....	22
3.3.1	<i>EInventory</i> .....	22
3.3.2	<i>EWInventoryAdd</i> .....	27
3.3.3	<i>InventoryObjectCreation (dodatak sustavu)</i> .....	31
<b>4</b>	<b>IMPLEMENTACIJU SUSTAVA U VANJSKIM PROJEKTIMA .....</b>	<b>32</b>
4.1	UBACIVANJE GOTOVOG SUSTAVA U UNITY.....	32
4.1.1	<i>Izrada novog Unity projekta</i> .....	32
4.1.2	<i>Dodavanje sustava u projekt (princip Unity paketa)</i> .....	33
<b>5</b>	<b>ZAKLJUČAK .....</b>	<b>36</b>
<b>6</b>	<b>LITERATURA.....</b>	<b>37</b>
<b>7</b>	<b>PRILOZI .....</b>	<b>38</b>



## 1 Uvod

U završnom radu prikazat ću programiranje skripti za Unity igru na primjeru programiranja sustava za spremište (*Inventory system*), kojeg sam izradio za potrebe rada.

Cilj rada je omogućiti korisnicima Unity-a korištenje sustava za spremište te da implementacija tog sustava u bilo koji projekt bude lagana, odnosno da se sustav može lagano postaviti u bilo koju igricu ili neku drugu aplikaciju izrađenu u Unity-u. Prilikom izrade podijelio sam sustav na četiri dijela: apstraktni dio, dodjeljivi dio, podatkovni dio i dio za uređenje. U radu ću opisati svaki od spomenutih dijelova i prikazati izvorne kodove skripti tih dijelova, a spomenuto će biti popraćeno i grafičkim prikazima u Unity-u. Posebno ću se osvrnuti na dodjeljivi dio u kojem se nalazi srž čitavog sustava koji omogućuje korisniku uređivanje postavki spomenutog sustava. Naposljetku ću prikazati mogućnosti njegove implementacije u vanjskim projektima. Naime, do sada sam u nekoliko navrata radio na nekim igricama, te sam uvidio potrebu za izradom sustava za spremište, te sam ga, nakon što sam ga napravio, nastojao i implementirati u pojedine od spomenutih igrica. To je bila glavna motivacija za izradu sustava za spremište koji je jednostavan, lako se implementira, a sadrži sve potrebne komponente.

## 2 Što je to *Unity Engine*

Unity Engine (hrv. Motor), je alat koji je izradila američka korporacija Unity Technologies 2005. godine te se razvija i dan danas. Cilj *Unity*-a je bio postati alat, tj. sustav koji bi omogućavao većem broju ljudi izradu videoigara. Iako je Unity izvorno pisan u C++-u, kako bi se razvoj aplikacije (najčešće igrice) olakšao korisnicima, *Unity* je standardizirao podršku za programskih jezika *C#* i *javascript* (koji je poslije izbačen) te posebne biblioteke (en. library) *UnityEngine*. *Unity* također sadrži već izrađene komponente i pakete koji olakšavaju cijeli proces izrade aplikacija. S tim znanjem, korisnik može izrađivati razne skripte koje se na kraju u cjelovitosti smatraju projektom.



### 2.1 Današnji logo *Unity*-a

#### 2.1 Razvoj *Unity*-a kroz povijest

Kako je razvoj *Unity*-a započeo 2000-tih godina, danas i on sam ima nekakvu povijest. Razvijao se kroz šest faza od kojih se šesta faza i dan danas razvija te je vrlo popularan, razdijelio se na besplatnu i plaćenu varijantu.

##### 2.1.1 Unity 2.0

Za razliku od prve početne verzije iz 2005. godine, 2007. godine izašla je nova inačica koja se sastojala od čak 50-ak novih značajki. Sastojala se od motora za teren (eng. *Terrain engine*) koja je omogućavala 3D okruženje. Ova inačica je također prvi put dopustila *Unity*-u da se rade mrežne aplikacije.

### 2.1.2 Unity 3.0

2010. godine izašla je treća inačica *Unity*-a. Ova inačica je omogućavala podršku za konzole te mobilne uređaje (Android i IOS). Također je dodala još neke značajke kao što su naprimjer generator stabala, filtriranje zvuka, UV mapiranje i slično.

### 2.1.3 Unity 4.0

U toj inačici, 2012. godine, *Unity*-a dodana je podrška za renderiajući sustav *DirectX*<sup>1</sup>, *Adobe Flash* te *Linux*. *Facebook* je također izradio svoj *SDK*<sup>2</sup>. Ova inačica se najviše usredotočila na oglašavanje igara.

### 2.1.4 Unity 5.0

*Unity 5* je jedna od najvećih inačica *Unity Engine*-a, upravo zato jer je revolucionarno, 2015. godine, omogućavala izradu igara za svaku konzolu besplatno (tada se još *Unreal Engine* plaćao). Ova inačica je dodala podršku za *WebGL* koji je omogućavao korisnicima da bez ikakvih dodataka mogu graditi igre za web, *Vulkan API*, *Nvidia PhysX*. Ova inačica je također uvela *Unity Cloud* koji omogućava korisnicima da spremaju *backup* od svojih projekata na internet.

### 2.1.5 Unity 2017- danas

Kako bi se verzije prestale pojavljivati svakih par godina već u malo češćem periodu vremena, pred kraj 2016. godine, *Unity Technologies* je najavio kako će se koristiti godina za ime inačice (npr. *Unity 2021.2.16f1*). Tako danas *Unity* ima svoj *launcher*<sup>3</sup> nazvan *Unity Hub*, podržava svijetlu i tamnu opciju *Editora* te omogućava tzv. *Visual scripting*<sup>4</sup> pomoću nekih *paketa*<sup>5</sup>.

*Unity Engine* se i danas razvija. Kod za *Built-in* komponente *Unity*-a javno je dostupan ja njihovom *GitHub*-u: <https://github.com/Unity-Technologies/UnityCsReference> te se svatko može njime koristiti.

---

<sup>1</sup> DirectX je skupina programskih okruženja (API)-a koji omogućavaju lakše izvođenje određenih zadataka za igre na *Windows* operativnom sustavu.

<sup>2</sup> SDK (eng. Software Development Kit) jest skup alata koji omogućuju stvaranje izvršnog software-a za neku platformu (npr. *PlayStation*).

<sup>3</sup> *Launcher* jest program koji omogućava korisniku da upali aplikaciju, u konkretnom slučaju *Unity*-a, korisnik može ući u projekt, napraviti ga, obrisati ga, promijeniti verziju, gledati videe za učenje (eng. *tutorial*) i sl.

<sup>4</sup> Visual Scripting jest način programiranja koji omogućava korisniku da programira koristeći blokove umjesto koda.

<sup>5</sup> *Paketi* ili *add-oni* u *Unity*-u su već gotovi projekti koje je potrebno povući i spustiti u projekt. To je skup skripti, slika, videa i sl. koji zajedno čine nekakav mali sustav.

## 2.2 Poznate igre Unity-a

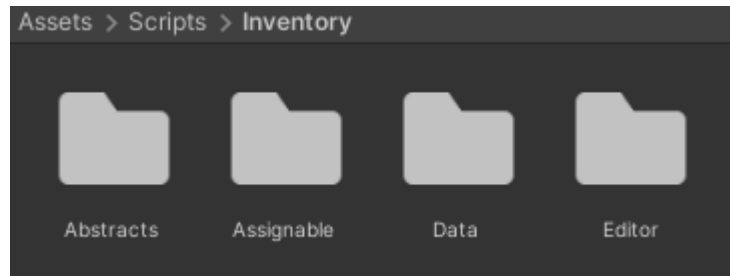
Kako se Unity razvijao tako je sve više ljudi saznavalo za njega te su se ljudi počeli njime i koristiti. Kako je popularnost rasla, tako je *Unity* danas, zajedno i sa *Unreal Engine*-om standard za izradu video igara.

Neke od popularnijih igara napravljenih u Unity-u su:

- Genshin Impact
- Fate/Grand Order
- Pokémon Go
- Cuphead
- Ori and the Will of the Wisps
- Among Us

### 3 Inventory System (Sustav prtljage)

Sustav prtljage, također poznat pod imenom *Inventory System*, jedan je od ključnih sustava za rad modernih video igri. Likovi u igrici, neprijatelji te ne-igrajući likovi (tzv. NPC-evi) imaju taj sustav ugrađen u sebi. Kako bih prilagodio korisnost sustava za bilo kakav objekt u igrici, rasporedio sam ga na četiri glavna dijela (koji su raspoređeni u foldere), a to su sljedeći:



3.1 Prikaz organizacije projekta u foldere

#### 3.1 Apstraktni dio

Apstraktni dio ovog sustava sastoji se od jedne glavne *Apstraktne klase*<sup>6</sup> → *AItem*.

```
[System.Serializable]
public abstract class AItem : MonoBehaviour
{
    ...
}
```

3.2 Definicija klase *AItem*

Ta klasa sadrži glavne informacije o objektu, a to su, tip (Enum vrijednost), ime (string vrijednost), kupovna i prodajna cijena (int vrijednosti) te broj koji pokazuje koliko istih stvari/item-a ima u spremištu, tj. Inventory-u te također sadrži (nasljeđuje) tzv. *MonoBehaviour* klasu koja omogućuje dodjele te klase *GameObjectu* koji se nalazi u igračoj sceni<sup>7</sup>, no kako je trenutna klasa apstraktna, *AItem* klasa se neće moći dodjeljivati.

