Univerzitet u Kragujevcu

Prirodno-matematički fakultet

Institut za matematiku i informatiku

Naziv radionice: Gaming With Unity

Tema: Goblins - Unity 2D Game

Studenti:

Bojan Đokić

Jovana Radovanović

Jovana Vučićević

Miloš Pavlović

Strahinja Milosavljevic

Ognjen Tanasijevic

Lazović Aleksandar

Mentori:

dr Ana Kaplarević-Mališić

dr Višnja Simić

Sadržaj:

Poglavlje1. Uvodni deo

1.1. Sumiranje ciljeva

S obzirom na ubrzano razvijanje sveta igara i korišćenje Unity3D platforme za kreiranje istih kao i velike moći jednostavnog editora za pomaganje u razvijanju 3D igrica, svet je polako počeo ponovo da vraća u modu 2D igrice tako da je Unity napravio razne modifikacije i uveo nove opcije za kreiranje i ovakvih tipova igrica čime je proširen opseg mogućnosti i ideja.

Ako tražite nešto što će Vam pomoći u kreiranju jednosatvne 2D igrice RPG onda je ovo uputstvo baš ono što Vam treba. Počećemo od same nule i kreiraćemo heroja, svet i protivnike. Upoznaćete se sa nekim novim tehnikama i trikovima koji će Vam pomoći u daljem usavršavanju vaših sposobnosti. Naučićemo kako se pravi RPG frejmvork kao i kodiranje u C# kako bismo izvukli najbolje od Unitija i jednostavno dostigli željene rezultate.

Kada proučite kompletnu skriptu bićete bolje upoznati sa nekim novijim i usavršenijim pogledima na svet kreiranja i dizajniranja igriča, pa čak iako niste možda bili previše „zagrejani“ za ovakve stvari posle ove knjige ćete sigurno videti kakve Vam sve čari pruža Unity.

Napomena: Nijedan goblin nije povređen tokom pisanja ove knjige!

1.2. Šta pokriva ova knjiga

Chapter 1. i 2. Pokriva neke osnove i ciljeve ove knjige kao i uvod u Unity

Chapter 3. Kreiranje glavnog junaka, rad sa sprajtovima

Chapter 4. Animacije, Kontroleri animacija i stanja

Chapter 5. Kreiranje okruženja , rad sa kamerom i planiranje veće slike

Chapter 6. Interakcija, razgovor sa drugim likovima i putovanje nadalje

Chapter 7. Kreiranje mape i navigacije kroz nju

Chapter 8. Nailaženje na neprijatelje, interakcija sa goblinima

Chapter 9. Inventar

Chapter 10. i 11. Biranje oružja i Borba

1.3. Šta Vam je potrebno za ovu knjigu

Poreban Vam je softver Unity koji možete preuzeti sa linka:
https://unity3d.com/get-unity/download

Možete koristiti bilo koju verziju od 4.3 ali savetujemo najnoviju.

Visual Studio ili neki drugi editor u kome ćemo pisati C# skripte.

Treba da imate predstavu o nekim osnovnim pojmovima kao što su GameObject , komponente i slično.

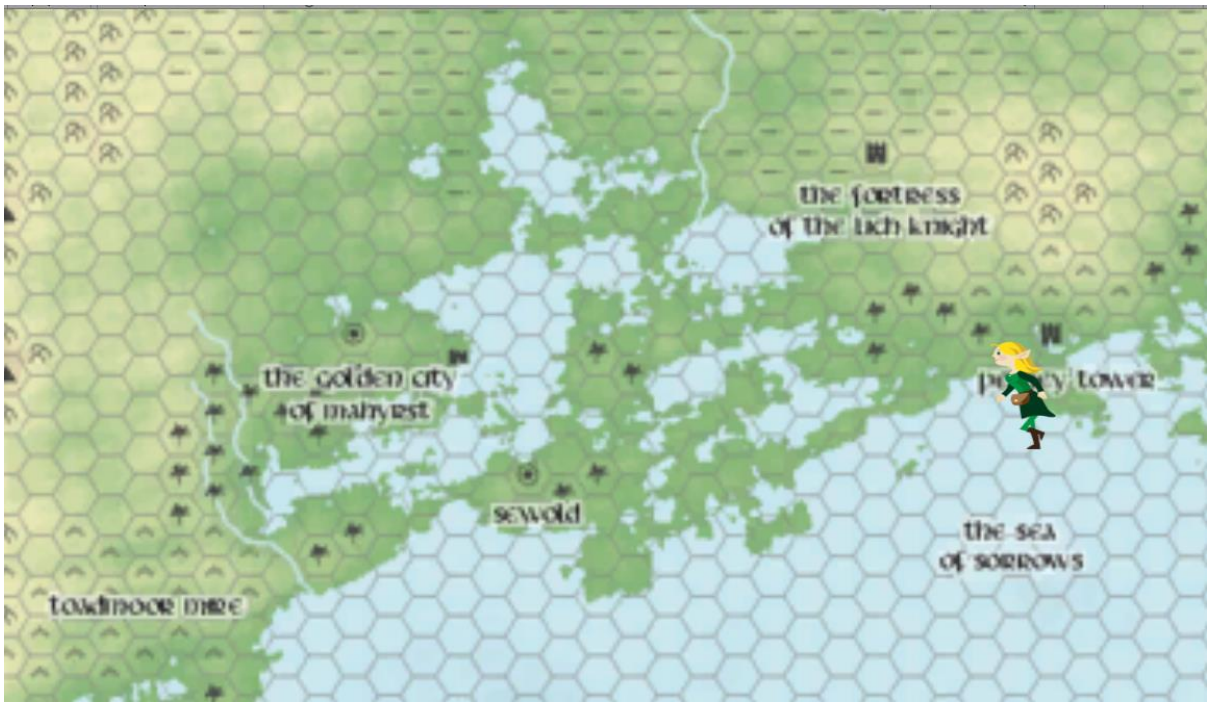
1.4. Pregled krajnjeg proizvoda

Glavni naš zadatak je da kreiramo potpuno funkcionalnu **RPG** (Role Playing Game) igricu koja će imati sve prethodno navedene pogodnosti.

Posetićemo neka mesta kao što je ovo:

- Tvoj grad, u kome ćeš moći da se krećeš:

- Mapa sveta u kome ćes moći da biraš gde dalje da putuješ:



- Borba sa goblinima:

- Kupovina oružija u prodavnici:

1.5. Šta je Unity?

Unity je game engine uz pomoć koga se prave 2D i 3D igre za PC, konzole, mobilne uređaje i Web sajtove.

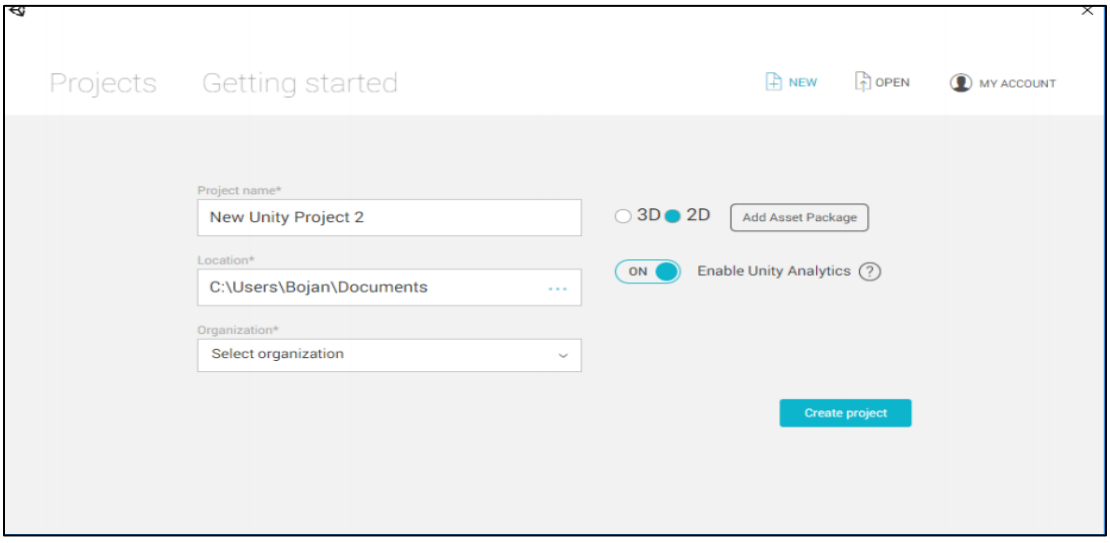
Game engine je softver koji pruža osnovne funkcije koje se često koriste za razvoj video igara. On brine o osnovnim zadacima kao što su iscrtavanje, detekcija i odgovor na koliziju, korisnički unos i slično. Game engine-i imaju relativno kratku istoriju, koja počinje 1990-tih, i u konstantnom su razvoju od tada.

Tipične funkcionalnosti koje pruža game engine su:

- Zvuk
- Skripte - podrška za pisanje novih programskih elemenata u određenom programskom jeziku
- Animacije - 2D i 3D
- Iscrtavanje (eng. Rendering) – 2D i 3D grafika
- Fizička enigma – detekcija kolizije, odgovor na koliziju
- Mrežni rad
- Upravljanje memorijom

1.6. Okruženje Unity programa

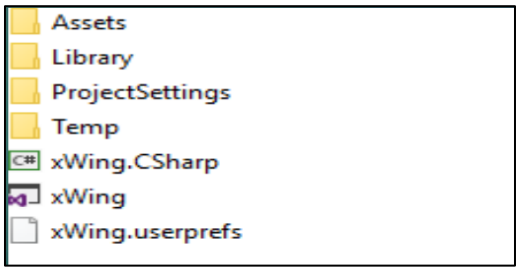
Nakon startovanja Unity programa otvara se prozor za kreiranje novog projekta, gde unosimo naziv projekta, biramo lokaciju na kojoj čuvamo projekat, i selektujemo 2D opciju. Klikom na dugme *Create Project* otvara se okruženje u kome možemo da počnemo sa kreiranjem igrice.



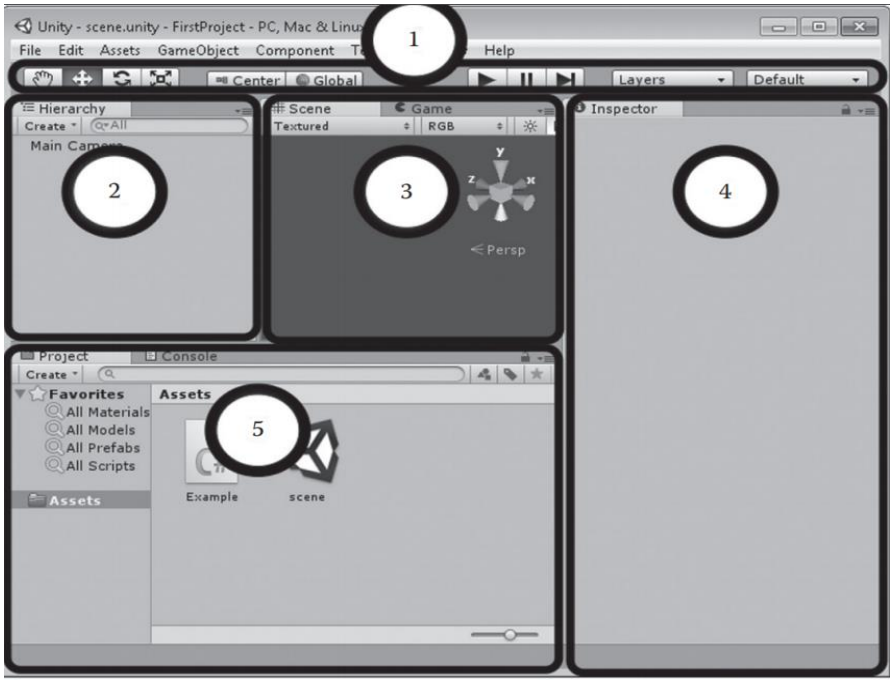
Prilikom kreiranja projekta, Unity u direktorijumu projekta kreira različite fajlove i direktorijume. Ovo omogućava da više ljudi radi na istom projektu, menjajući različite delove projekta.

Unity kreira četiri različita direktorijuma:

1. **Assets**– Primarni direktorijum za objekte igre, kao što su modeli, teksture, zvukovi i skripte. U okviru ovog foldera je preporuka držati fajlove organizovane u raznim direktorijumima, kako se fajlovi ne bi mešali.
2. **Library** – lokalni keš za importovana sredstva.
3. **ProjectSettings** – u ovom folderu se skladište sva podešavanja projekta kao što su fizika, oznake, podešavanja igrača itd.
4. **Temp** – folder za smeštanje privremenih fajlova generisanih tokom bildovanja projekta.



Kada napravimo projekat, otvara se glavni editorski panel, koji je podeljen u pet glavnih delova, kao na sledećoj slici.



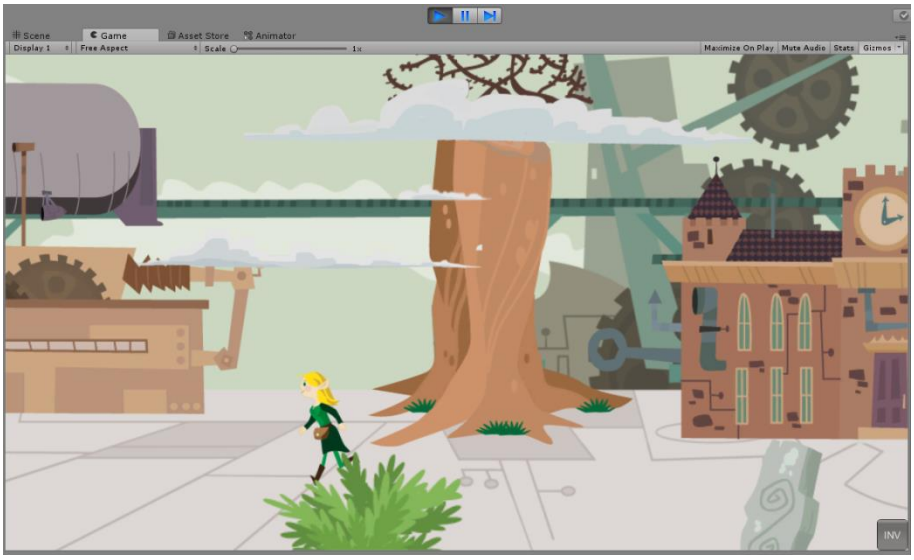
1. Toolbar
2. Hierarchy panel
3. Scene and Game view
4. Inspector panel
5. Project and Console panel

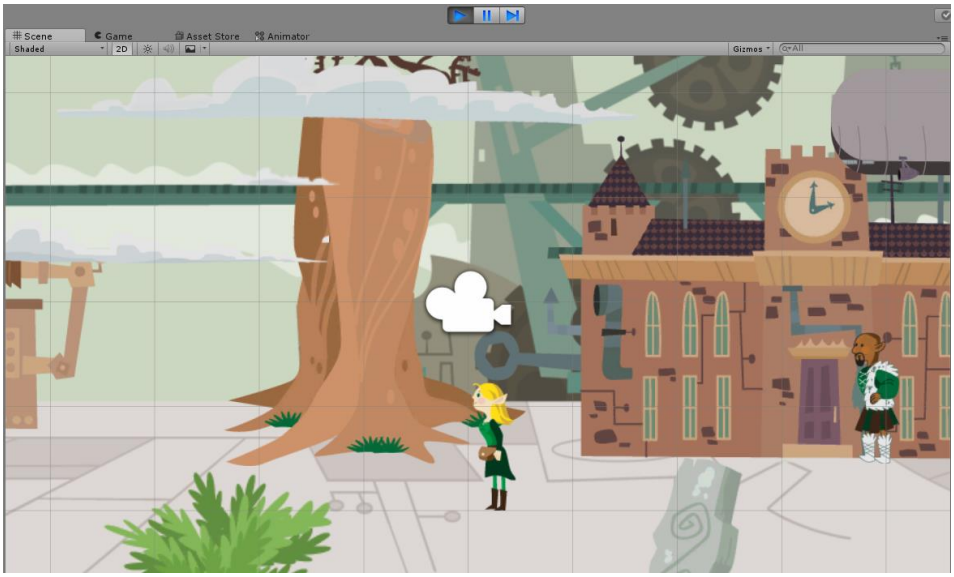
Paneli se lako mogu pomerati u meniju *Window-> Layouts*.

1. **Toolbar**-sastoji se od 5 osnovnih kontrola, svaka povezana sa drugim delom editora. Alati za transformaciju(1 i 2) služi za manipulisanje prikazom scene. Play, Pause i Step tasteri(3) služe za prikaz igre, tj. animiranje scene. Layers(4) služi za odabir objekta koji se prikazuje na sceni. Layouts (5) služi za kontrolu svih prikaza.

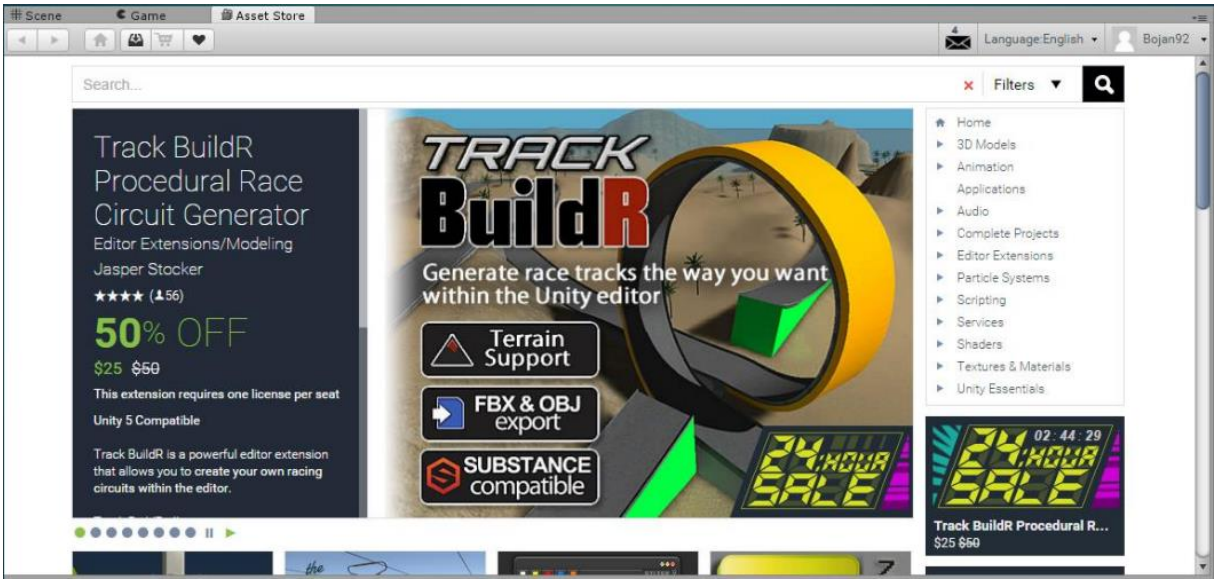
2. **Hijararhija** – sadrži sve objekte projekta u trenutnoj sceni. Unity koristi koncept nazvan *roditeljstvo* (eng. *Parenting*) koji koristimo kada želimo da neki objekat nasledi svojstva drugog objekta. Tada objekat koji nasleđuje nazivamo *dete* (eng. *Child*) dok početni objekat *roditelj* (eng. *Parent*). Nova scena će sadržati samo "Main Camera".

3. **Scene and Game view** – scena je mesto gde se igra izrađuje. Prikazuje koje objekte imamo u igri i gde su u prostoru jedan u odnosu na drugi. Game panel je pogled na igru kroz glavnu kameru (Main Camera).





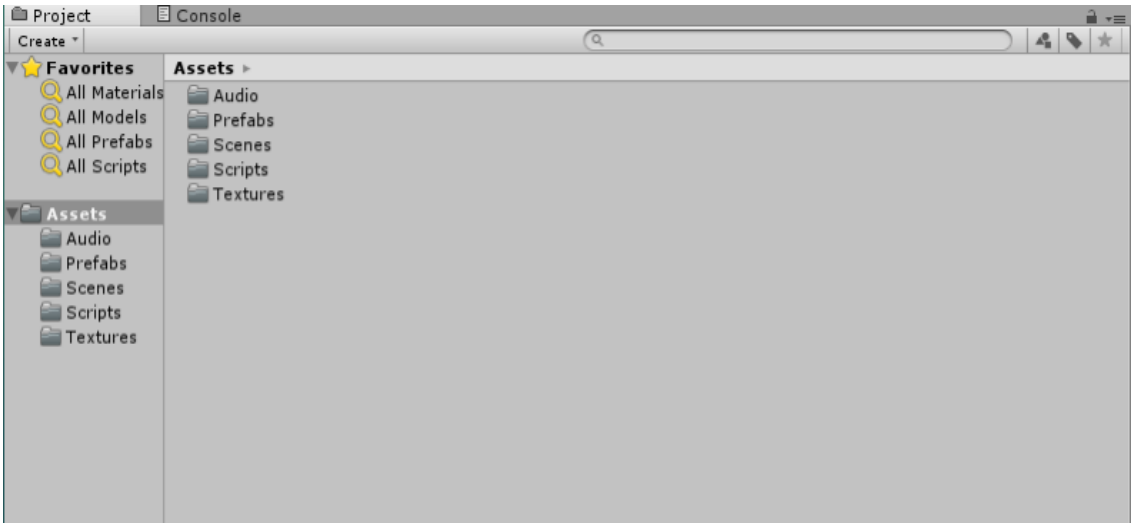
Takođe, u okviru ovog panela možemo naći i Asset Store, gde možemo kupiti ili besplatno skinuti gotove modele, skripte, materijale, audio fajlove i još mnogo toga.



4. **Inspector panel** – mesto gde možemo prilagoditi aspekte svakog elementa koji je na sceni. Kada se odabere objekat u prozoru hijararhije ili dvostruko klikne na objekat u prozoru scene, taj će objekat pokazati svoje atribute u Inspector prozoru, pa odatle možemo upravljati tim atributima.

Dugme Add *Component* nam služi da dodajemo nove komponente objektu. Ovo je i jedan od načina kako ćemo dodavati C# skripte objektu.

5. **Project and Console Panel** – korisnik može prisupiti svim objektima i ostalim datotekama koje su sadržane u projektu. *Project panel* sadrži folder *Assets* koji ima razne objekte koje možemo ubaciti u scenu. Jako je važno držati neku strukturu direktorijuma i deliti objekte u odgovarajuće foldere/podfoldere.

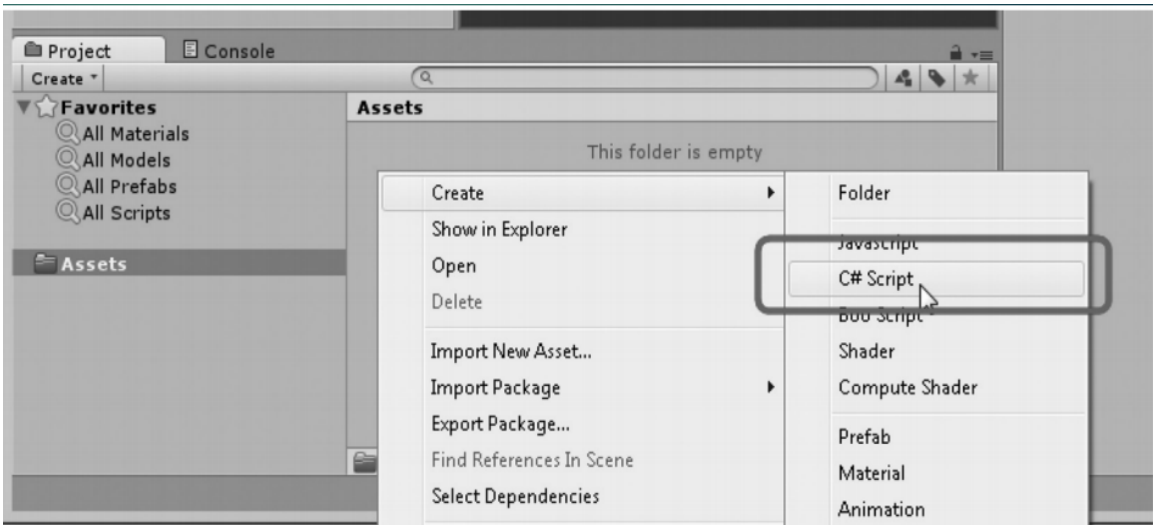


Console panel nam prikazuje sve informacije kreirane u okviru C# koda. Ovde možemo videti podatke ispisane uz pomoć Debug.Log(Podatak), greške i upozorenja.

1.7. C# skripte

Za kreiranje skripti Unity podržava JavaScript, C#, Boo. Skripte su važan i ključan koncept svakog razvoja igre i unutar Unity engine-a se smatraju komponentama. Pomoću skripti možemo upravljati objektima igre, tako što napišemo željeno ponašanje za tu skriptu, i dodelimo je objektu igre.

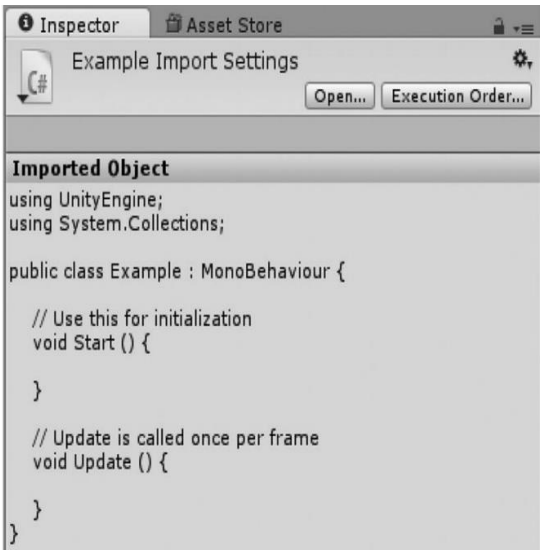
Za programiranje naše igrice koristićemo *C# skripte*. Skriptu možemo kreirati i dodeliti objektu na više načina. Desnim klikom na Project Panel i izaberemo *Create -> C# Script*. Poželjno je napraviti u okviru *Assets* direktorijuma direktorijum*Scripts* za čuvanje skripti.



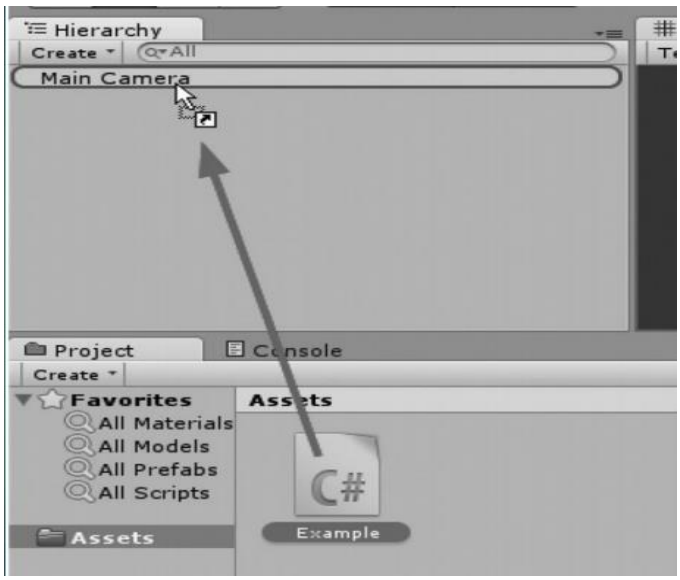
Za Unity se instalira *MonoDevelop* za otvaranje i manipulisanje skripti. Takođe se može koristiti *Visual Studio* ili bilo koji drugi tekstualni editor.

Klikom na kreiranu skriptu možemo videti u okviru Inspector panela kod koji odatle ne možemo da menjamo.

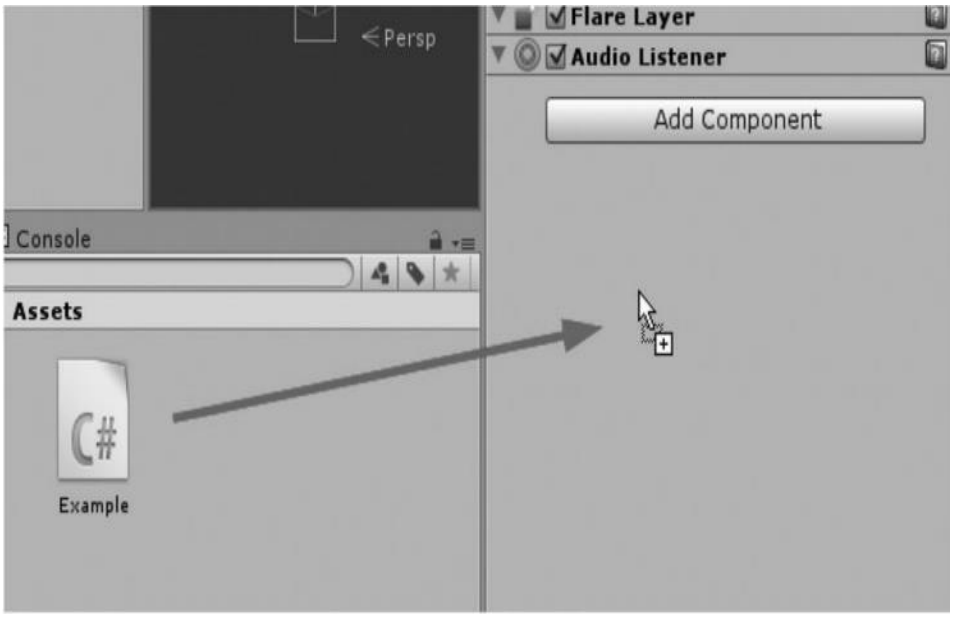
Osnovne metode koje kreira Unity su *Start(),* koja se poziva prilikom pokretanja skripte odmh posle *Update()* metode. Update metoda se poziva za svaki frame. Takođe, možemo koristiti *Awake()* koja se poziva prilikom instanciranja skripte.



Kako bismo dodali skriptu objektu možemo je prevući iz Project panela na objekat na sceni ili u Hierarchy panelu.

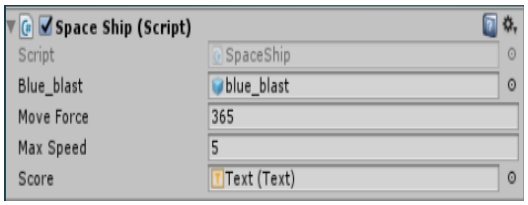


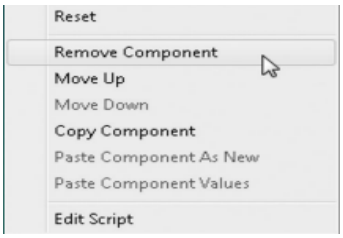
Takođe se može dodati kada je selektovan objekat prevlačenjem na Inspector panel odmah ispod dugmeta *Add Component*.



Jednostavnim klikom na *Add Component*, možemo potražiti skriptu među komponentama.

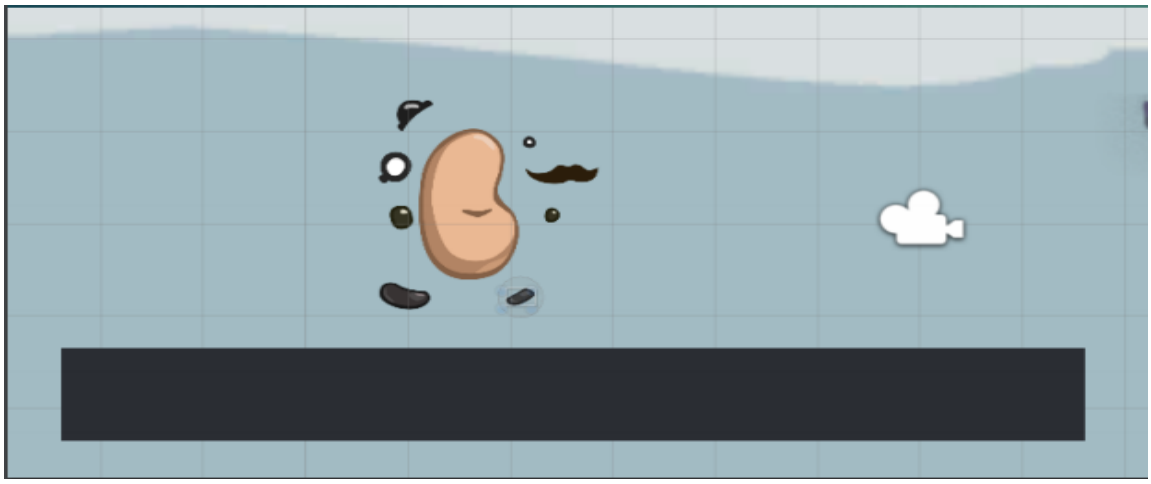
Kada se skripta doda objektu, možemo je videti u okviru Inspector panela. Skripte se mogu lako ukloniti klikom na točkić u desnom ćosku komponente, i na *Remove Component*.