<sup>6</sup> Apstraktna klasa jest tip klase koja ne može imati objekte. Njezina glavna svrha jest da bude *parent* klasa koju se nasljeđuje unutar druge klase.

<sup>7</sup> Scena unutar Unity Engine-a označuje nivo ili mapu (ponekad i cijeli svijet igrice). Korisnik unutra može dodavati, micati te uređivati sve *GameObject*-e. Primjer scena: Grad(ulice u gradu), škola (prostorije škole), šuma, itd.



```

#region Definicija Tipova Stvari/Itema
public enum ItemType
{
    WEAPON,
    ARMOR,
    HEALTH,
    COLLECTABLE
}
#endregion

#region Definicija atributa klase
[Header("Item Settings")]
protected int itemID;
[SerializeField] protected ItemType itemType;           //tip stvari
[SerializeField] protected string itemName;             //ime stvari

[Space]
[SerializeField] protected int buyValue;                //kupovna vrijednost
[SerializeField] protected int sellValue;               //prodajna vrijednost
[SerializeField] protected int amountInInventory = 1;    //broj u spremištu
#endregion

```

### 3.3 Definicija atributa klase AItem.

Nakon što smo izradili attribute, kako bi smo omogućili mijenjanje i dodjeljivanje vrijednosti u klasama različitim od *AItem*, potrebno je napraviti svojstva za potrebne attribute. To sam napravio tako da sam definirao novo „javno“ (public) svojstvo koje ima metode *get* i *set* te da te metode gledaju na određenu vrijednost koje svojstvo treba primiti odnosno promijeniti.

```

#region Definicija svojstava
public int ItemID { get => itemID; set => itemID = value; }
public string ItemName { get => itemName; }
public ItemType ItemTypeValue { get => itemType; }
public int BuyValue { get => buyValue; set => buyValue = value; }
public int SellValue { get => sellValue; set => sellValue = value; }
public int AmountInInventory { get => amountInInventory; set => amountInInventory = value; }
#endregion

```

### 3.4 Definicija svojstava klase AItem.

Na kraju imamo *Start()* metodu koja se poziva od strane Unity-a pri početku instance igre. U toj metodi se pozivam samo jednu naredbu, a to je dodjeljivanje vrijednosti atributu *amountInInventory* pomoću ternarnog operatora. Ternarni operator jest tip operatora gdje se *if-else* uvjeti pišu u samo jednoj liniji koda. U ovom slučaju koristim ternarni operator kako bih

provjerio nalazili se trenutna stvar/item u spremištu te dodijeliti mu trenutnu vrijednost u spremištu ako postoje dok ukoliko ne postoji dodijeliti 1.

```
void Start()
{
    amountInInventory = amountInInventory == 0 ? 1 : amountInInventory;
}
```

### 3.5 Start() metoda klase AItem.

Također sam napravio dvije dodatne klase koje se potencijalno mogu koristiti ukoliko korisnik želi stvarati drugačije tipove stvari/itema, a to su *AWeapon* i *AArmor* koje su također apstraktne te nasljeđuju glavnu *AItem* klasu no u ovom radu baviti ću se samo radom s *AItem-om*.

## 3.2 Dodjeljivi dio

Dodjeljivi dio ovog sustava se može podijeliti na još dvaju manja dijela. To su „*InventorySystem*“ i „*InventorySave*“ klase, tj. metode.

### 3.2.1 InventorySystem skripta

Sada dolazimo do dodjeljivog dijela sustava spremišta koji je ujedno i najvažniji dio sustava, a to je zato jer je ovdje smještena jezgra sustava, klasa (tj. Skripta) pod nazivom *InventorySystem*. No za razliku od prijašnje „*AItem*“ klase, klasa „*InventorySystem*“ koristi oznaku „*RequireComponent*“ kako bi se limitiralo dodavanje ove klase (kao komponente) na određeni *GameObject* s obzirom je li na tom *GameObject-u* već postavljena klasa „*InventorySave*“.

```
[RequireComponent(typeof(InventorySave))]
[System.Serializable]
public class InventorySystem : MonoBehaviour
{
    ...
}
```

### 3.6 Definicija klase InventorySystem

Nakon što je klasa definirana, slijedi definicija glavnog *inventory* atributa tipa „*Dictionary<integer, AItem>*”<sup>8</sup>. Naravno, kako bi se zaštitilo ove podatke, atribut *inventory* sam učinio privatnim atributom te sam napravio posebnu „get-set” metodu kojom se *inventory*-u dodaju te miču vrijednosti.

```
private Dictionary<int, AItem> inventory = new Dictionary<int, AItem>();  
public Dictionary<int, AItem> Inventory { get => inventory; set => inventory = value; }
```

### 3.7 Definicija *inventory*-a

#### 3.2.1.1 Metode *InventorySystem* skripte

*InventorySystem* klasa u sebi sadrži metode za dodavanje → *Add()* i uklanjanje → *Remove()* elemenata u *inventory* također ima pomoćne metode *PrintInventory()* za ispis svih elemenata određenog spremišta te metodu *GetKeyFromValue()* koja daje pristup određenom ključu pomoću njegove vrijednosti (reverzibilna metoda).

##### 3.2.1.1.1 *Add()* metoda (metoda za dodavanje)

*Add()* metoda jest javna metoda (procedura) koja kao parametar zahtjeva „*AItem*” objekt. Metoda prvo definira privremenu varijablu *x* kojoj pridružuje vraćenu vrijednost *GetKeyFromValue()* metode. Drugi stupanj rada ove metode jest ispitivanje postoji li *x* u *inventory*-u. Nakon što je ovaj uvjet ispitan događa se jedno od sljedećeg:

- Ukoliko postoji, vrijednosti u *inventory*-u pod ključem *x* se vrijednost povećava za 1
- Ukoliko ne postoji, dodaje se nova vrijednost u *Inventory*.

---

<sup>8</sup> Dictionary je tip spremanja podataka u obliku „ključ → vrijednost” što omogućuje svakom ključu da pokazuje na neku vrijednost, (slično listi „pokazivača”).

```

public void Add(AItem item)
{
    int x = GetKeyFromValue(item.ItemName);
    if (Inventory.ContainsKey(x))
    {
        Inventory[x].AmountInInventory += 1;
    }
    else
    {
        Inventory.Add(Inventory.Count + 1, item);
        item.ItemID = Inventory.Count;
    }
}

```

3.8 Add() metoda klase InventorySystem

#### 3.2.1.1.2 Remove() metoda (metoda za micanje)

Remove() metoda također je javna metoda (procedura) koja zahtjeva „AItem“ objekt. Metoda prvo provjeri postoji li vrijednost danog objekta u *inventory*-u te ukoliko postoji nastavlja dalje → smanjuje vrijednost ID-a danog objekta u *inventory*-u i ako mu se vrijednost smanji na 0, dani objekt se briše iz *inventory*-a.

```

public void Remove(AItem item)
{
    if (Inventory.ContainsValue(item))
    {
        Inventory[item.ItemID].AmountInInventory -= 1;
        if (Inventory[item.ItemID].AmountInInventory == 0)
        {
            Inventory.Remove(item.ItemID);
        }
    }
}

```

3.9 Remove() metoda klase InventorySystem

### 3.2.1.1.3 GetKeyFromValue() metoda (metoda za pristup ključu pomoću vrijednosti)

GetKeyFromValue() metoda jest metoda koja za razliku od Add() i Remove() metoda vraća vrijednost tipa integer (za razliku od prošlih metoda koje su bile procedure, ova metoda je funkcija). Jednostavnim linearnim traženjem pretražujem koja vrijednost u inventory-u ima isto ime kao i dani string koji metoda zahtjeva. Na kraju, ako je vrijednost pronađena, vraćam ključ te vrijednosti, a ako nije, vraćam broj vrijednosti inventory-a + 1 (kako bi se izradila nova vrijednost u inventory-u).