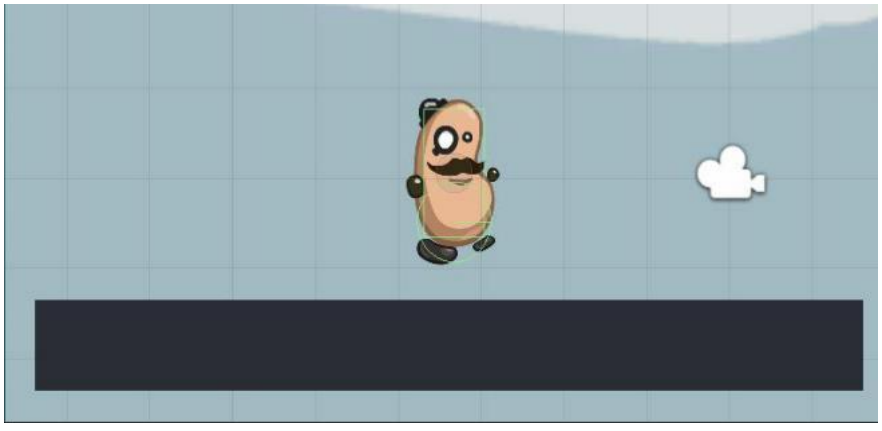


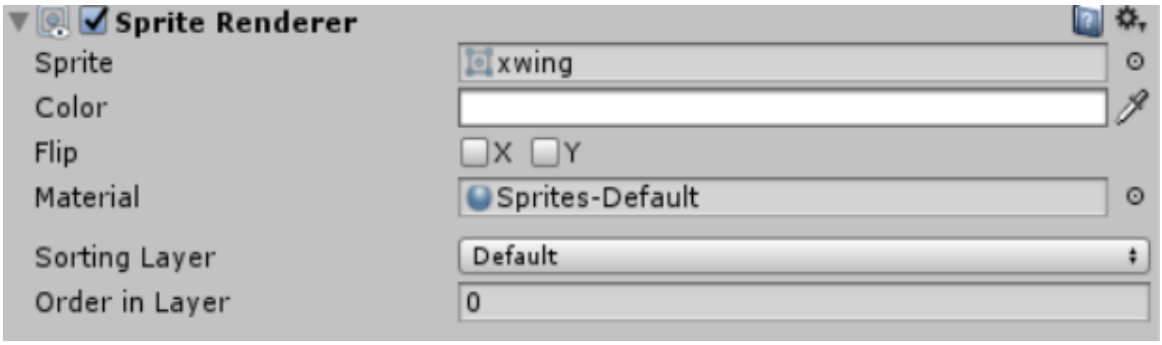
1.8. Sprites

U Unity-u 2D sprite-ovi su jednostavne slike koje uglavnom prikazuju jedan objekat. Nekoliko sprite-ova može činiti jedan objekat u pojedinačnim frejmovima kako bi bila moguća animacija lika. Na slici možemo videti objekat sačinjen od više sprite-ova.



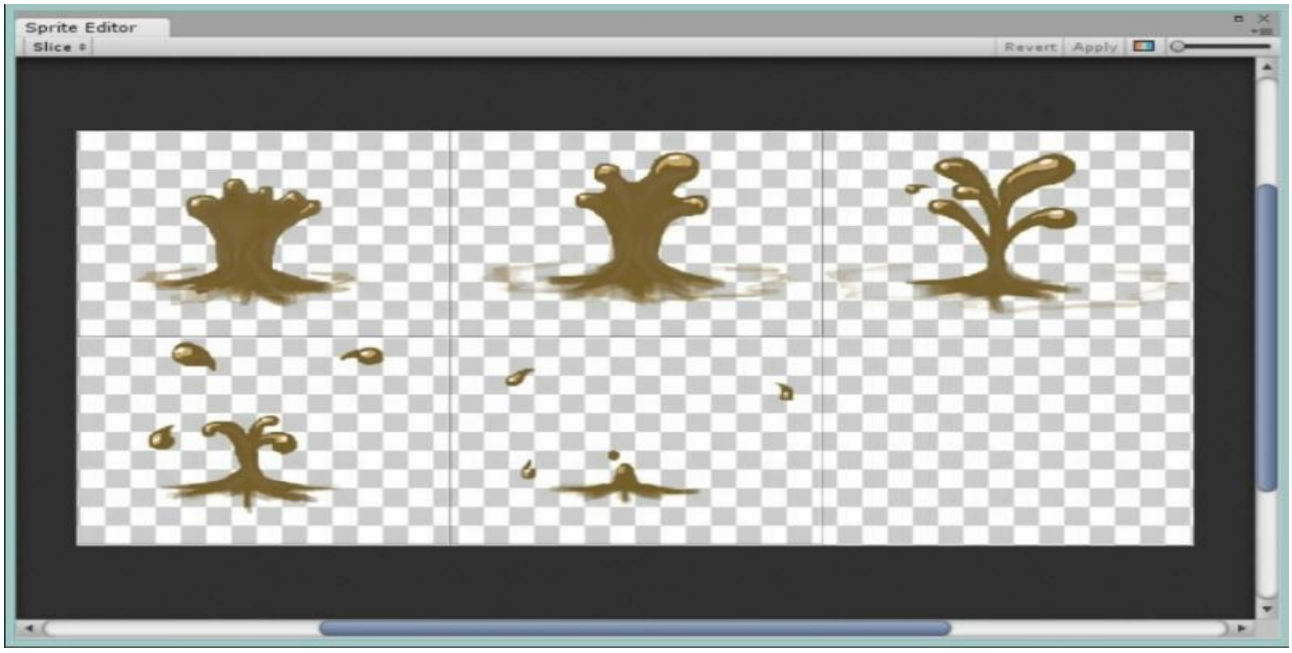
U jezgru 2D Sistema se nalazi novi uvoznik sprite tekstura, koji uvozi vaše teksture i priprema ih kao sprite-ove u vašem folderu gde se nalazi projekat. Kada se prenesu na scenu automatski dobijaju *Sprite Render* objekat koji je spreman da se prikaže u igri.*Sprite Render* je novi 2D pokazivač koji iscrtava sprite-ove na ekran.





U okviru *Sprite Render* komponente možemo videti ime sprite-a, možemo menjati boju, okretati sprite po x i y osi, menjati material sprite-a, birati layer u kome se nalazi i određivati njegov raspored u layer-u.

Spritesheets je ključni deo bilo kog 2D sistema, posebno 2D animacije. Ujedinjavanje svih tekstura u jednu veću teksturu daje bolje performanse prilikom slanja sprite-ova na grafičku karticu, nego slanje puno manjih fajlova. Ovo se odnosi na sledeću sliku:



1.9. Box2D physics system

Box2D je već korišćen za mnoge druge platforme (uključujući *XNA*), tako da je i sa razlogom deo Unity-ja. U većini 2D igara, 4 puta povećava performanse fizike. Međutim, ne možemo očekivati da će rešiti bas sve probleme.

Kroz *Box2D* Unity je ubacio nove opcije za fiziku igre, kao što su:

- Rigidbody 2D
- CircleCollider 2D
- BoxCollider 2D
- PolygonCollider 2D
- Edgecollider 2D

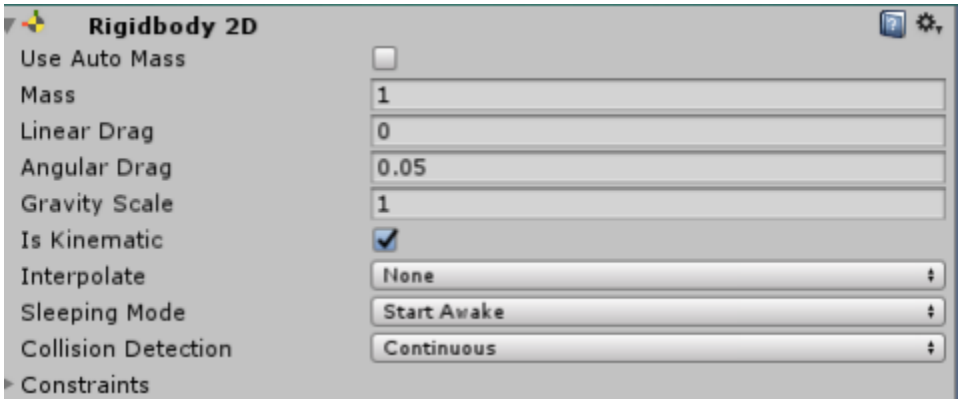
Takođe je ubačena i fizika za kontrolu spajanja dva fizička tela:

- Distance Joint 2D
- Hinge Joint 2D
- Slider Joint 2D
- Spring Joint

1.10. Rigidbody2D

Rigidbody2D komponenta stavlja sprite-ove pod kontrolu engine-a za fiziku. To omogućava da sprite bude pod uticajem gravitacije i može se programirati u skriptama korišćenjem sila.

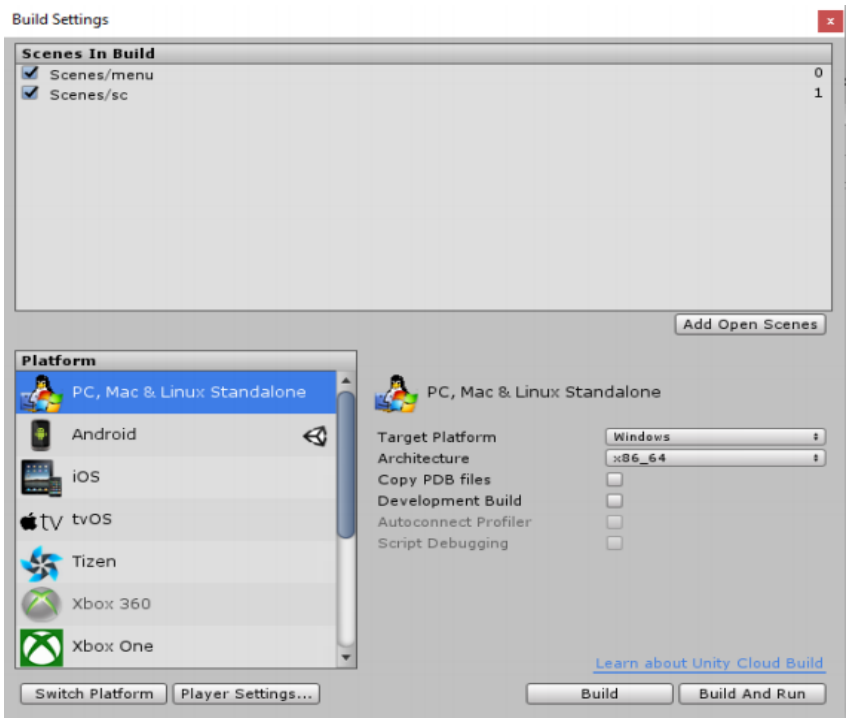
Ako dodamo odgovarajući Collider sprite će reagovati na koliziju sa drugim sprite-ovima, što omogućava metoda *void OnCollisionEnter2D(Collision2D coll).*



- Gravity Scale određuje kojiko jako će gravitacija uticati na objekat
- Is Kinematic određuje da li će na objekat uticati sile i kolizije.

1.11. Build and run

- Kako bismo napravili igru u meni-ju *File -> Build Settings*
- U deo *Scenes In Build* prevlačimo scene koje će biti u igri
- Izaberemo platformu i opciju *Build*



- Za android je neophodno imati *Android SDK* (koji ide uz Android Studio) i u okviru *Player Settings*-a podesiti *Identification* deo, kako na slici ispod:

- *Bundel Identifier* mora biti u formi com.ImeFirme.ImeAplikacije
- *Minimum API Level* mora biti postavljen na najmanju verziju Androida koja može pokrenuti aplikaciju

Poglavlje 2. Kreiranje igrača

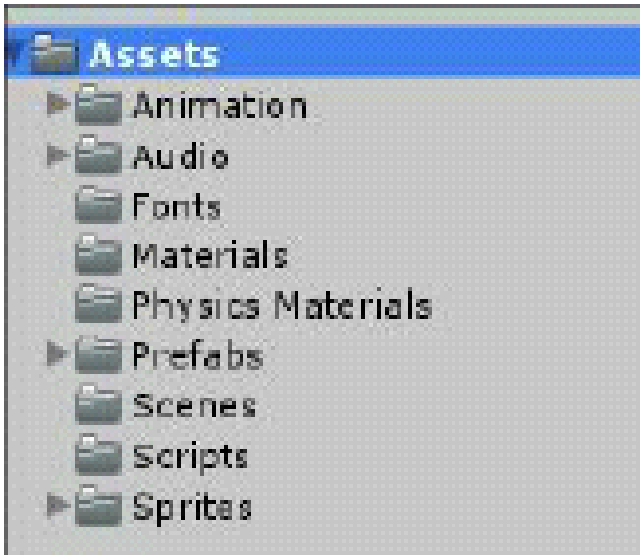
U ovom poglavlju započinjemo izradu naše igrice. Prvo ćemo kreirati projekat, a zatim i našeg glavnog junaka.

Sledi lista tema koje će biti obrađene u ovom poglavlju:

- Dizajniranje dobre strukture projekta
- Planiranje i dizajniranje ponašanja objekta
- Importovanje sprajtova
- Postavljanje korisničkih kontrola

2.1. Organizovanje strukture projekta

Najbolji način za ogranizovanje projekta koji se pravi je grupisanje objekata po njihovom tipu u korenu foldera *Assets* na sledeći način:

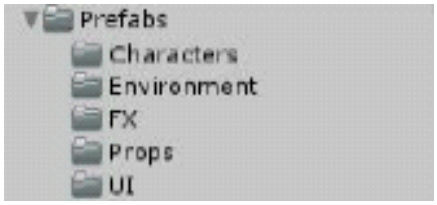


Možemo napraviti sabfoldere u zavisnosti od korišćenja.

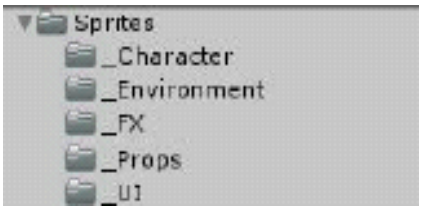
- Odvojimo klipove animacija od kontrolera za naše modele:

- Grupisanje audio snimaka u skladu sa korišćenjem u igrici:

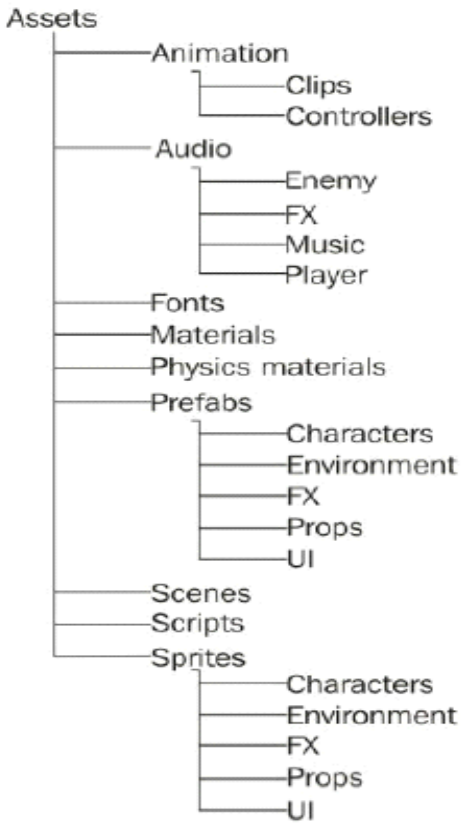
- Grupisanje predefinisanih komponenti po slojevima:



- Sprajtove smeštamo takođe po sličnom principu u stablo projekta u zavisnosti od toga kako ćemo ih koristiti:

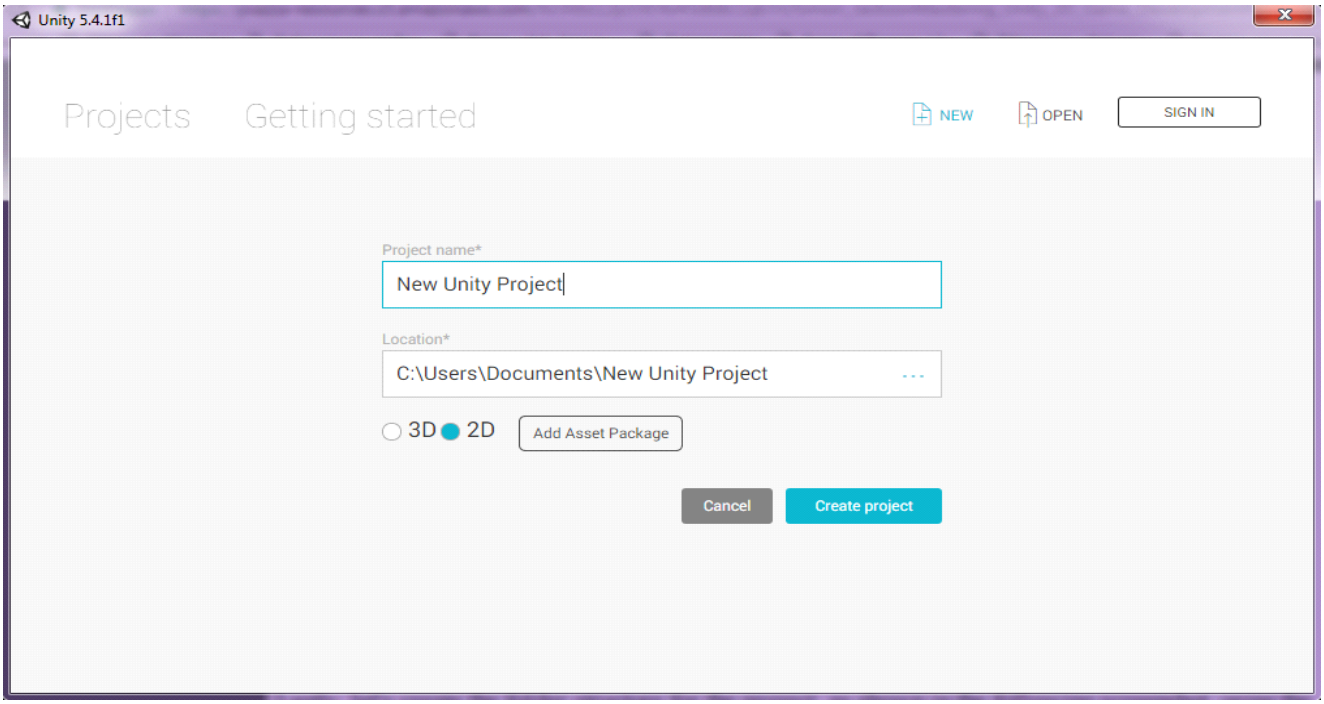


Praćenje prethodnih uputstava doprinosi boljoj efikasnosti, uređenosti i preglednosti našeg projekta. Struktura foldera *Assets* bi na kraju trebalo da izgleda ovako:



2.2. Kreiranje projekta

Nakon startovanja Unity programa prikazuje se prozor za kreiranje novog projekta, koji će biti 2D.



2.3. Planiranje i dizajniranje ponašanja objekta

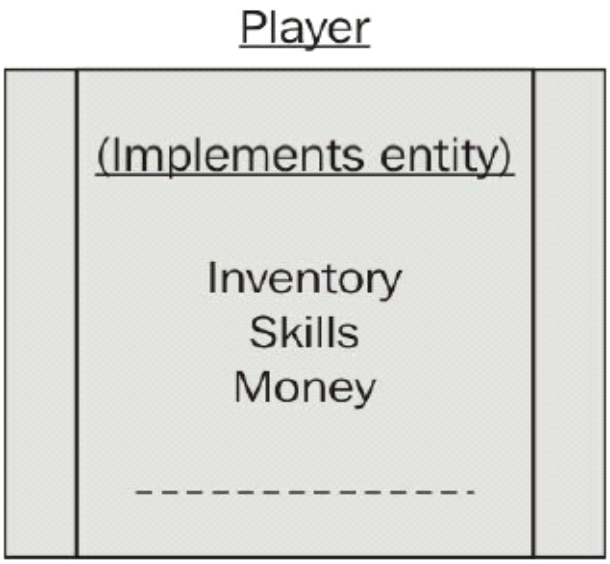
Skoro svaki entitet u našoj igrici će imati neka osnovna ponašanja, pa ćemo početi sa kreiranjem objekta *Entity* kako bi definisali neophodne atribute koje će svi naši entiteti posedovati:

Kako će postojati samo jedan tip entiteta nema potrebe za kreiranjem interfejsa, pošto će svi objekti koristiti ovu definiciju.

Klasa Entity nasleđuje specifičnu klasu ***ScriptableObject,*** koja nam omogućava da sačuvamo podatke unutar klase koja će ih koristiti za .asset fajlove u našem projektu. Vise detalja o ovoj klasi biće u nekom od sledećih poglavlja.

```
using UnityEngine;
public class Entity : ScriptableObject
{
  public string Name;
  public int Age;
  string Faction;
  public string Occupation;
  public int Level = 1;
  public int Health = 2;
  public int Strength = 1;
  public int Magic = 0;
  public int Defense = 0;
  public int Speed = 1;
  public int Damage = 1;
  public int Armor = 0;
  public int NoOfAttacks = 1;
  public string Weapon;
  public Vector2 Position;
}
```

Zatim kreiramo igrača koji će sada imati samo atribute koji su karakteristični za njega i koji ga razlikuju od ostalih objekata u igrici, što će biti mnogo lakše jer će sve ostale atribute koji su mu zajednički naslediti iz klase *Entity*.



U igrici su zastupljena sledeća ponašanja:

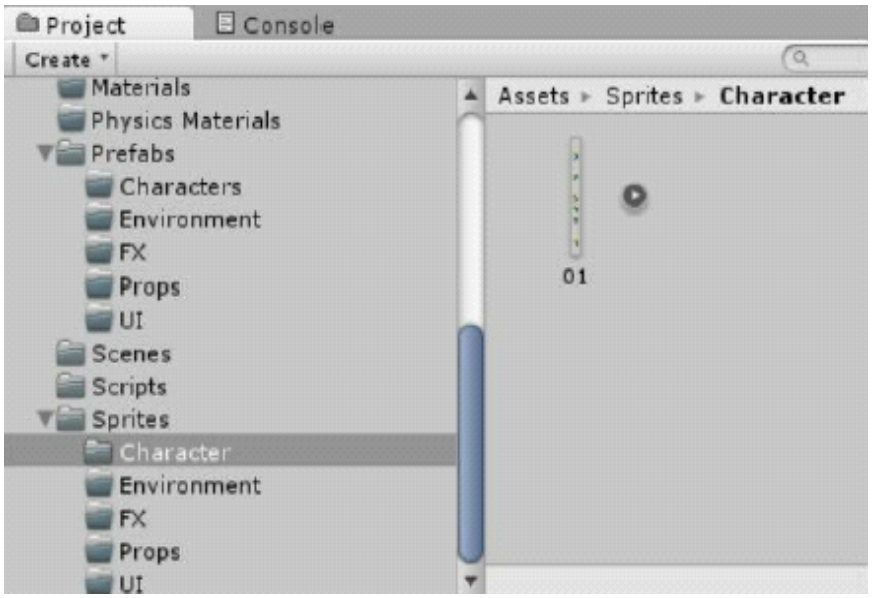
- Napadanje drugog entiteta
- Zadobijanje povreda od drugog entiteta
- Sakupljanje novca ili predmeta
- Teleportovanje u drugu zemlju

Dve metode možemo dodati sada entitet klasi, ali će o tome biti više reči u odeljku vezanom za borbu i interakciju:

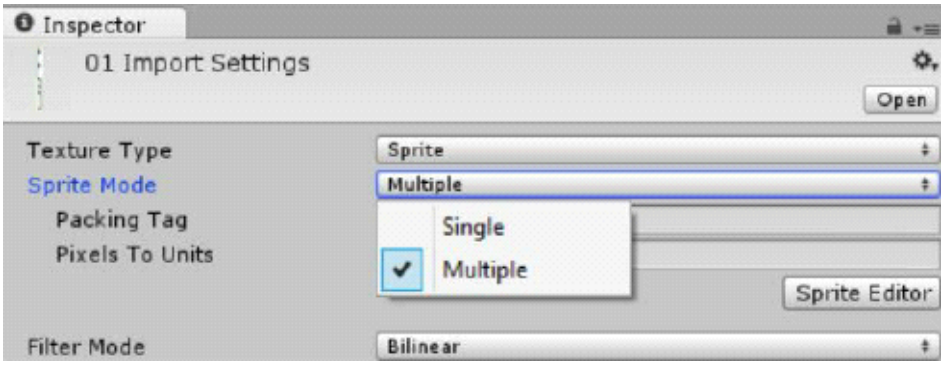
- *TakeDamage*
- *Attack*

2.4. Importovanje glavnog junaka

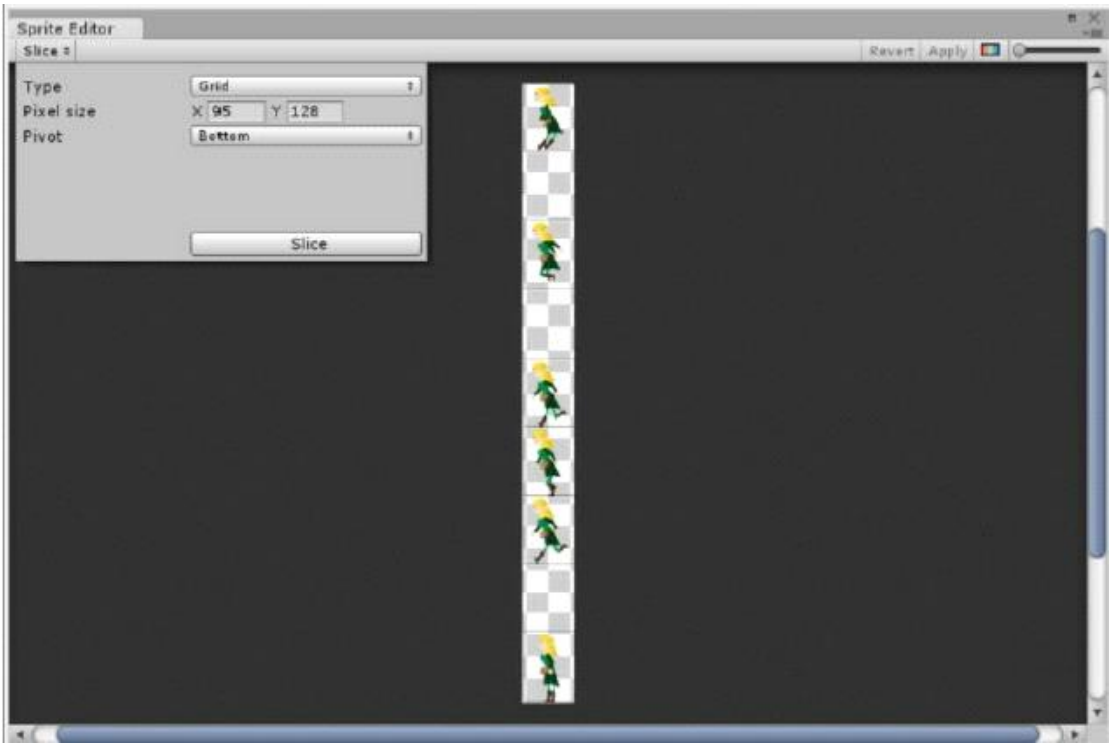
Selektujemo sliku pod nazivom *01.png* iz foldera *01_characters*, koji se nalazi u folderu *FANTASY_PACK*, i prevučemo u *Characters*, folder koji se nalazi unutar *Assets\Sprites* našeg Unity projekta, kao što je pokazano na sledećoj slici:



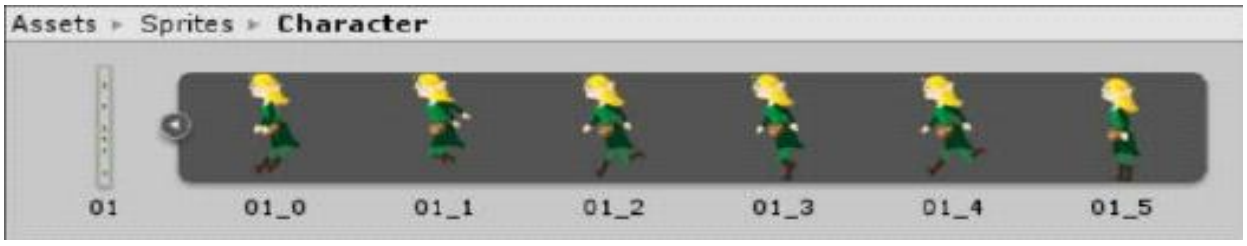
Zatim, selektujemo sliku koju smo prebacili, i u Inspector-u izvršimo potrebna podešavanja. Za ***Texture Type*** odaberemo *Sprite*, a ***Sprite Mode*** postavimo na *Multiple*.



Klikom na dugme ***Sprite Editor*** otvara se sledeći prozor:

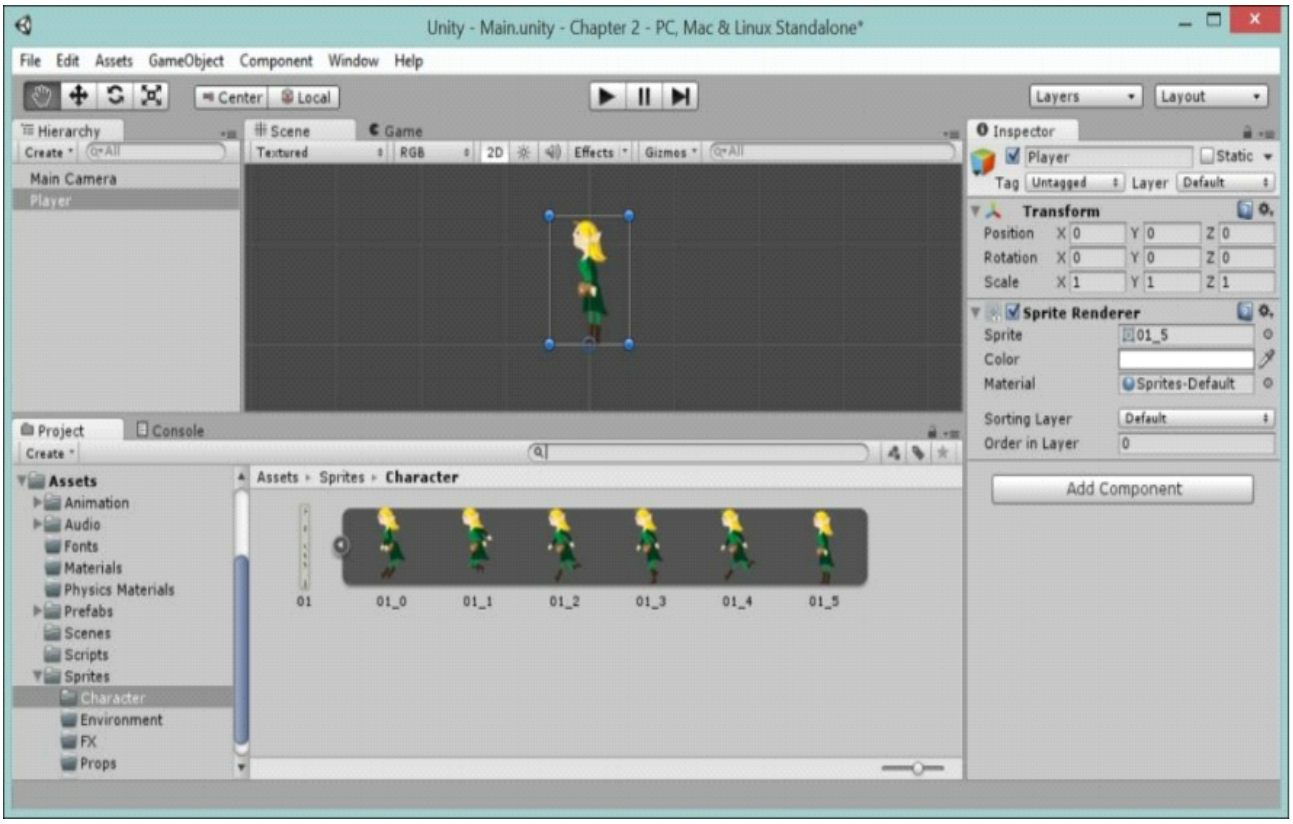


Da bi podelili sprajtove, polje ***Type*** podesimo na *Grid*, nakon čega dobijamo mogućnost da unesemo dimenzije ćelije, za ***X*** unosimo 95, a za ***Y*** 128. Polje ***Pivot*** ostavljamo podešeno na *Bottom*. Potom kliknemo na dugme ***Slice***, pa na ***Apply*** u gonjem desnom uglu Sprite Editor-a. Sada klikom na strelicu pored slike u folderu *Character* vidimo pojedinačne sprajtove:



Sada je potrebno našeg heroja prebaciti na scenu, prateći sledeće korake:

- dodajemo sprite render komponentu (**Add Component** |Kreiramo prazan objekat u prozoru Hierarchy (**Game Object |Create empty**) i imenujemo ga *Player*
- U Inspector prozoru **Rendering**|**Sprite Render**)
- Prevučemo 01_5 (sprajt u stanju mirovanja) sprajt na Sprite Render komponentu



U sledećem poglavlju ćemo se bolje i detaljnije upoznati sa Sprajtovima i nastavićemo sa nadogradnjom naše igrice, konkretno videćemo kako kontrolisati heroja pomoću strelica, levo i desno.