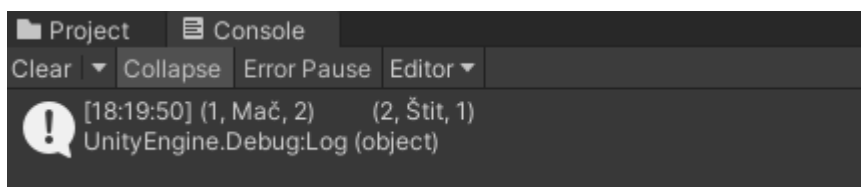
```
public int GetKeyFromValue(string name)
{
    foreach (var kv in Inventory)
    {
        if (kv.Value.ItemName == name)
        {
            return kv.Key;
        }
    }
    return Inventory.Count + 1;
}
```

3.10 GetKeyFromValue() metoda klase InventorySystem

### 3.2.1.1.4 PrintInventory() metoda (metoda za ispis prtljage)

PrintInventory() metoda jest dodatna metoda (procedura) koja služi za proces debugiranja. Radi tako da definira privremeni početnu string vrijednost te ispita je li inventory prazan. Ako nije, dodaje u string svaku vrijednost iz *inventory*-a u taj string koji se ispisuje na ekran u sljedećem obliku:

```
„(<ID>, <Ime>, <broj_istih>)“.
```



3.11 Prikaz vrijednosti inventory-a u konzoli.

```
public void PrintInventory()
{
    string value = "";
    if (Inventory != null)
    {
        foreach (var item in Inventory)
        {
            value += $"({item.Key}, {item.Value.ItemName}, {item.Value.AmountInInventory})
            \t\r";
        }
    }
    if (value != string.Empty) Debug.Log(value);
}
```

3.12 PrintInventory() dodatna metoda klase InventorySystem za svrhe debugiranja

### 3.2.2 *InventorySave* skripta i podatkovni dio

Kako bi *InventorySystem* skripta bila postavljena na objekt, zajedno sa sobom zahtjeva korištenje *InventorySave* klase. *InventorySave* po svom samom imenu predstavlja rad s diskom → učitavanje i spremanje podataka. Za postupak spremanja, naravno, trebamo nekakvu posebnu datoteku, pa zato, pri dodavanju ove skripte trebamo upisati ime datoteke te njezin tip (npr. *skolska\_torba.json*). Potom tu datoteku ukoliko ne postoji kreira sustav te posprema sve podatke iz glavnog *Inventory dictionary*-a u novonastalu/promijenjenu datoteku. Ta se datoteka potom može opet učitati te mijenjati.

Pri definiranju klase sam naslijedio *MonoBehaviour* klasu kako bi mogao ovu klasu koristiti kao komponentu u Editoru.

```
public class InventorySave : MonoBehaviour
{
    ...
}
```

#### 3.13 Definicija klase *InventorySave*

Nakon što je klasa izrađena potrebno joj je definirati već prije navedene attribute, kao što su filename (datoteka u koju se sprema), *inventory* te objekti koji se spremaju → *items*. Kako bih olakšao snalaženje korisnika među datotekama napravio sam tzv. *Path* (hrv. putanja). To je string atribut koji određuje točno mjesto gdje će se novonastala ili izmijenjena datoteka nalaziti.

```
[SerializeField] private string filename;
private InventorySystem inventory => gameObject.GetComponent<InventorySystem>();
private List<ItemData> items = new List<ItemData>();

protected string Path => Application.persistentDataPath + "/Inventory/" + filename;
```

#### 3.14 Definicija atributa klase *InventorySave*

### 3.2.2.1 Metode *InventorySave* skripte

Iako ova klasa ima četiri metode, možemo je podijeliti na dva glavna dijela, a to su dio za spremanje ili `Save()` metoda te dio za učitavanje ili `Load()` metoda. Tu su također `Clear()` i `Start()` metode koje imaju malo manju ulogu u radu s podacima.

#### 3.2.2.1.1 `Save()` metoda

`Save()` metoda vrlo je važna za rad cijelog ovog sustava. Bez ove metode igrač igrice koje bi korisnik napravio ne bi mogao nastaviti od mjesta gdje je stao već bi morao započeti iznova svaki put kada bi zaželio igrati.

Kako bi spremio podatke trebalo mi je nekakvo spremište podataka, pa sam tako odlučio napraviti posebnu „privremenu“ klasu *InventoryData* unutar iste skripte i u nju postaviti Listu objekata koji bi mi služili kao podaci. Naravno, kako bih omogućio spremanje te klase u datoteku označio sam je sa *Serializable* oznakom.

```
public class InventorySave : MonoBehaviour
{
    ...
}

[System.Serializable]
public class InventoryData
{
    public List<ItemData> itemDatas;
}
```

3.15 Prikaz dviju klasa (*InventorySave* i *InventoryData*) unutar jedne skripte.

U Metodi `Save()` prvo izrađujem instancu klase *InventoryData* pod nazivom *data*, zatim prolazim po svakoj vrijednosti *inventory*-a te izrađujem novu *ItemData* instancu s vrijednostima trenutne vrijednosti *inventory*-a. *ItemData* jest vanjska klasa koja posprema određene vrijednosti u instancu te se definira posebnim konstruktorom.



```

[System.Serializable]
public class ItemData
{
    public int _itemID, _itemType, _buyValue, _sellValue, _amount;
    public string _itemName;

    public ItemData(int itemID, int itemType, string itemName, int buyValue, int
sellValue, int amount)
    {
        _itemID = itemID;
        _itemType = itemType;
        _itemName = itemName;
        _buyValue = buyValue;
        _sellValue = sellValue;
        _amount = amount;
    }
}

```

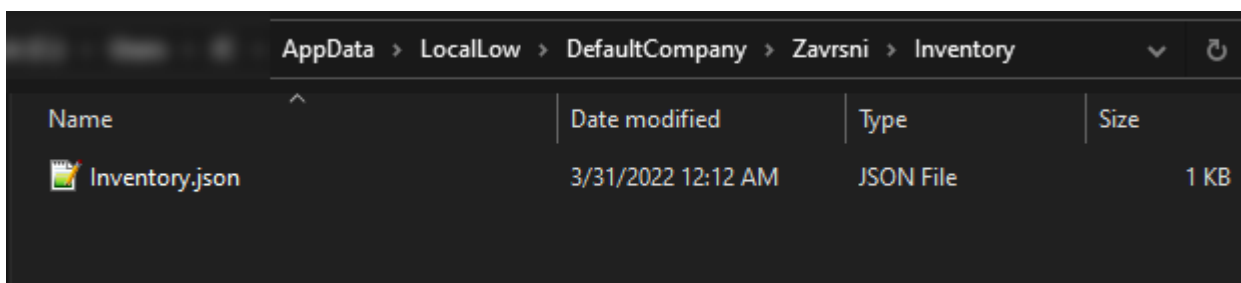
### 3.16 Klasa ItemData

Nakon što je instanca napravljena provjeravamo Linq izrazom postoji li već vrijednost s istim ID-em te ako postoji, ta vrijednost se briše. Na kraju se dodaje trenutna vrijednost tj. instanca *ItemDate* koju smo prije bili napravili.

Kada je petlja završena, globalnu varijablu (koju smo definirali na početku klase *InventorySave*) *items* postavljamo kao vrijednost listi objekata za spremanje u instanci *data*.

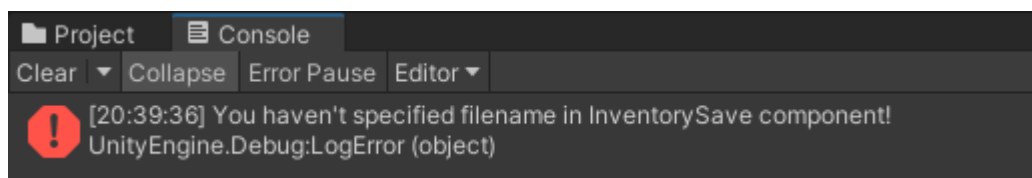
Sada se započinje s procesom spremanja podataka. Kako bih omogućio spremanje podataka bio mi je potreban jedan od poznatih biblioteka, *System.IO library*. Prvo se provjerava postoji li datoteka *Path* na disku. Ovo je nužni dio ukoliko bi se *Path* izradio ako ne postoji. Do ovog slučaja se može doći ako:

- Prvi put igrač pali igru
- Prvi put se sprema *inventory*
- Postoje više različitih *inventory*-a u projektu