Zadaci za vežbu

1. Kreirajte sami nov projekat sa proizvoljnim nazivom, i formirajte pravilnu strukturu foldera.
2. Importujte sliku (proizvoljno), podelite na sprajtove i prebacite jedan sprite na scenu.

Kratak pregled

Mi smo dosta toga naučili u ovom poglavlju, ali se nećemo zaustaviti na ovome. U sledećim poglavljima, nastavljamo nadograđivanje naše igrice. Do sada smo govorili o sledećim temama:

- Osnove projektovanja igre i njene strukture
- Pregled svih glavnih komponenti (Sprite i Sprite Render)
- Importovanje novih sprite-ova
- Dobijanje pojedinačnih sprite-ova iz Spritesheets-a
- Dodavanje sprite-ova našoj igri

Poglavlje 3.Sprite (Sprajt)

U prethodnom poglavlju smo se upoznali sa osnovnim pojmovima framework-a Unity. Videli smo:

- kako se kreiraju folderi
- kako možemo kreirati C# skriptu
- kako kreiramo objekat i kako ga povezujemo sa C# skriptom
- kako dodajemo komponente objektu
- kako organizujemo naš projekat
- kako možemo u naš projekat importovati **sliku**.

U ovom folderu ćemo govoriti zapravo o tim **slikama**.

Cilj: Naš zadatak u ovom poglavlju je da kreiramo našeg glavnog junaka i da mu dodelimo određeno ponašanje – kretanje pomoću strelica levo i desno. Da bismo to uspeli upoznaćemo se sa Sprajtovima i videćemo šta su Sprajtovi, kako se kreiraju tj importuju, šta sve možemo raditi sa njima, koje vrste postoje i šta sve možemo da napravimo pomoću njih. Pa da krenemo.

3.1. Uvod – Šta je Sprajt

Sprajtovi predstavljaju bitmap sliku koju koristimo u razvoju naše igrice.

Sprajt možemo koristiti da predstavimo:

- bilo koju figuru u igrici
    - našeg glavnog junaka
    - njegovog neprijatelja
    - oružje
    - prepreke
- pozadinu (čak i delove pozadine npr oblaci, drveće, priroda) itd.

Sada se postavlja pitanje koji format fotografija koristiti i koje slike su najpogodnije za naš projekat?

Pravilo je da se teži da slike budu u .png formatu i da to budu transparentne slike (slike koje nemaju pozadinu). Sve je stvar estetike i izbora, ali mi ćemo u našoj igrici poštovati ova dva pravila.

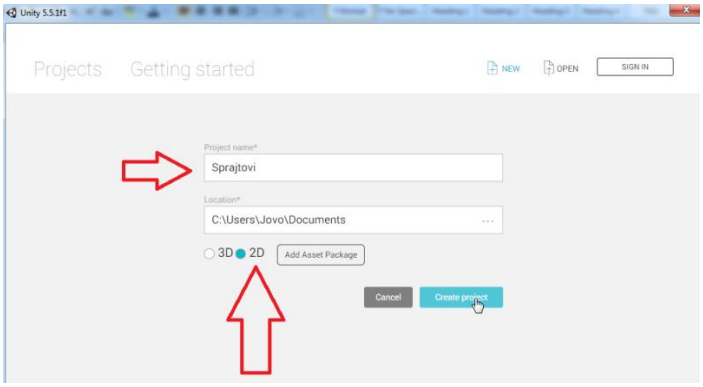
3.2. Vrste Sprajtova

Kada manipulišemo sa Sprajtovima, Unity nam pruža veliki spektar mogućnosti. Kada želimo da ubacimo Sprajt u naš projekat imamo 3 vrste Sprajtova, samim tim i 3 opcije:

1. možemo kreirati Default Sprite (tada nije potrebno importovati Sprite)
2. možemo samostalno u nekom programu (Paint, PhotoShop, itd) da kreiramo sliku tj naš Sprajt i zatim da ga importujemo (ubacimo) u naš projekat
3. možemo downloadovati gotov Sprajt sa Interneta i u praksi je najčešće korišćena ova metoda.

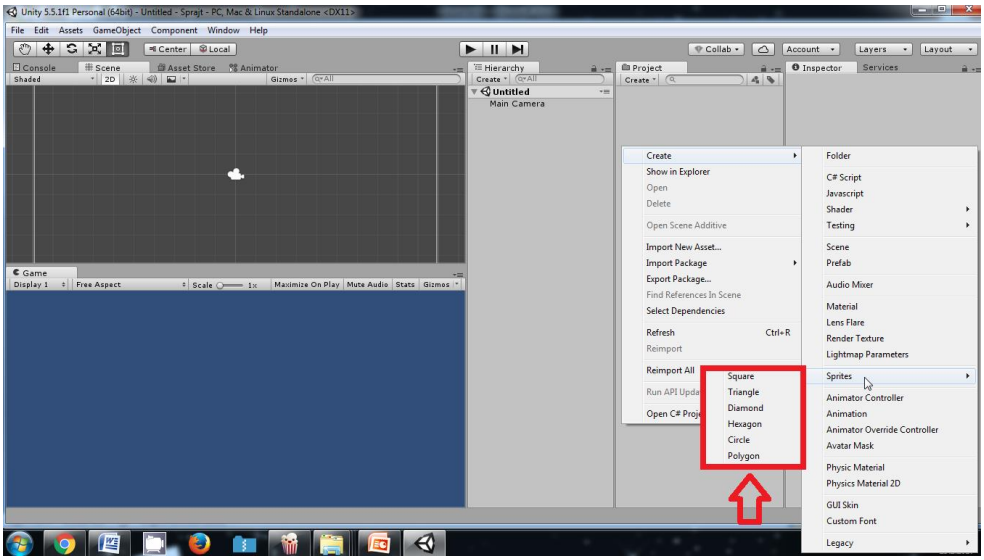
3.3. Manipulisanje sa Sprajtovima

Sada kada smo teoretski videli koje vrste Sprajtova postoje, da bismo ih koristili u našem projektu, moramo ih kreirati, importovati, pa da vidimo kako to izgleda. Pokrenućemo Unity. Da se podsetimo, radimo **2D** projekat, pa dodelite naziv projektu koji Vama odgovara npr „Sprajtovi“ iklikom na **Create Project** možemo da krenemo.

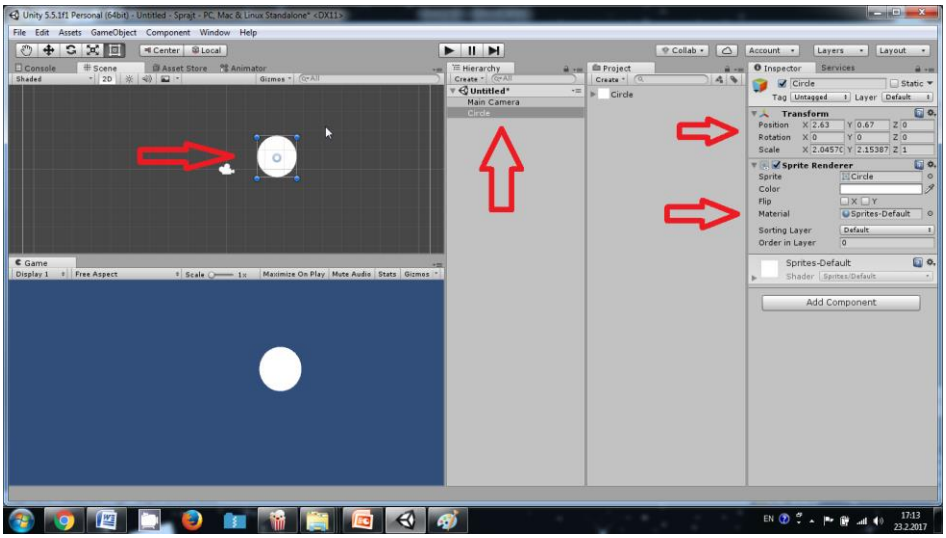


Ako želimo da kreiramo Default Sprite koji Unity već poseduje postupak je sledeći:

- U delu Project idemo desni klik -> Create -> Sprites, gde vidimo ponudu Default Sprajtova
    a. Square
    b. Triange
    c. Diamond
    d. Hexagon
    e. Circle
    f. Polygon



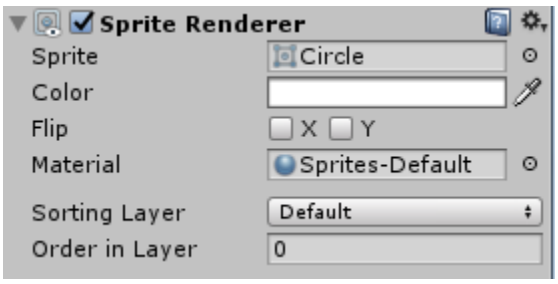
- Izabraćemo neki od ponuđenih, recimo Circle i klikom na njega možemo videti da će se pojaviti u delu Project, ali ne i na sceni i u delu Hierarchy. Sada da bi se pojavio na sceni jednostavnim prevlačenjem ćemo ga prevući i on će se pojaviti i u delu Hierarchy, tako da je sada deo našeg projekta. Šta zapažate na sledećoj slici?
-



Pored toga što je naš Sprajt dodat na scenu i u delu Hierarcy, možemo primetiti da u delu za Inspector su dodate 2 komponente: Transform i Sprite Renderer. Kao što smo napomenuli u prethodnom poglavlju kada god kreiramo objekat njemu će Unity automatski dodeliti komponentu Transform pomoću koje možemo da:

- Position – menjamo poziciju objekta
- Rotation – da rotiramo naš objekat
- Scale – da skaliramo naš objekat tj da mu menjamo dimenzije

Sprite Renderer je kako sama reč kaže prikazivač za Sprajt, to je komponenta koja je neophodna za Sprajtove:



U delu Sprite se nalazi referenca na naš objekat tj prostije rečeno nalazi se slika koja je dodeljena nekom objektu, a ta slika je Sprajt.

Color deo služi ako želimo da promenimo boju objektu.

Flip deo se koristi ako želimo da okrenemo npr našeg junaka ako gleda u levo, da gleda u desno, ili da ga okrenemo naopako. U prvom slučaju bismo štiklirali X, a u drugom slučaju Y.

Order in Layer se koristi kada imamo više slika tj Sprajtova, pri čemu preklapaju jedan drugi, i da bi se to izbeglo dodeljuju se prioriteti, tj onaj Sprajt koji ima veći Order in Layer broj će biti preko drugog. Ako imamo dva Sprajta npr Sprite1 i Sprite2 sa Order in Layer redom 0 i 1, tada će drugi Sprajt da bude preko prvog zato što ima veći prioritet.

Pored ove dve komponente možemo sami dodati koliko želimo komponenti klikom na

u Inspector delu. Neke od njih su npr BoxColider2D, Rigibody2D, bilo koja C# skripta itd.

Ostale elemente (Sorting Layer, Sprites-Default) ćemo detaljnije obraditi u nekom od sledećih poglavlja.

Videli smo kako funkcionišu Default Sprajtovi koje nam nudi Unity okruženje, a sada ćemo da vidimo koja je razlika u odnosu na Sprajtove koji sami kreiramo i onih koje downloadujemo sa Interneta. S obzirom da je jedina razlika između ove dve vrste Sprajtova (onih koje smo kreirali sami i onih koje smo downloadovali sa Interneta) zapravo ta da smo jedne kreirali mi a druge nismo, pa samim ti pošto to nije značajno za

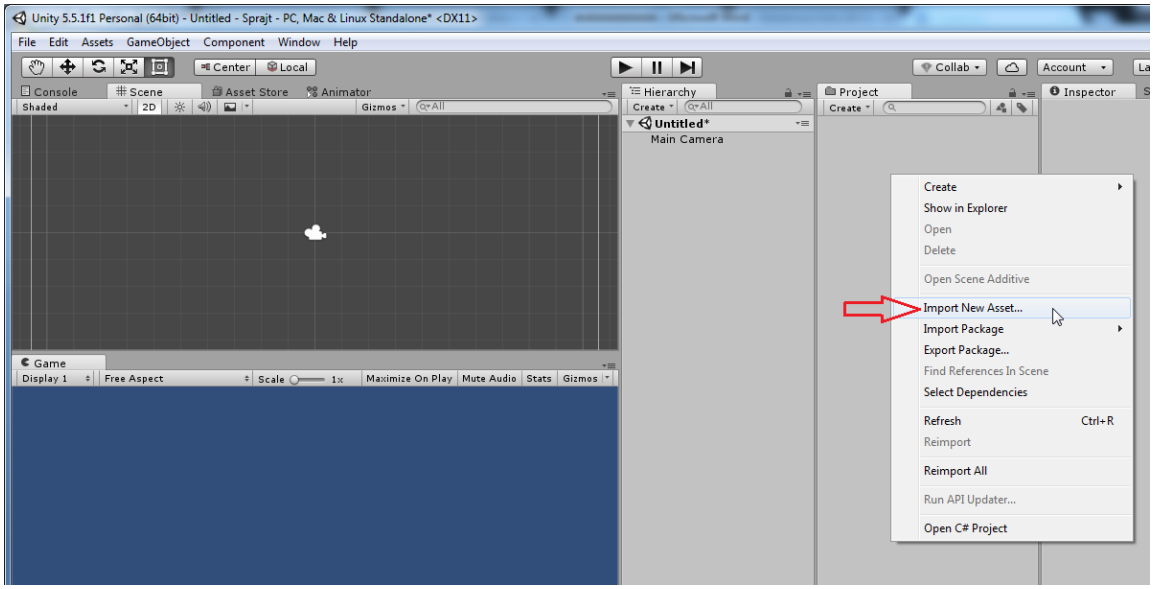
Unity nećemo pojedinačno objasniti kako funkcionišu, nego ćemo videti samo kako to ide sa Sprajtovima sa Interneta, a između ostalog zato što sve isto funkcioniše.

A ključna razlika između ove dve vrste Sprajta (onih koje smo kreirali sami i onih koje smo downloadovali sa Interneta) sa Default Unity Sprajtovima je ta da one koje smo kreirali sami i one koje smo downloadovali sa Interneta moramo importovati u naš projekat, a kako ćemo to uraditi videćemo u nastavku.