3.17 Prikaz lokacije Path datoteke na disku

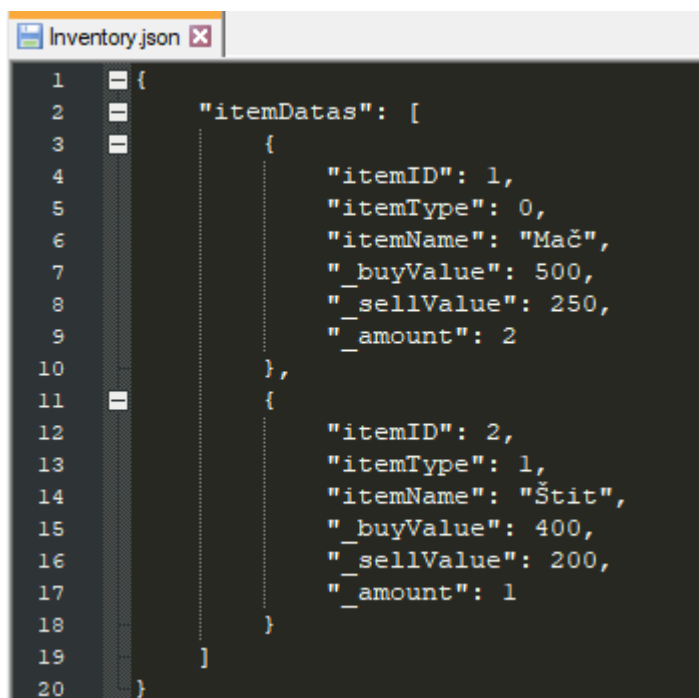
Kada je *Path* pronađen, sve što je preostalo jest spremiti podatke. No kako bi se spriječio mogući problem → ukoliko bi *filename* bio null, dolazilo bi do pogreške, pa sam tako napravio *try-catch* blok kojim sam odijelio funkciju za spremanje i poruku koja bi se trebala ispisati u konzoli u slučaju moguće pogreške.



3.18 Prikaz greške koja se dogodi u slučaju da je string filename prazan.

Za spremanje sam koristio pretvarač u JSON<sup>9</sup> format, pa će se tako svi objekti spremiti u obliku JSON te će to raditi kao mala baza podataka u trenutnoj igrici.

<sup>9</sup> JSON (eng. JavaScript Object Notation) jest, zajedno sa *xml*-om jedan od standardnih formata (tj. tipova) datoteke i format za razmjenu podataka koji koristi ljudski čitljiv tekst za pohranu i prijenos podataka objekata koji se sastoje od parova ključ-vrijednost i nizova.



```
1  {
2    "itemDatas": [
3      {
4        "itemID": 1,
5        "itemType": 0,
6        "itemName": "Mač",
7        "_buyValue": 500,
8        "_sellValue": 250,
9        "_amount": 2
10     },
11     {
12       "itemID": 2,
13       "itemType": 1,
14       "itemName": "Štit",
15       "_buyValue": 400,
16       "_sellValue": 200,
17       "_amount": 1
18     }
19   ]
20 }
```

3.19 Prikaz JSON-a jednog inventory-a

```

public void Save()
{
    InventoryData data = new InventoryData();
    foreach (KeyValuePair<int, AItem> pair in inventory.Inventory)
    {
        ItemData curVal = new ItemData(pair.Value.ItemID,
        (int)pair.Value.ItemTypeValue, pair.Value.ItemName, pair.Value.BuyValue,
        pair.Value.SellValue, pair.Value.AmountInInventory);
        if (items.Any(item => item.itemID == pair.Value.ItemID))
        {
            items.Remove(items.Find(item => item.itemID == pair.Value.ItemID));
            //continue;
        }
        items.Add(curVal);
    }
    data.itemDatas = items;

    if (!File.Exists(Path))
    {
        (new FileInfo(Path)).Directory.Create();
    }
    try
    {
        File.WriteAllText(Path, JsonUtility.ToJson(data, true));
    }
    catch (UnauthorizedAccessException)
    {
        Debug.LogError("You haven't specified filename in InventorySave component!");
    }
}

```

### 3.20 Save() metoda klase InventorySave

### 3.2.2.1.2 Load() metoda

Nakon što imam Save() metodu, kako bih te podatke vratio nazad u program, potrebna je Load() metoda tj. metoda za učitavanje. U ovom slučaju ne bavim se učitavanjem cijele scene, igrača, neprijateljskih AI-eva i slično već se bavim čitanjem jedne obične JSON datoteke, smještene na lokaciji *Path*-a.

Prvo se provjerava postoji li datoteka *Path* jer ukoliko ne postoji, program ne može ništa pročitati pa bi se tako pojavila greška. Kao i u Save() metodi prva potrebna stvar je instanca klase *InventoryData* te sam je opet, kao i prije nazvao *data*, no za razliku od Save() metode ovdje *data* nije nova instanca, već joj se pridružuju vrijednosti koje su pročitane iz JSON datoteke (pomoću ReadAllText() metode iz File klase) te tako poprma sve vrijednosti. Zatim se globalnom atributu *items* pridružuje vrijednosti liste iz *data* instance.

Kako bi dodali svaki od učitanih vrijednosti u *inventory*, potrebno je proći kroz *items* te dodati svaku vrijednost u *inventory*.

```
public void Load()
{
    if (File.Exists(Path))
    {
        InventoryData data =
        JsonUtility.FromJson<InventoryData>(File.ReadAllText(Path));
        items = data.itemDatas;
        foreach (var item in items)
        {
            AItem tempItem = new Item(item._itemID, (AItem.ItemType)item._itemType,
            item._itemName, item._buyValue, item._sellValue, item._amount);
            inventory.Add(tempItem);
        }
        inventory.PrintInventory();
    }
}
```

3.21 Load() metoda klase InventorySave

### 3.2.2.1.3 Clear() metoda

Clear() je dodatna metoda koja služi kako bi omogućila korisniku, tj. osobi koja radi program koristeći ovaj paket, da očisti *inventory* te krene iznova. Kako *Dictionary* i *Liste* već imaju metodu Clear(), koristio sam tu ugrađenu metodu umjesto pisanja svoje.

```
public void Clear()
{
    inventory.Inventory.Clear();
    items.Clear();
}
```

3.22 Clear() dodatna metoda klase InventorySave za svrhe debugiranja

### 3.2.2.1.4 ShowExplorer() metoda

ShowExplorer() je dodatna metoda koja služi kako bi omogućila korisniku otvaranje mjesto datoteke *Path* u *File Exploreru*.

Radi na način da uzme *Path* te pretvori sve iz „/“ u „\“ te popravljene string spremi u *itemPath* koji onda otvara u *File Exploreru*.

```
public void ShowExplorer()
{
    string itemPath = Path.Replace(@"\", @"\");
    System.Diagnostics.Process.Start("explorer.exe", "/select," + itemPath);
}
```

3.23 ShowExplorer() dodatna metoda klase InventorySave za svrhe debugiranja

### 3.3 Dio za uređenje

Uređivački/Sučeljni dio sustava je izrađen kako bi se korisniku olakšalo korištenje ovog projekta kao paket u drugim projektima. Radi tako da kada se komponenta (tj. skripta) pridruži određenom *GameObject*-u, umjesto da se prikazuju vrijednosti te komponente, prikazuje se *Editor* skripta, ali glavna skripta i dalje zadržava svoju funkcionalnost. Ovaj dio sam podijelio u 3 glavne skripte koje također predstavljaju i klase, a to su *EInventory*, *EWInventoryAdd* i *InventoryObjectCreation*.

*Editor* skripte se dijele na dva različita tipa:

- *Editor* (koji radi s skriptom u Inspector prozoru)
- *EditorWindow* (koji stvara novi prozor u projektu)

Skripte koje nasljeđuju *Editor* klasu za prikaz svojstava potrebne su dvije metode *OnEnable()* i *OnInspectorGUI()* dok za one koje nasljeđuju *EditorWindow* klasu trebaju imati tri metode *ShowWindow()*, *OnEnable()* i *OnGUI()*.

#### 3.3.1 EInventory

*EInventory* skripta odnosi se na prikaz (korisničko sučelje) *InventorySystem* skripte unutar *Inspector*-a korisnika kojoj je cilj lak pregled, spremanje te učitavanje *inventory*-a.

Pri definiciji klase sam naslijedio *Editor* klasu koja omogućava uređivanje „Inspektora“ te je označio s *CustomEditor*-oznakom koja pokazuje na klasu *InventorySystem*

```
[CustomEditor(typeof(InventorySystem))]  
public class EInventory : Editor  
{  
    ...  
}
```

3.24 Definicija klase *EInventory*

Kako će mi za rad ovog sustava biti potrebni i *InventorySystem* i *InventorySave* klase, prvo su mi trebale definicije atributa njihovih tipova.

```
InventorySystem inventoryReferenceP;  
InventorySave _dataMangement;
```

### 3.25 Definicija instanci klasa *InventorySystem* i *InventorySave*

Prije crtanja korisničkog sučelja za skriptu prvo je trebamo uključiti. Za to služi metoda `OnEnable()`.