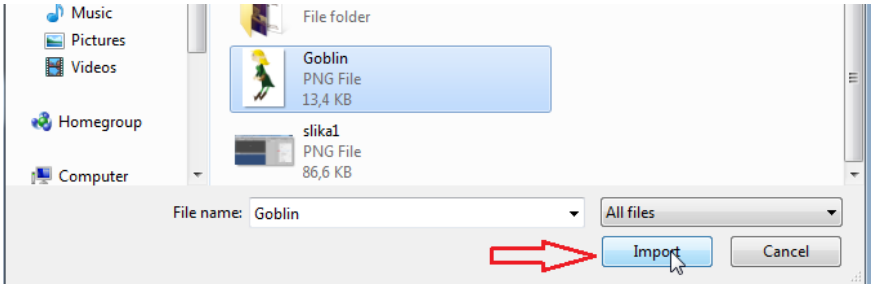
3.4. Importovanje Sprajtova

Kada želimo da importujemo Sprajt u naš projekat, za to postoje 2 načina:

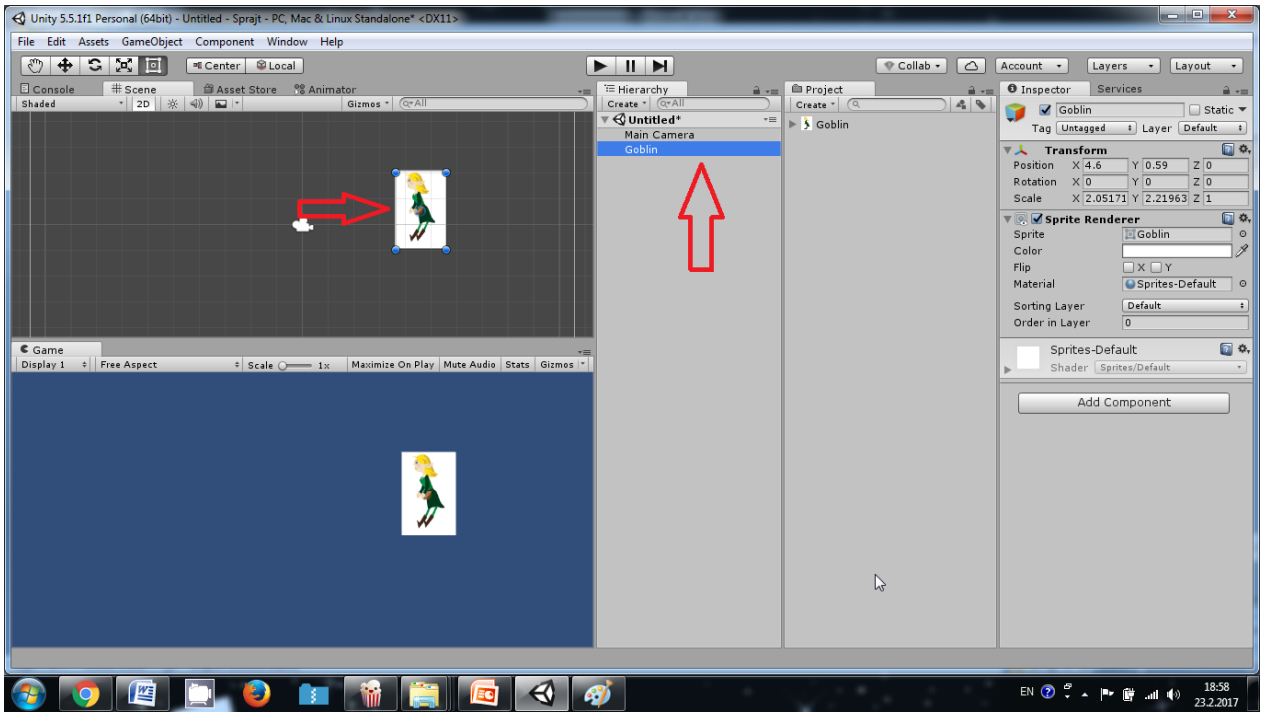
1. Prostim prevlačenjem Sprajta (slike) u delu Project
2. Desni klik u delu Project ->Import New Asset…



Pronađemo našu sliku ili Sprajt (rekli smo da će to biti .png format). Klikom na

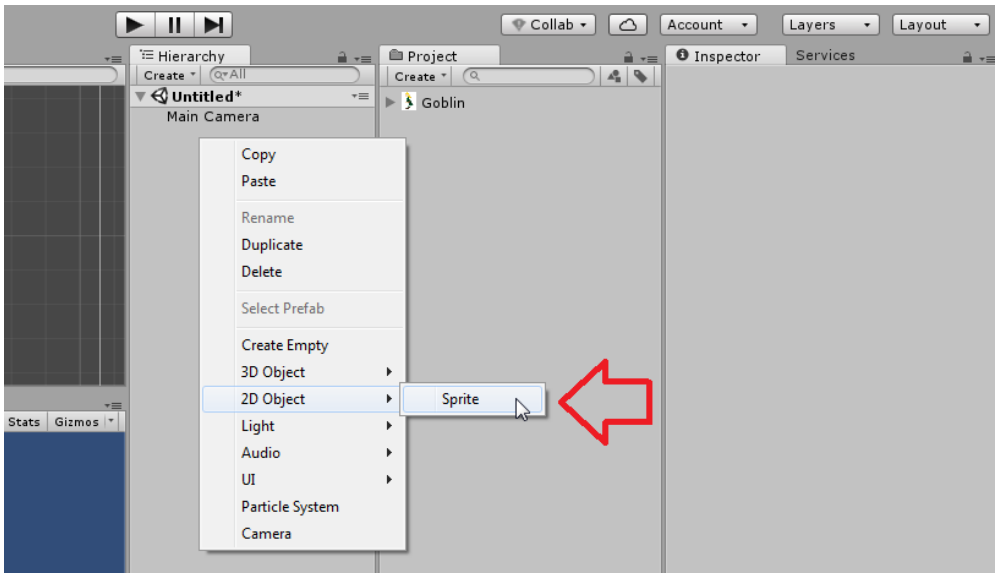


Import ona će biti u našem projektu.



Ali kao što vidimo, ona ne nije pojavila na sceni, niti u delu Hierarchy. To je zato što smo mi importovali Sprajt, importovali smo našu sliku, ali nismo kreirali objekat niti mu dodelili taj izgled. Isto kao što smo radili kod Default Sprajtova, možemo i ovde da prevučemo našu sliku iz dela Project na scenu i Unity je tako sam kreirao objekat i dodelio mu ovaj izgled. Drugi način bi bio sledeći:

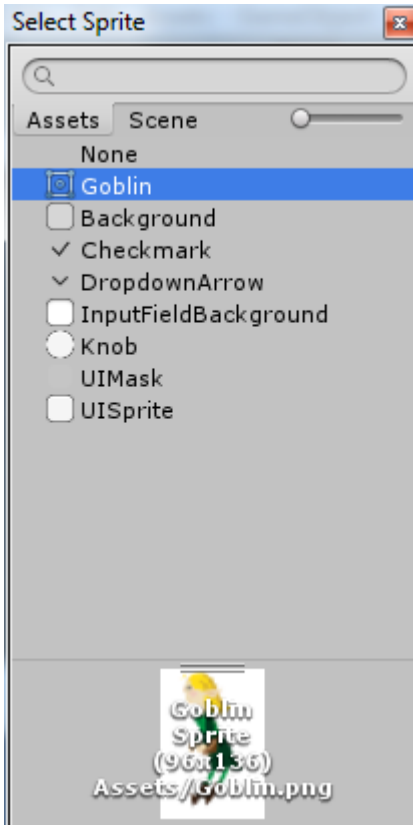
- Nakon importovanja u delu Hierarcy, desni klik, 2D Object -> Sprite



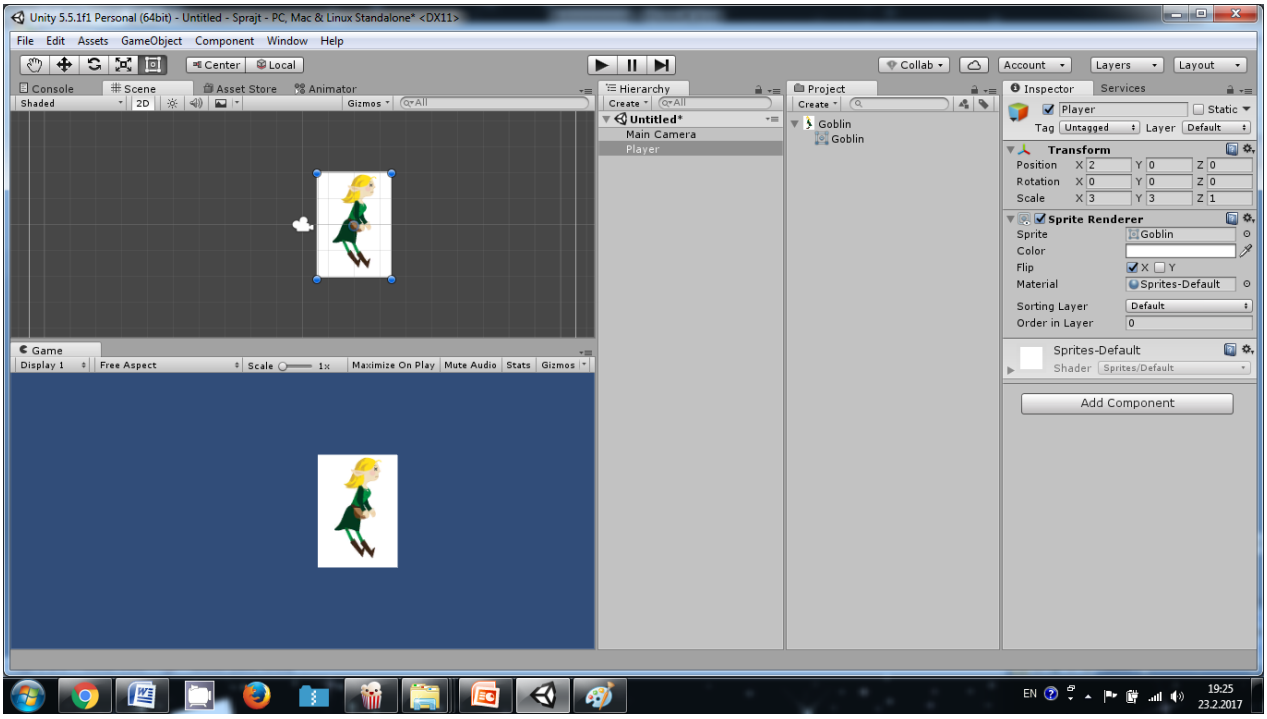
Ovim smo kreirali Objekat koji je tipa Sprite, a Goblin Sprajt tj slika koju smo importovali predstavlja samo izgled koji treba da dodelimo našem objektu. Idemo u deo Inspector i izvršićemo sledeće promene:

- Promenićemo ime iz New Sprite u Player
- U Transform komponenti nećemo ništa menjati (trenutno nema svrhu zato što naš igrač nije još na sceni)
- U Sprite Renderer komponenti, u delu Sprite, kliknućemo na mali krug pored dela None(Sprite)

pri čemu će se otovriti nov prozor kao na sledećoj slici



Izabraćemo i selektovaćemo našu sliku Goblin. Sada možemo videti da je naš junak na sceni. Proizvoljno mu možete promeniti neke elemente, npr u delu Transform dimenzije, nakon svih promena imamo izgled kao na sledećoj slici.



Ukoliko želimo da obrišemo objekat tipa Sprite ili sliku Sprite, jednostavno kliknemo na onaj koji želemo da obrišemo i na taster Delete će biti uklonjen.

Da sumiramo:

Treba razlikovati sliku koja je Sprajt i objekat 2D koji je tipa Sprite. Mi kada importujemo neku sliku ona je Sprajt, i to je zapravo izgled koji treba da dodelimo nekom objektu. U našem slučaju objekat je tipa Sprite, ali ne mora da bude. Pored Sprite-a to može biti i klasičan objekat koji se kreira u delu Hierarchy desni klik pa izabrati CreateEmpty.

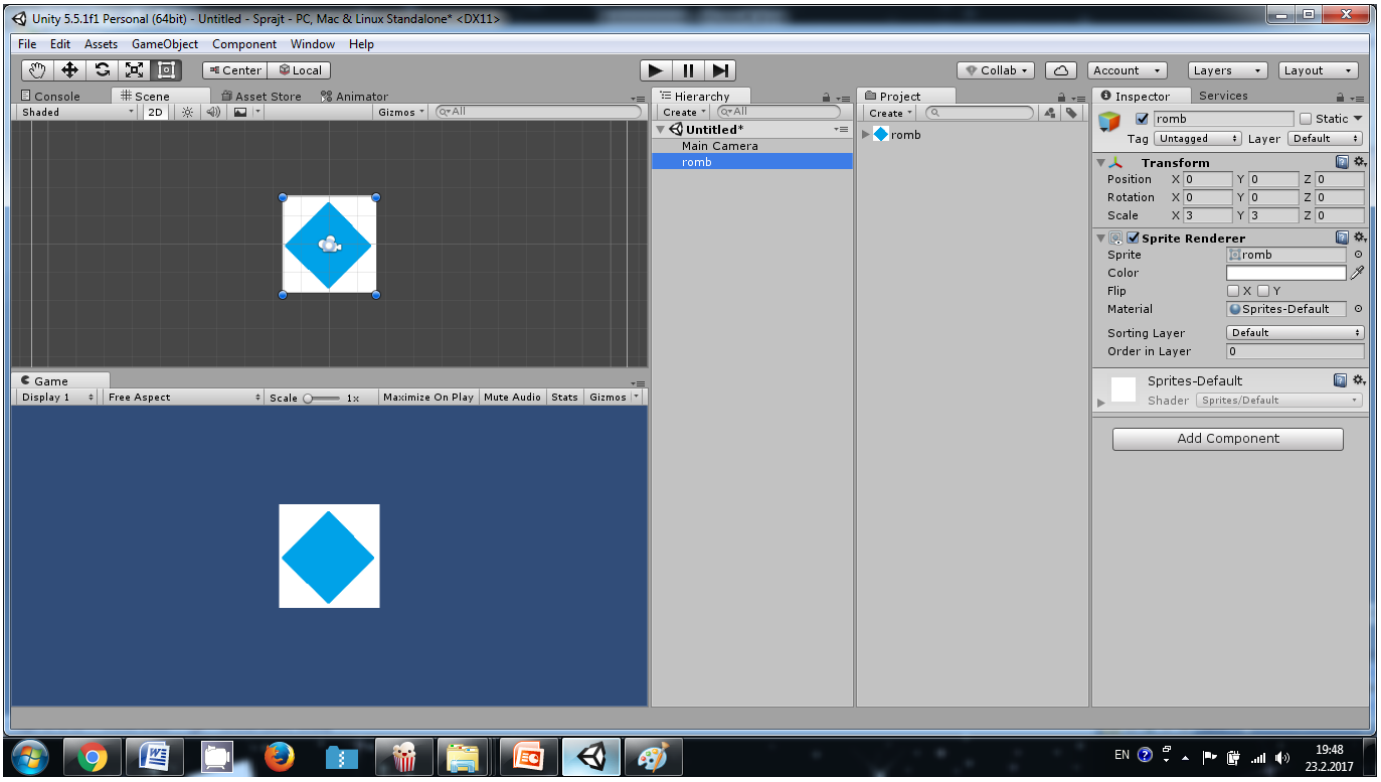
3.5. C# Skripta kao komponenta Sprajta

U ovom delu ćemo videti kako pomoću skripte možemo da dodelimo našem objektu tipa Sprite ponašanje. Utvrdićemo dosadašnje gradivo i odradićemo primer u kome ćemo omogućiti našem objektu da se rotira oko svoje ose.

Prvi korak jeste da kreiramo objekat i dodelimo mu izgled tj sliku (Sprajt) i to :

- Desni klik u delu Project, zatim Import New Asset... Odaberemo našu sliku tj Sprajt
- Da bi se pojavio na sceni i da bismo dodelili izgled objektu prevučemo ga jednostavno na scenu
- U delu Hierarcy će se kreirati objekat koji će biti vidljiv na sceni
- Klikom na naš objekat u delu Hierarcy postaće vidljiv deo Inspector gde ćemo u komponenti Sprite Renderer u delu Sprite videti da je Unity automatski dodelio izgled našem objektu tj nismo morali None (Sprite) da menjamo u npr Romb. Nekada ćemo morati da odradimo taj korak, ako imamo više objekata i više Sprajtova, a nekada će to Unity odraditi za nas, zavisi od verzije.