Unutar `OnEnable()` metode prvo se dodjeljuje vrijednost instancama *InventorySystem* i *InventorySave* skripti te se odmah učitavaju vrijednosti u *inventory*.

```
private void OnEnable()  
{  
    inventoryReferenceP = FindObjectOfType<InventorySystem>();  
    try  
    {  
        _dataMangement = inventoryReferenceP.gameObject.GetComponent<InventorySave>();  
    }  
    catch  
    {  
        Debug.LogError("You have to add InventorySave component to your inventory  
GameObject");  
    }  
    _dataMangement.Clear();  
    _dataMangement.Load();  
}
```

### 3.26 *OnEnable()* metoda editor klase za *InventorySystem*

Kada imamo vrijednosti u atributima, možemo započeti s crtanjem polja i vrijednosti na ekran. Za to služi `OnInspectorGUI()` metoda. No nju ne definiramo već je proširujemo koristeći ključne riječi *override* te *base*. Prije bilo kakvog crtanja provjerava se je li *inventory* prazan ako nije prazan, program zahvaća svaku vrijednost *inventory*-a te je ispisuje u obliku ID, Ime te broj istih u *inventory*-u. Na kraj te linije dodaje dva dugmića, prvi ima simbol „+“ te označava povećanje iste vrijednosti dok drugi dugmić ima simbol „-“ te označava smanjenje iste vrijednosti u *inventory*-u. Kako sam sve to htio smjestiti koristio sam *BeginHorizontal()* i *EndHorizontal()* metode koje su napravile blok koji smješta vrijednosti jednu do druge u istoj liniji.



Na kraju sam kao dodatak dodao mogućnosti spremanja i učitavanja te otvaranje mjesta datoteke što bi omogućilo korisniku ručno mijenjanje podataka.

Kako bi se podaci osvježili bilo je potrebno pozvati `Repaint()` metodu koja omogućava cijelom editoru da osvježava vrijednosti u stvarnom vremenu.

```

public override void OnInspectorGUI()
{
    base.OnInspectorGUI();
    if (inventoryReferenceP.Inventory != null)
    {
        try
        {
            foreach (var kvp in inventoryReferenceP.Inventory)
            {
                EditorGUILayout.Space();
                EditorGUILayout.BeginHorizontal();
                EditorGUILayout.LabelField($"ID: {kvp.Key} Name: {kvp.Value.ItemName} Amount: {kvp.Value.AmountInInventory} ");
                if (GUILayout.Button("+"))
                {
                    inventoryReferenceP.Add(kvp.Value);
                }
                if (GUILayout.Button("-"))
                {
                    inventoryReferenceP.Remove(kvp.Value);
                }
                EditorGUILayout.EndHorizontal();
            }
        }
        catch (InvalidOperationException e) Debug.Log($"You've erased all items of this name.");
    }

    EditorGUILayout.Space();

    EditorGUILayout.BeginHorizontal();
    if (GUILayout.Button("Save"))
    {
        _dataMangement.Save();
    }

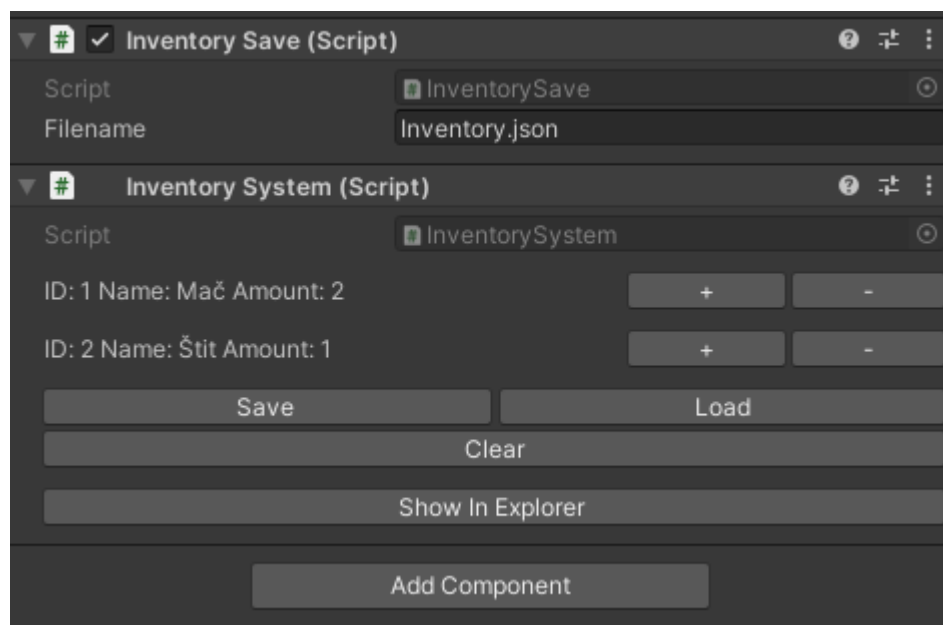
    if (GUILayout.Button("Load"))
    {
        _dataMangement.Clear();
        _dataMangement.Load();
    }
    EditorGUILayout.EndHorizontal();

    if (GUILayout.Button("Clear"))
    {
        _dataMangement.Clear();
    }

    EditorGUILayout.Space();

    if (GUILayout.Button("Show In Explorer"))
    {
        _dataMangement.ShowExplorer();
    }
    Repaint();
}

```



3.28 Prikaz Sadržaja inventory-a te ime datoteke u koju se sprema unutar Editor.

### 3.3.2 EWInventoryAdd

*EWInventoryAdd* služi kao umjetna vanjska skripta, tj. klasa koja omogućava korisniku da preko glavnog izbornika otvori posebnu formu kojom može unositi nove vrijednosti u bilo koji *inventory* unutar jedne scene.

Prvo sam definirao klasu *EWInventoryAdd* koja nasljeđuje klasu *EditorWindow*. Zatim sam definirao attribute te klase koji će biti potrebni za spremanje vrijednosti kod forme.

```
public class EWInventoryAdd : EditorWindow
{
    InventorySystem inventory;

    string _itemName;
    AItem.ItemType _type;
    int _buyVal;
    int _sellVal;

    bool shouldSave = false;
    bool hasSaved = false;

    InventorySave _dataMangement;

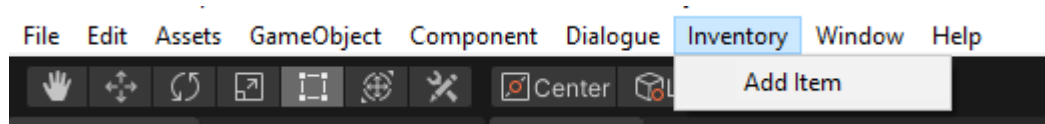
    ...
}
```

3.29 Definicija klase *EWInventoryAdd* i njezinih atributa

Kako je ovo klasa koja nasljeđuje *EditorWindow* klasu, biti će potrebno je otvoriti. Kao što sam rekao već prije, otvaranje ove klase se vrši pomoću glavnog izbornika. Napravio sam metodu *ShowWindow()* te je označio s *MenuItem* oznakom koja je potrebna programu da napravi novu vrijednost na glavnom izborniku. I na kraju sam postavio osnovne vrijednosti novog prozora (ime i veličinu prozora).

```
[MenuItem("Inventory/Add Item")]
public static void ShowWindow()
{
    EditorWindow window = GetWindow(typeof(EWInventoryAdd), true, "Add Item to
Inventory");
    window.maxSize = new Vector2(500, 350f);
    window.minSize = window.maxSize;
}
```

3.30 ShowWindow() metoda služi za otvaranje prozora forme.



3.31 Prikaz dugmeta koje otvara formu pomoću glavnog izbornika

Kada je forma otvorena, može se vidjeti naslov *Add Item* zajedno s još nekoliko polja za upise podataka. Prvo polje je posebno odijeljeno te se tamo pridružuje *GameObject* u sceni koji na sebi sadrži komponentu (tj. skriptu) *InvenotrySystem*. Zatim imamo postavke za „stvar“ ili *item*, tj. vrijednost koju želimo dodati → Ime, tip te prodajna i kupovna vrijednost. Na kraju imamo opciju za spremanje (ova opcija omogućava formi da pri dodavanju nove stvari u spremište odmah i spremi trenutne vrijednosti, no ako je ta opcija uključena ne može se promijeniti *inventory* jer dođe do problema sa serializacijom, tj. spremanja objekta). Nakon upisivanja cijele forme, preostaje samo pritisnuti dugme te spremiti trenutnu vrijednost u spremište.

```

private void OnGUI()
{
    #region Title
    GUILayout.Space(15);
    GUILayout.Label("Add Item", new GUIStyle()
    {
        fontSize = 20,
        normal = new GUIStyleState() { textColor = Color.white },
        alignment = TextAnchor.MiddleCenter
    }));

    EditorGUILayout.Space(50);
    #endregion
    EditorGUILayout.LabelField("Inventory Settings", EditorStyles.boldLabel);
    if (hasSaved)
    {
        EditorGUILayout.LabelField("Reopen window to add item to another inventory!",
        EditorStyles.boldLabel);
        EditorGUILayout.LabelField("Character inventory", inventory.gameObject.name);
    }
    else
    {
        inventory = (InventorySystem)EditorGUILayout.ObjectField("Character inventory",
        inventory, typeof(InventorySystem), true);
    }

    EditorGUILayout.Space(10);
    #region Item Settings
    EditorGUILayout.LabelField("Item Settings", EditorStyles.boldLabel);

    _itemName = EditorGUILayout.TextField("Item Name", _itemName);
    _type = (AItem.ItemType)EditorGUILayout.EnumPopup("Primitive to create:", _type);
    _buyVal = EditorGUILayout.IntField("Buy Value", _buyVal);
    _sellVal = EditorGUILayout.IntField("Sell Value", _sellVal);
    #endregion
    EditorGUILayout.Space(10);
    shouldSave = EditorGUILayout.Toggle("Should save", shouldSave);
    //Add Button Part
    float button_height = 50;
    if (GUI.Button(new Rect(0, maxSize.y - button_height, maxSize.x, button_height),
    "Add new item"))
    {
        try
        {
            inventory.Add(new Item(0, _type, _itemName, _buyVal, _sellVal, 1));
            if (shouldSave)
            {
                inventory.PrintInventory();
                _dataMangement.Save();
                hasSaved = true;
            }
        }
        catch (NullReferenceException) Debug.LogError("You have to assign the inventory
        object first!");
    }
}

```

Add Item to Inventory

Add Item

Inventory Settings

Character inventory

None (Inventory System)

Item Settings

Item Name

Primitive to create:

WEAPON

Buy Value

0

Sell Value

0

Should save

Add new item

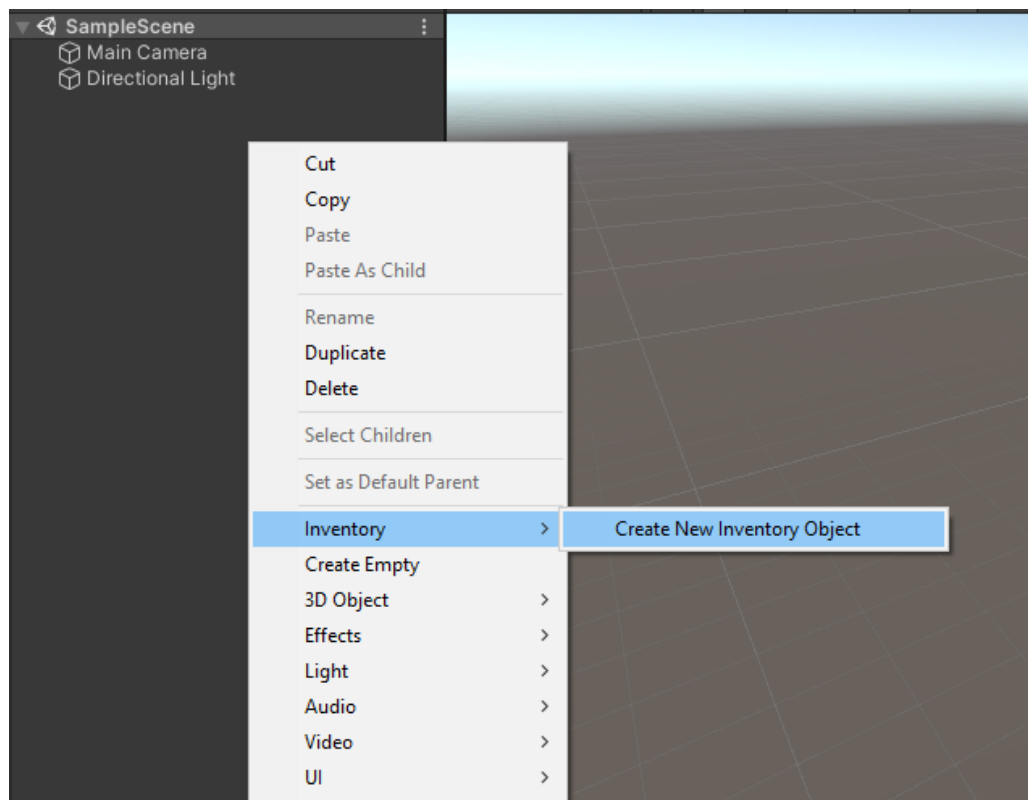
3.33 Osnovni izgled forme za dodavanje vrijednosti u inventory

### 3.3.3 InventoryObjectCreation (dodatak sustavu)

Kako bi cijeli ovaj sustav radio potrebno nam je dodati komponente na *GameObject*, no kako bi se olakšalo korisniku da ne mora tražiti točne komponente za dodati već imati objekt kojim se izrađuje novi objekt putem *context*<sup>10</sup> izbornika, dodao sam ovu klasu.

```
public class InventoryObjectCreation
{
    [UnityEditor.MenuItem("GameObject/Inventory/Create New Inventory Object", false, 0)]
    public static void CreateNewInventoryObject()
    {
        GameObject go = new GameObject("Inventory Object");
        go.AddComponent<InventorySystem>();
    }
}
```

3.34 Definicija klase *InventoryObjectCreation* i metode *CreateNewInventoryObject()*



3.35 Prikaz context izbornika s opcijom za izradu novog inventory objekta

<sup>10</sup> Context izbornik jest tip izbornika koji se pojavi kada korisnik negdje klikne desnim klikom.



## 4 Implementaciju sustava u vanjskim projektima

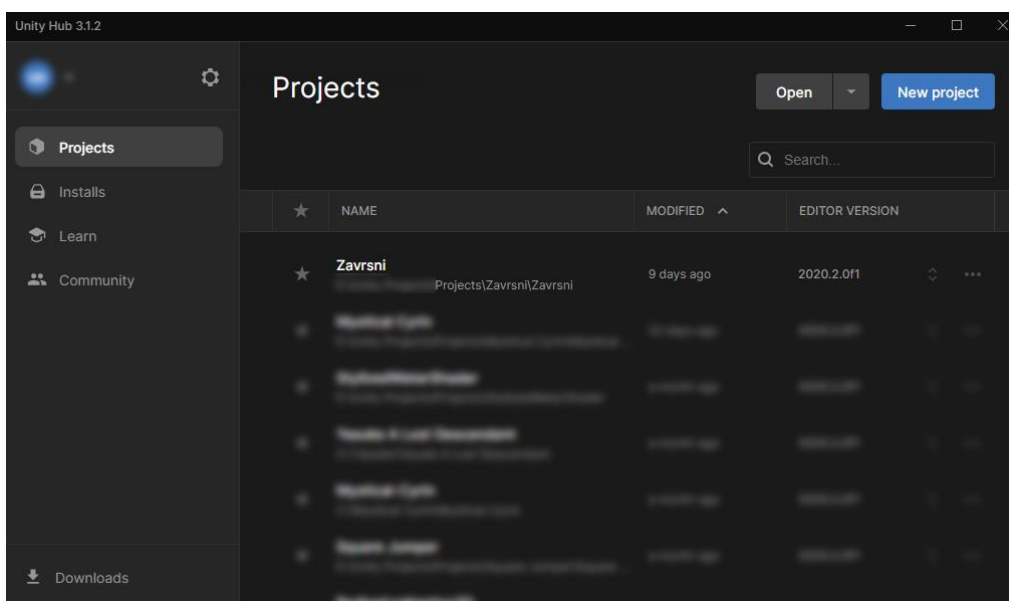
Na ideju za rad ovog sustava sam došao radeći na pravim igrama. S obzirom da sam radio na više mobilnih igara, i nekim računalnim/konzolnim (npr. *Monkey Idle Arm Extender*, *YASUKE: A Lost Descendant*, *Mystical Cyryn*, *Retribution of the Sick*) te kako je u svakoj od ovih igara bio potreban nekakav sustav spremanja, uvijek sam koristio ovaj serijalizacijski pristup.

### 4.1 Ubacivanje gotovog sustava u *Unity*

Kako bi se ovaj sustav ubacio u novi *Unity* projekt bilo je potrebno izvesti projekt u određenu datoteku. To sam napravio tako što sam sve, već prije navedene skripte i datoteke označio pretvorio u datoteku *inventory.unitypackage*. No kako bi se uvele datoteke, prvo je potrebno imati nekakav projekt ili ga napraviti.