Za sada naš projekat izgleda kao na sledećoj slici:



Sledeći korak je da kreiramo Skriptu u Visual Studio – u i da je dodelimo našem objektu.

Izgled skripte je sledeci:

```
using UnityEngine;
using System.Collections;
public   float promenljiva = 1f;
public   class   Skripta : MonoBehaviour {
        public   float promenljiva = 1f;
        void   Start()  {
        }
        Void   Update()  {
        transform.Rotate(0,0,promenljiva);
}
}
```

Nakon što smo kreirali skriptu, izvršili navedene promene možemo da je sačuvamo i vratimo se u Unity okruženje. Klikom na romb u delu Hierarcy i prikazivanja Inspector dela idemo na Add Component i u Search delu nađemo našu skriptu, kliknemo na nju i kao što vidimo ona je sada jedna od komponenti našeg objekta. Šta to znači. To znači da smo sada našem objektu koji je romb dodelili ponašanje koje je u skripti Skripta. Klikom na dugme Play romb će krenuti da se rotira. Hajde malo da pojasnimo šta radi linija koda **transform.Rotate(0,0,promenljiva);**

transform.Rotate(0,0,promenljiva) će omogućiti objektu kome je dodeljena ova skripta da se rotira.

Rotate ima 3 argumenta, prvi se odnosi na ugao koji zaklapa sa x osom, drugi argument se odnosi na

Ugao koji zaklapa sa y osom, i treći ugao koji zaklapa sa z osom.

Zadaci za vežbu:

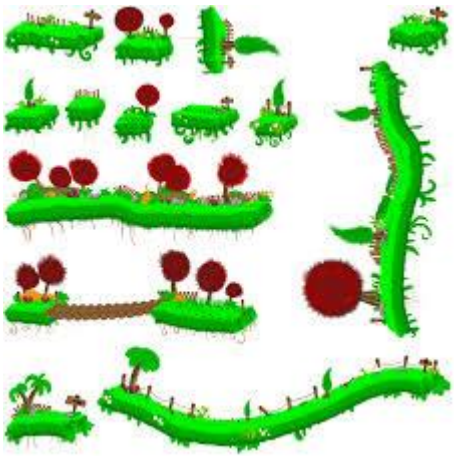
1. Probajte da u kodu promenite redosled argumenata tj da budu poređani redom 0,promenljiva,0 Ili promenljiva,0,0 da vidite šta će se dogoditi.
2. Takođe možete promeniti vrednost promenljive umesto 1f npr 5f da vidite razlike.
3. Kreirati C# skriptu koja će da omogući krugu da menja boje zeleno, crveno, crno, redom koji vama odgovara i brzinom koja je proizvoljna.
   Mala pomoć: možete importovati 3 Sprite-a, tj 3 različite slike koje će preklapati jedna drugu. Na scenu prevucite samo 1 krug.

3.6. Spritesheet

Pogodnost koju pruža Unity je da kada imamo dosta slika tj Sprajtova, ne moramo da importujemo jedan po jedan nego možemo importovati Spritesheet, tj Sprajt koji se sastoji iz više elemenata, odnosno iz više Sprajtova. Spritesheet se uglavnom koristi za animacije, ali može se koristiti npr i za kreiranje neke pozadine što ćemo videti u našoj igrici.

Sada ćemo importovati jedan Spritesheet, naznačiti neke razlike u odnosu na dosadašnje pomenute Sprajtove i kreiraćemo jednu animaciju.

Na sledećoj slici možete videti kako izgleda jedan proizvoljan Spritesheet.



Importovanje Spreitesheet-a je isto kao i klasičnog Sprajta. Znači desni klik u delu Project, zatim Import New Asset… i izaberemo naš Spritesheet. Neka to bude naš glavni junak. Da napomenemo da u poglavlju 2 nismo još konkretno počeli da nadograđujemo našu igricu, nego se i dalje upoznajemo sa Sprajtovima.

Sledeći deo numerisan sa 2.7. je posvećen nadogradnji igrice. Da se vratimo na naš Spritesheet tj na pravljenje animacije.

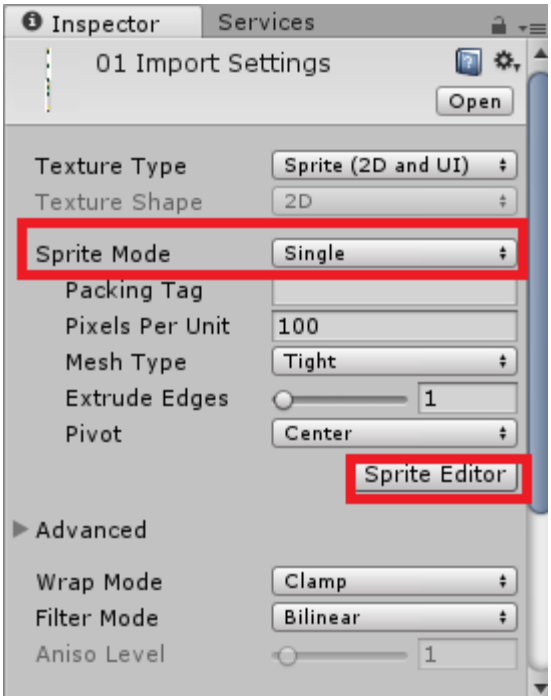
Importovanje Spreitesheet-a je isto kao i klasičnog Sprajta. Znači desni klik u delu Project, zatim Import New Asset… i izaberemo naš Spritesheet. Klikom na Import pojaviće se u delu za Project ali ne i na sceni i u delu Hierarchy. Za sada nećemo kreirati objekat i dodeliti mu izgled našeg Sprajta. Kod Spritesheet-ova je procedura malo drugačija.

Najvažniji delovi kod Spritesheet-a su: **Sprite Mode i Sprite Editor.**

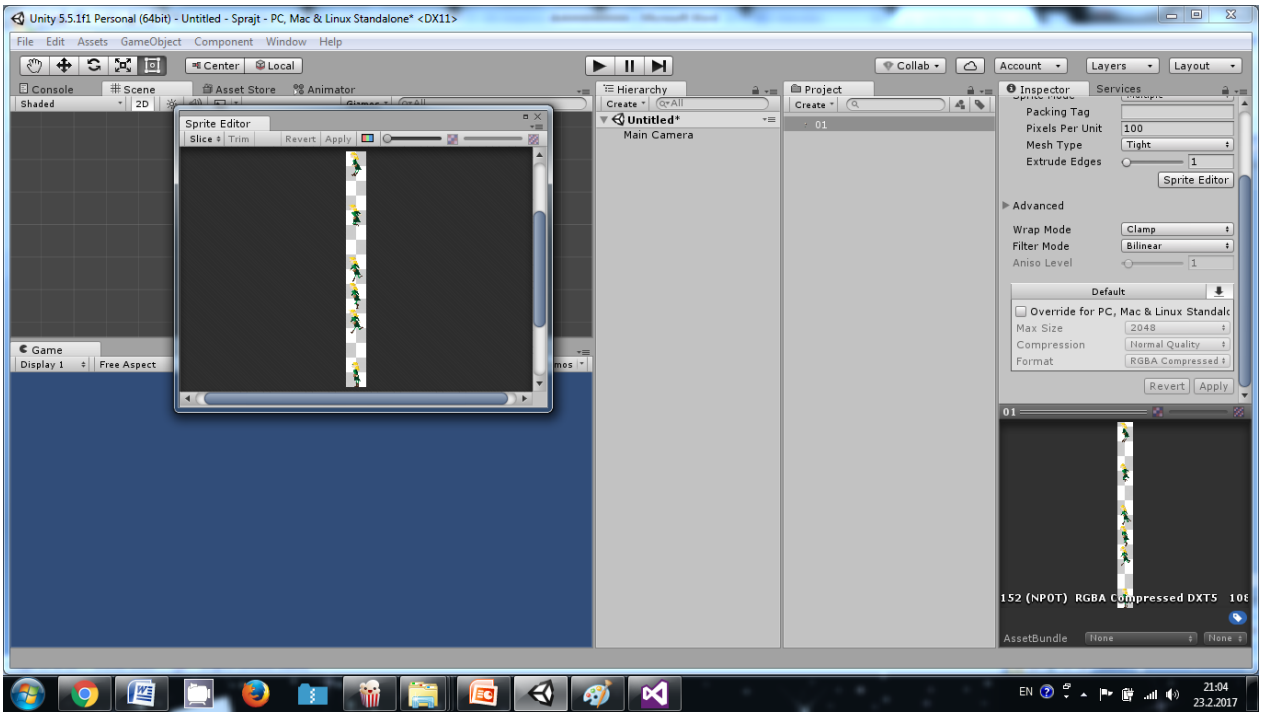
Kada radimo sa Spritesheet-om Sprite Mode mora obavezno biti **Multiple**. Tako ćemo obavestiti Unity da

Želimo da on taj Sprite prepozna kao Spritesheet kako bi mogao da izvrši određene promene na njemu. Nama zapravo treba da taj Spritesheet izdelimo na više elemenata tj više slika.

Na sledećoj slici možete videti da smo u delu Sprite Mode izabrali **Multiple**.

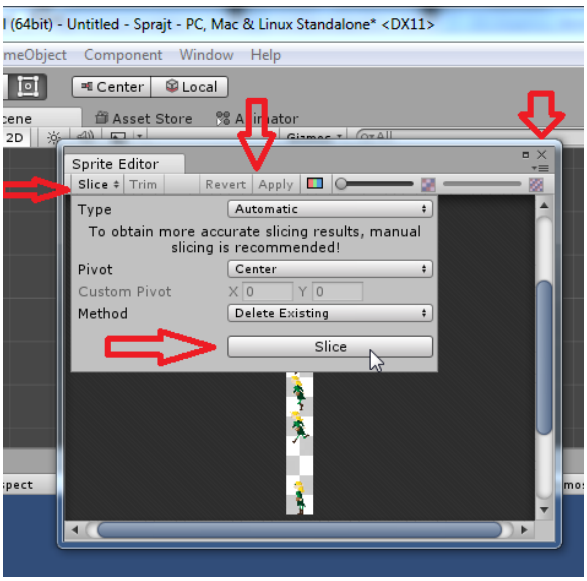


Sledeći korak je Apply kako bi Unity to registrovao, a zatim Sprite Editor nakon čega će se otvoriti prozor kao na slici.

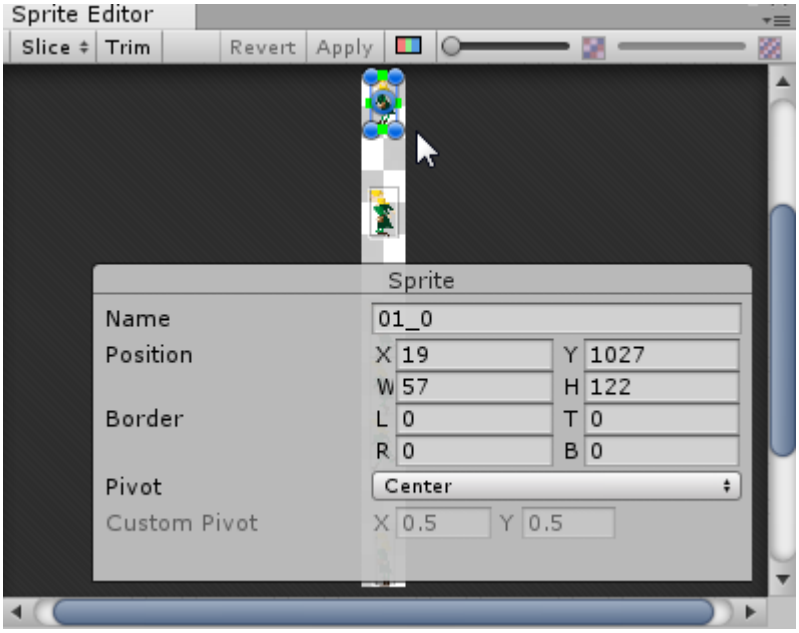


Nakon što se otvorio prozor partite sledeće korake:

- Kliknite na deo Slice
- Zatim klik na dugme Slice
- Sledeci korak je Apply

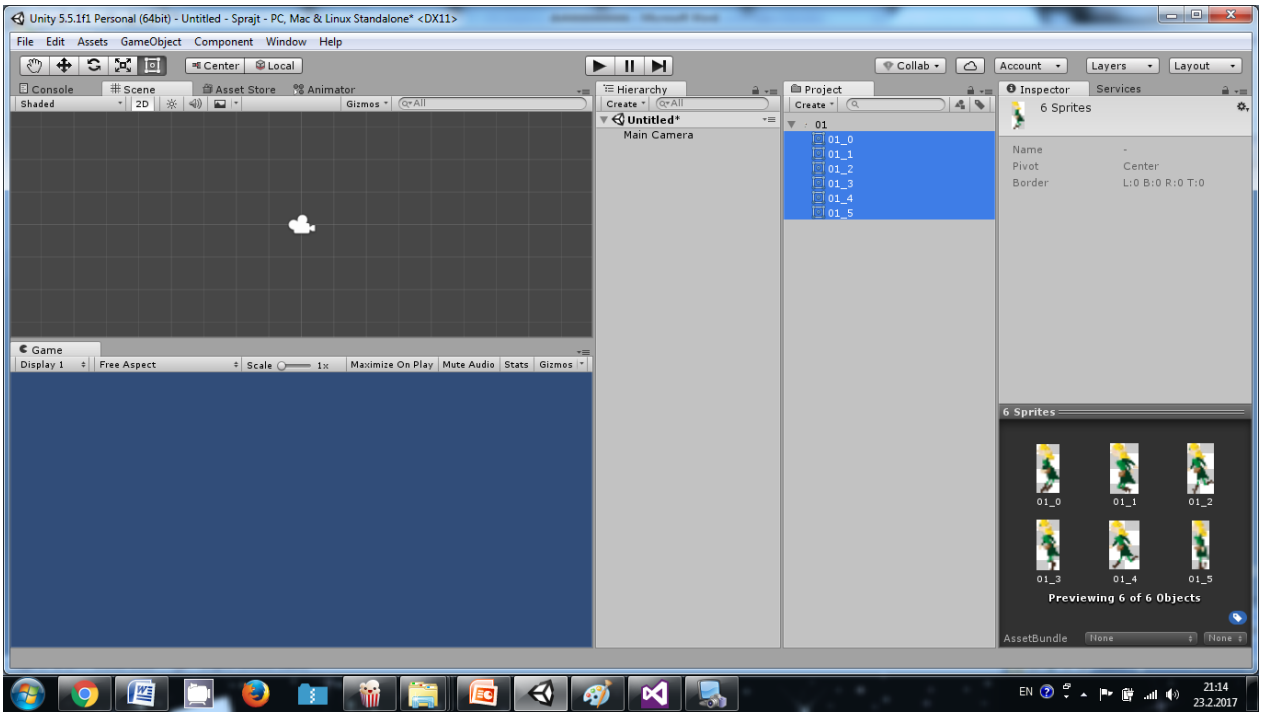


Možemo videti da je Unity izdelio Spritesheet na više manjih Sprajtova. Svaki od njih klikom na proizvoljni možemo menjati pojedinačno neke elemente kao što vidite na slici. Naglasimo da je Pivot tačka oko koje se rotira naš objekat.