#### 4.1.1 Izrada novog *Unity* projekta

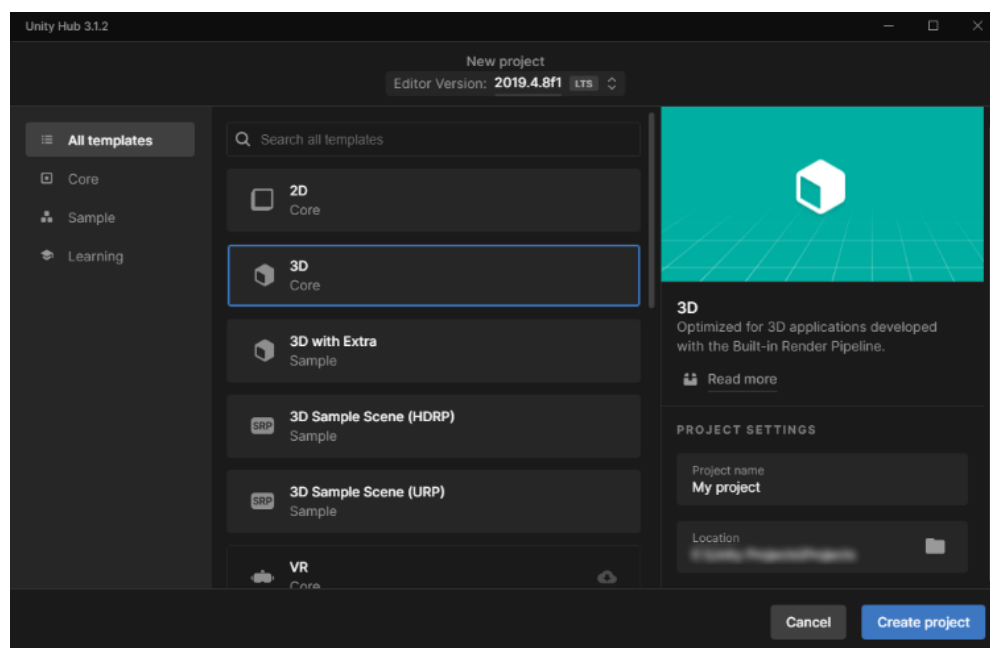
Izrada novog *Unity* projekta vrlo je lagan proces za korisnika. Potrebno je otvoriti *Unity Hub* gdje se prikazuju svi projekti koje ste napravili.



4.1 Prikaz *Unity Hub*-a

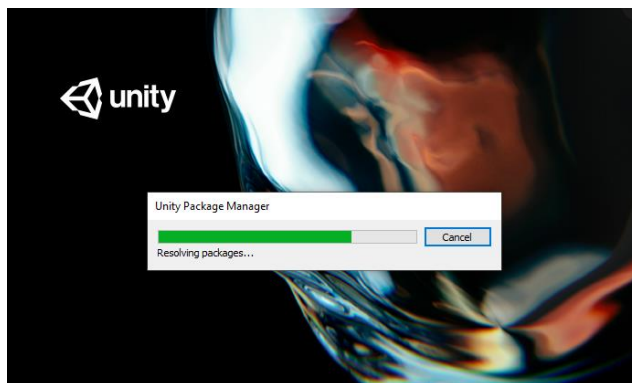
Kada je *Unity Hub* otvoren, potrebno je kliknuti na *New Project*. Ovdje je kratka forma koju treba popuniti s podacima tako što odabirete tip teme ili nekakvog predloška, dajete ime projektu

te mu se zadaje lokacija na koju će projekt biti spremljen. U svojem ću slučaju sve vrijednosti pustiti na već zadanim postavkama te odabrati opciju *Create*.



4.2 Prikaz New Project forme

S klikom na opciju *Create* ispasti će novi prozorčić koji nalaže da se projekt krenuo izrađivati te ako se bolje pogleda, može se vidjeti koje su trenutne datoteke koje se dodaju u novi projekt.

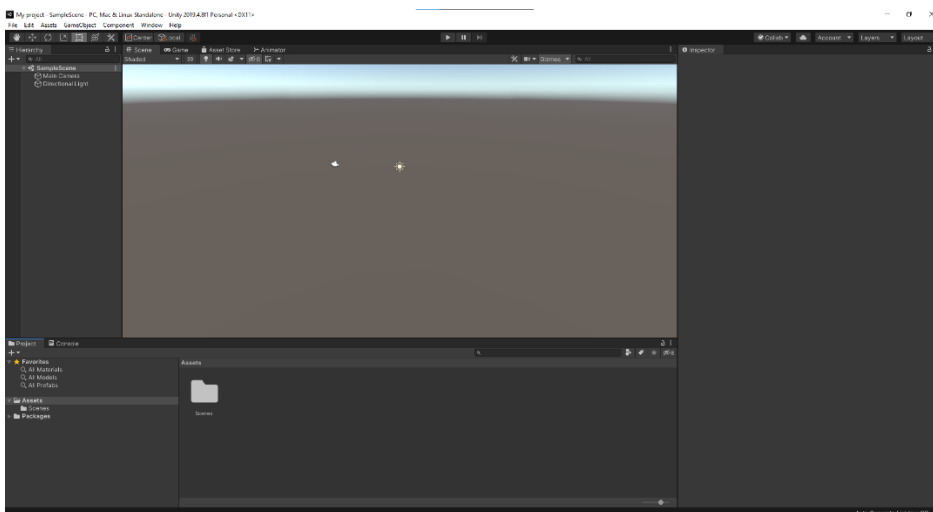


4.3 Slika izrade novog projekta

#### 4.1.2 Dodavanje sustava u projekt (princip *Unity* paketa)

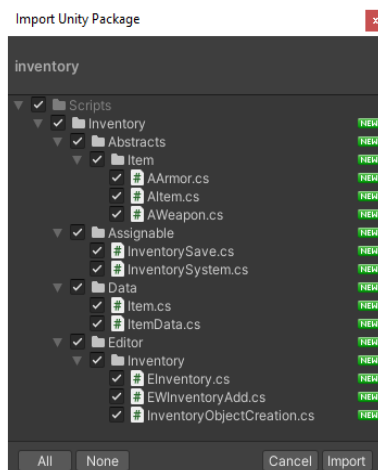
Sada kada imamo novi *Unity* projekt, možemo dodati nove pakete, izraditi nove skripte, animacije, modele, slike i sl. Možemo vidjeti da je projekt podijeljen na četiri manja prozora koja čine osnovni izgled *Unity Editor*a, a to su *Inspector*, *Project*, *Console* i *Hierarchy*. Naravno uvijek

se mogu dodavati dodatni prozori pod glavni izbornik → *Window* koji mogu još dodatno pomoći pri izradi igre jer sadrže sustav za zvuk, video, slike, *VFX*<sup>11</sup> i sl.



#### 4.4 Prikaz novog Unity projekta unutar Unity Editora

Sustav je vrlo lagano ubaciti u bilo kakav *Unity* projekt. Datoteku *inventory.unpackage* se ili treba povući u dio za raspored datoteka u sustavu ili se treba otići pod *Assets* → *Import Package* → *Custom Package* i pojaviti će se prozor *Import Unity Package* prozor. Kada je taj prozor učitao sve što je potrebno jest kliknuti na *Import* dugme i sve će se automatski ubaciti u projekt.

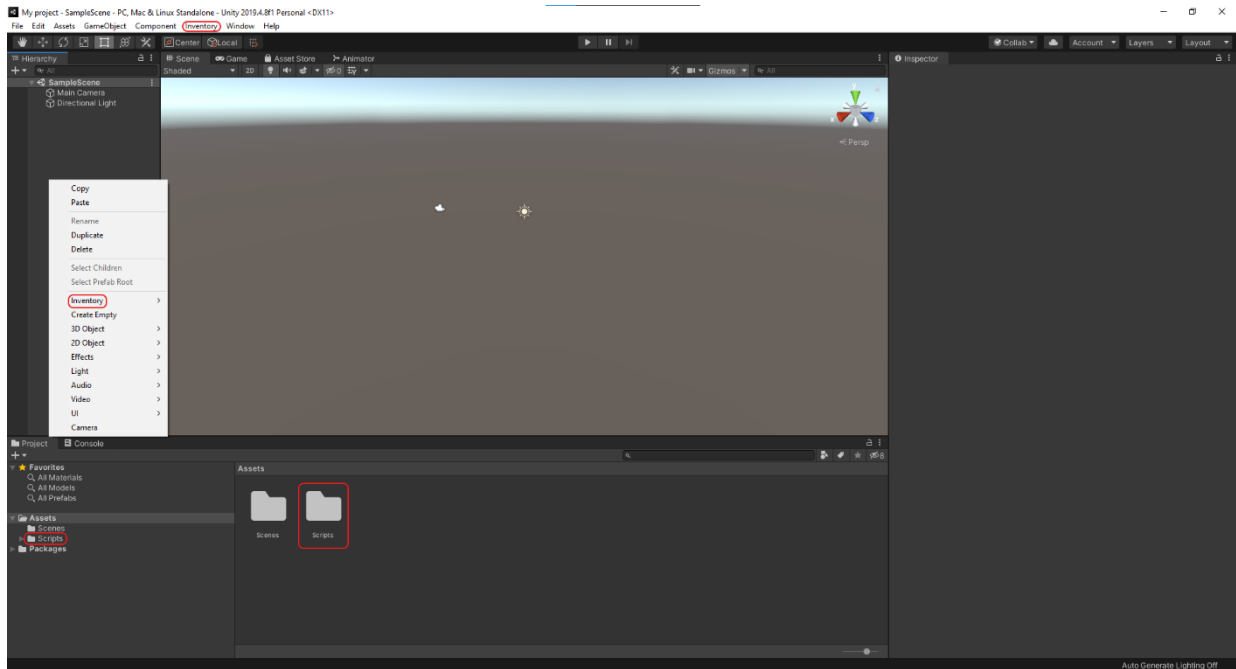


#### 4.5 Prikaz dodavanja inventor.unpackage datoteke u Unity

<sup>11</sup> *VFX* (eng. *Visual Effects*) jest proces mijenjanja ili izrade nekakve slike ili 3D modela koji služi da se integrira zajedno sa scenom i uljepša igru (npr. kad lik zamahne mačem, mač se zapali i sl.)

Nakon što je paket ubačen u projekt moći ćemo vidjeti da su se pojavile tri nove stvari:

- Glavni izbornik je promijenjen → dodan mu je *Inventory* izbornik
- Context izbornik unutar hierarhije je promijenjen → dodan mu je *Inventory opcija*
- Dodan je *Scripts* direktorij unutar *Project* prozorčića



4.6 Prikaz izmjene projekta nakon što je paket dodan.

## 5 Zaključak

Programiranje skripti za Unity igru nastojao sam prikazati na primjeru programiranja sustava za spremište (*Inventory system*), koji predstavlja jedan od ključnih sustava za rad modernih video igara. Sustav sam pri izradio na način koji bi omogućavao drugim korisnicima *Unity*-a da se njime koriste kao vanjskim paketom te pomoću toga unaprijeđuju svoje igre. Svaki od dijelova sustava sam posebno definirao i povezao ih u cjelinu tako da:

- Apstraktni dio sadrži glavnu srž objekta kojom se sustav koristi, to sam realizirao pomoću klase *AItem*.
- U dodjeljivi dio sam postavio logiku sustava koja korisniku omogućava interakciju sustava te spremanje i učitavanje podataka iz sustava. Za to sam izradio skripte *InventorySystem.cs* te *InventorySave.cs*.
- Podatkovni dio je čvrsto vezan uz dodjeljivi dio jer mu omogućava rad s podacima pomoću *Item* i *ItemData*.
- Dio za uređivanje služi za korisniku dostupniji prikaz cijelog sustava unutar korisničkog sučelja *Unity*-a.

Naime, u pojedinoj igrici likovi, neprijatelji, ne-igrajući likovi imaju taj sustav ugrađen u sebi. Radi se o jednostavnom sustavu koji se uz sitne promjene može primijeniti na bilo kakvu vrstu aplikacija. Bila aplikacija namijenjena za učenje, igranje, simulaciju, kada god je u pitanju spremanje u neakvu vrstu baze podataka koja je potrebna korisniku, ovaj projekt/paket se može iskoristiti. Osobno me privlače igrice, ne samo za igranje već volim raditi i na njihovu nastajanje. Pri izradi određenih, već gore spomenutih igrica, a i već gotovih igrica velikih studija i kompanija (kao što su na primjer *Resident Evil 4*, *Zelda*, *Xenoblade Chronicles*, i slične) uočio sam da je takav sustav vrlo koristan u igrici. Stoga sam nastojao programirati sustav prtljage koji bi se uz male preinake i prilagodbe mogao koristiti za različite igrice i projekte. Naravno, uvijek postoje i mogućnosti poboljšanja i usavršavanja postojeće skripte te njezina prilagođavanja i za ostale aplikacije.

## 6 Literatura

1. Unity Technologies, 2022. Unity - Scripting API:. [online] Docs.unity3d.com. Dostupno na: <<https://docs.unity3d.com/ScriptReference/>> [Pristupljeno 8. Siječnja 2022.]
2. Tadres, A., Extending Unity with editor scripting. Packt Publishing., Birmingham-Mumbai, 2015.
3. Wells, R., Unity 2020 by Example. Packt Publishing, Birmingham-Mumbai, 2020.
4. En.wikipedia.org. 2022. Unity (game engine) - Wikipedia. [online] Dostupno na: <[https://en.wikipedia.org/wiki/Unity\\_\(game\\_engine\)](https://en.wikipedia.org/wiki/Unity_(game_engine))> [Pristupljeno 15. Svibnja 2022].
5. Kids' Coding Corner. 2022. 10 Top Games Made with Unity: Unity Game Programming. [online] Dostupno na: <<https://www.create-learn.us/blog/top-games-made-with-unity/>> [Pristupljeno 15. Svibnja 2022].

## 7 Prilozi

2.1 Današnji logo Unity-a.....	4
3.1 Prikaz organizacije projekta u foldere .....	7
3.2 Definicija klase AItem.....	7
3.3 Definicija atributa klase AItem.....	8
3.4 Definicija svojstava klase AItem.....	8
3.5 Start() metoda klase AItem.....	9
3.6 Definicija klase InventorySystem.....	9
3.7 Definicija inventory-a.....	10
3.8 Add() metoda klase InventorySystem .....	11
3.9 Remove() metoda klase InventorySystem.....	11
3.10 GetKeyFromValue() metoda klase InventorySystem .....	12
3.11 Prikaz vrijednosti inventory-a u konzoli. ....	13
3.12 PrintInventory() dodatna metoda klase InventorySystem za svrhe debugiranja .....	13
3.13 Definicija klase InventorySave.....	14
3.14 Definicija atributa klase InventorySave .....	14
3.15 Prikaz dviju klasa (InventorySave i InventoryData) unutar jedne skripte.....	15
3.16 Klasa ItemData .....	16
3.17 Prikaz lokacije Path datoteke na disku .....	17
3.18 Prikaz greške koja se dogodi u slučaju da je string filename prazan. ....	17
3.19 Prikaz JSON-a jednog inventory-a.....	18
3.20 Save() metoda klase InventorySave .....	19
3.21 Load() metoda klase InventorySave.....	20
3.22 Clear() dodatna metoda klase InventorySave za svrhe debugiranja.....	21
3.23 ShowExplorer() dodatna metoda klase InventorySave za svrhe debugiranja .....	21
3.24 Definicija klase EInventory .....	22
3.25 Definicija instanci klasa InventorySystem i InventorySave.....	23
3.26 OnEnable() metoda editor klase za InventorySystem .....	23
3.27 OnInspectorGUI() metoda editor klase za InventorySystem .....	25
3.28 Prikaz Sadržaja inventory-a te ime datoteke u koju se sprema unutar Editoru. ....	26
3.29 Definicija klase EWInventoryAdd i njezinih atributa .....	27
3.30 ShowWindow() metoda služi za otvaranje prozora forme. ....	28
3.31 Prikaz dugmeta koje otvara formu pomoću glavnog izbornika.....	28
3.32 OnGUI() metoda koja služi za crtanje forme i dodavanje vrijednosti u inventory .....	29
3.33 Osnovni izgled forme za dodavanje vrijednosti u inventory .....	30
3.34 Definicija klase InventoryObjectCreation i metode CreateNewInventoryObject() .....	31
3.35 Prikaz context izbornika s opcijom za izradu novog inventory objekta.....	31
4.1 Prikaz Unity Hub-a.....	32
4.2 Prikaz New Project forme.....	33
4.3 Slika izrade novog projekta .....	33
4.4 Prikaz novog Unity projekta unutar Unity Editoru.....	34

4.5 Prikaz dodavanja inventor.unitypackage datoteke u Unity .....	34
4.6 Prikaz izmjene projekta nakon što je paket dodan. ....	35



Mentor je prihvatio izradbu: \_\_\_\_\_

(potpis)

Datum predaje rada: \_\_\_\_\_

Ocjena pisanog rada: \_\_\_\_\_

Datum obrane rada: \_\_\_\_\_

Ocjena obrane rada: \_\_\_\_\_

**Konačna ocjena:** \_\_\_\_\_

Povjerenstvo:

1. mentor: \_\_\_\_\_

2. profesor struke: \_\_\_\_\_

3. profesor struke: \_\_\_\_\_

Prostor za izdvojeno mišljenje ili eventualni komentar:

---

---

---

Konzultacijski list za učenika : \_\_\_\_\_, razred: \_\_\_\_\_

<b>Rb.</b>	<b>Datum konzultacija</b>	<b>Bilješke o napredovanju</b>	<b>Potpis mentora</b>