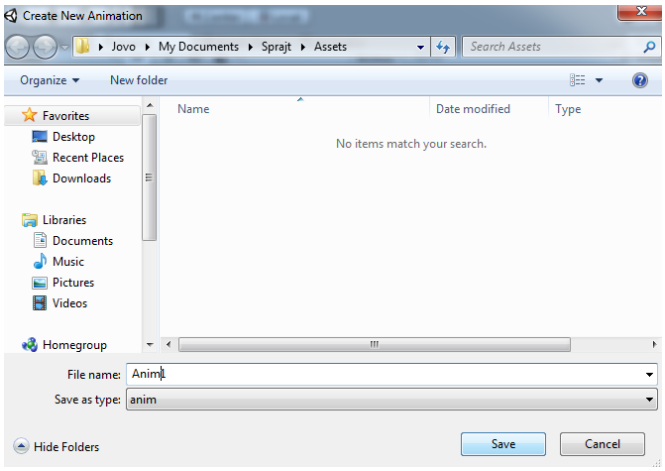


Nakon što smo završili (pretposlednji korak je bio Apply da bi se registrovale promene) klikom na x zatvorićemo prozor. Šta zapažamo sada? Nešto se promenilo sa našim Sprajtom u delu Project.

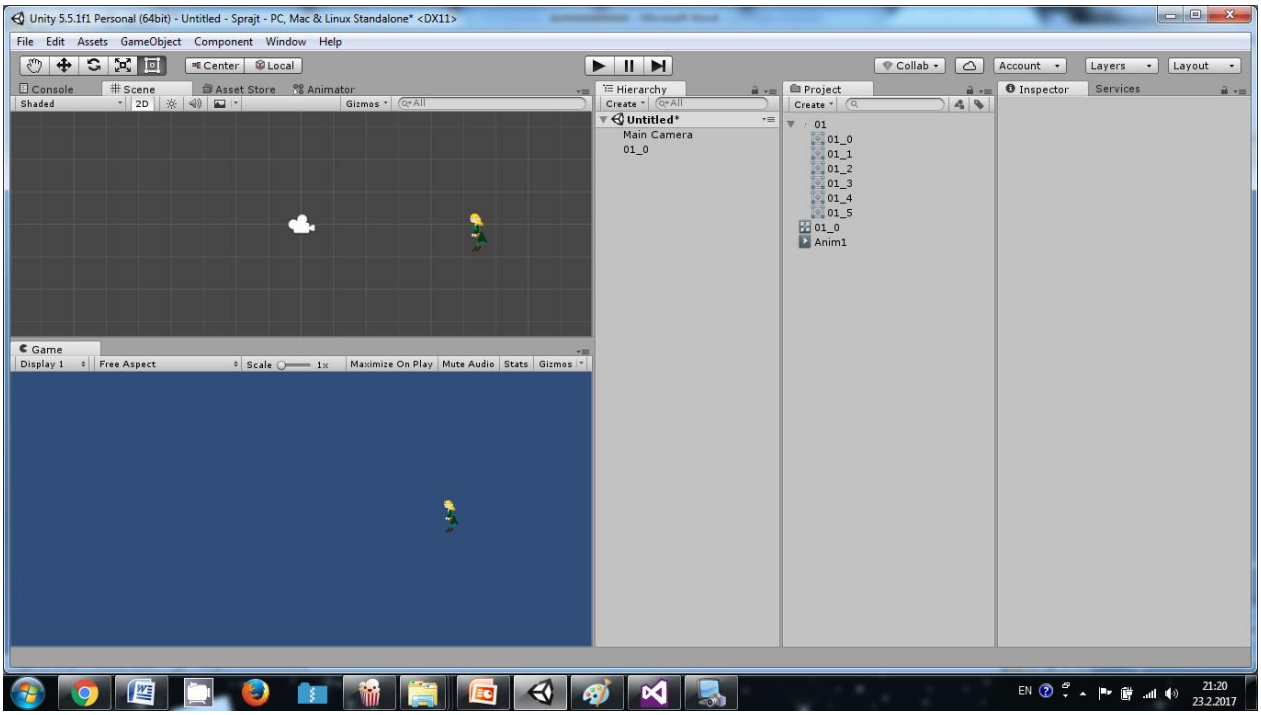
Klikom na strelicu pored imena našeg Sprajta vidimo da je on izdeljen na više manjih. Da bismo napravili animaciju pratite sledeće korake. Obeležite svih 6 Sprajtova na koje je izdeljen naš Spritesheet.



I zatim ih prevucite na scenu. Unity će automatski prepoznati da ste želeli da napravite Animaciju i na vama je samo da je sačuvate pod nekim imenom proizvoljno, recimo Animacija. Pogledajte sliku.



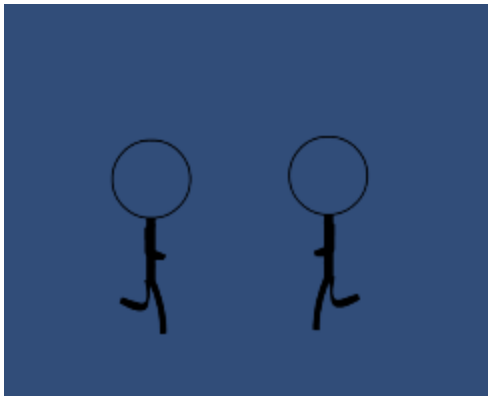
Kada smo sačuvali Animaciju, naš projekat izgleda kao na slici



I klikom na dugme Play naš karakter će krenuti da trči. Ovo je bila kratka Animacija, u sledećim poglavljima ćemo videti i upoznati se s malo složenijim aplikacijama, ovo je bio primer kako biste se upoznali i videli za šta sve možemo upotrebiti Sprite-ove i Spritesheet-ove.

Zadaci za vežbu:

Downloadujte sa internet Spritesheet nekog karaktera i napravite projekat u kome ćete importivati istog junaka 2 puta pri čemu će on trčati u susret samom sebi. Kao na slici



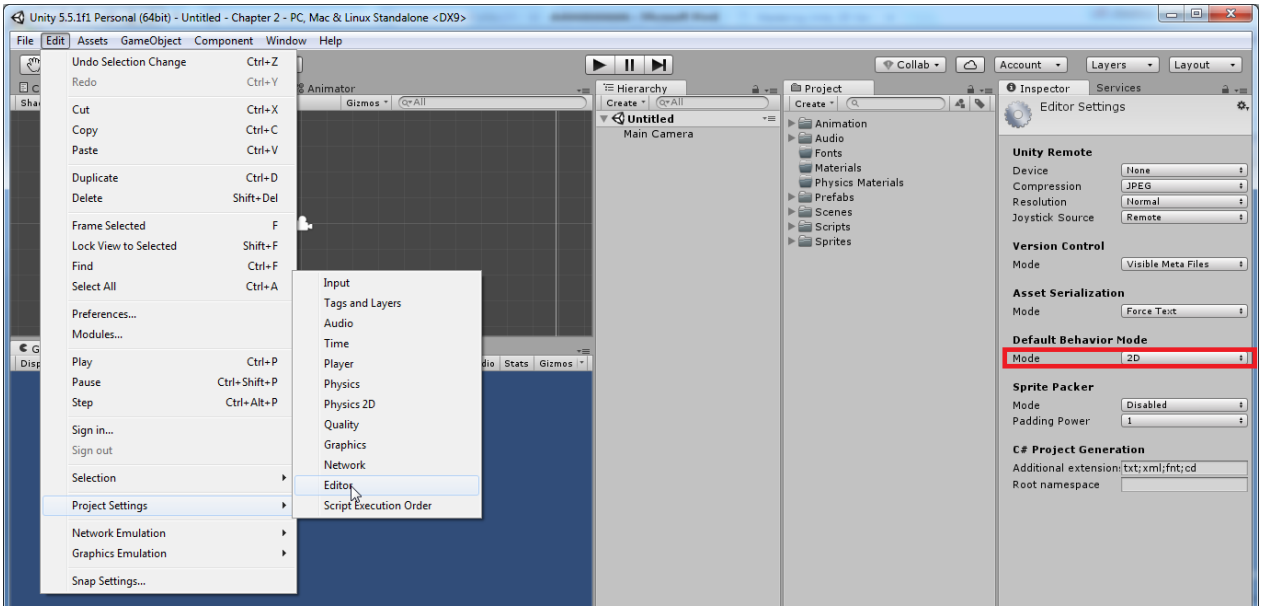
3.7. Kontrolisanje heroja

U ovom poglavlju smo se upoznali sa Sprajtovima, naučili smo kako možemo da ih importujemo, kako manipulišemo njima, šta su Spritesheet-ovi, a sada ćemo sve naučeno primeniti na našoj igrici. Nastavljamo da nadograđujemo našu igricu. Da se podsetimo – do sada smo kreirali foldere koji su struktuirani i organizovani, videli smo kako importovati našeg junaka, i sada je naš zadatak da mu omogućimo kretanje pomoću strelica, levo i desno. Pa da krenemo.

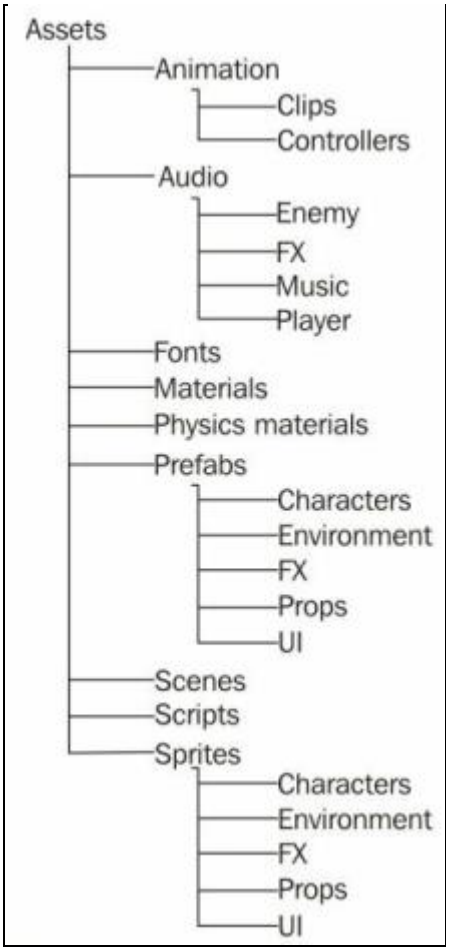
Projekat koji smo kreirali je 2D. Ukoliko ste kreirali greškom 3D projekat i želite da ga promenite u 2D ili obratno, to možete promeniti na sledeći način:

Potrebno je izvršiti promene u Editor Settings i to :

-Edit -> Project Settings ->Editor izmeniti deo Mode kao na sledećoj slici



Strukturu foldera možete videti u prethodnoj slici u delu Project (Animation, Audio,...,Scripts,Sprites) ili na sledećoj slici.



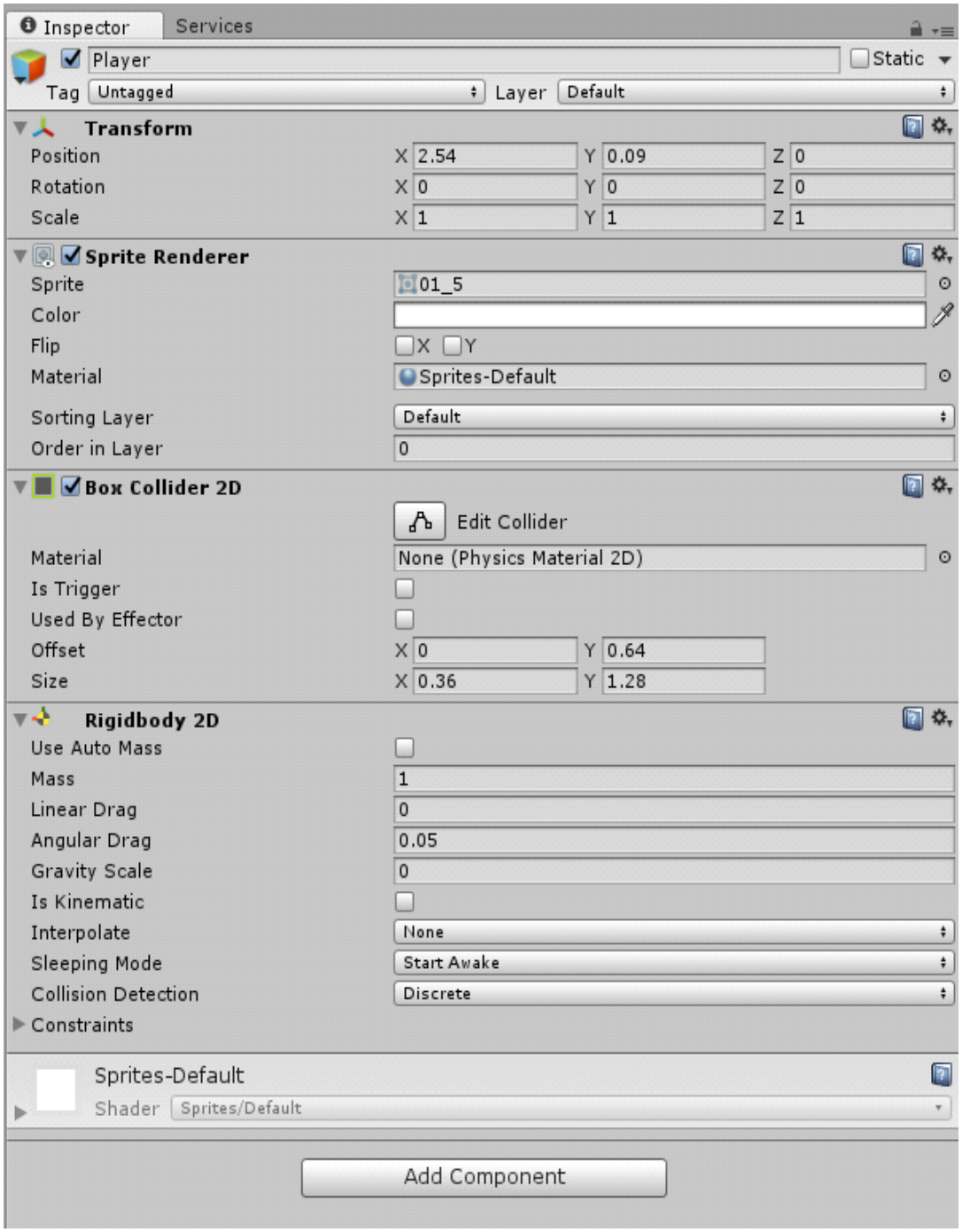
Sada kada smo se podsetili, možemo da nastavimo na razvoju naše igrice. Importovali smo glavnog junaka i sada ćemo mu dodeliti skriptu Charactermovement koja će mu omogućiti pomoću strelica levo i desno da se kreće.

Slika koju smo importivali u folderu Sprites / Character je naziva 01.png i ona je Spritesheet.

Da bi sprajtu dali mogućnost fizičkih pokreta moramo da uključimounapred definisane komponente:

- **Rigidbody 2D** koji nam omogućava da nas heroj ne može da prolazi kroz druge objekte u svetu.
- **BoxCollider 2D** koji igraču stavlja do znanja da je udario u drugi objekat, tj. da je došlo do interakcije.

Postaviti za Rigidbody *Gravity Scale* na 0 (da ne bi koristili gravitaciju) i čekirati opciju *Is Kinematic.* Za BoxCollider *Size X* na 0,36.


Inspector
Services
Player
Static
Tag
Untagged
Layer
Default
Transform
Position
X 2.54
Y 0.09
Z 0
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Sprite Renderer
Sprite
01_5
Color
Flip
X
Y
Material
Sprites-Default
Sorting Layer
Default
Order in Layer
0
Box Collider 2D
Edit Collider
Material
None (Physics Material 2D)
Is Trigger
Used By Effector
Offset
X 0
Y 0.64
Size
X 0.36
Y 1.28
Rigidbody 2D
Use Auto Mass
Mass
1
Linear Drag
0
Angular Drag
0.05
Gravity Scale
0
Is Kinematic
Interpolate
None
Sleeping Mode
Start Awake
Collision Detection
Discrete
Constraints
Sprites-Default
Shader
Sprites/Default
Add Component

Kako bi se naš heroj mogao kretati levo ili desno, moramo napisati C# skript, pod nazivom *CharacterMovement.cs, i* sačuvati ga u folderu *Assets**Scripts*. Skripta ima sledeći sadržaj:

```
using UnityEngine;

public  class  CharacterMovement : MonoBehaviour
{
//instanca RigidBody komponente za igrača
private  Rigidbody2D playerRigidBody2D;
private float movePlayerVector;
private bool  facingRight;

// brzina igrača
Public  float speed = 4.0f;

// inicijalizovanje bilo koje reference [1]na komponentu Awake se poziva pre start-a
void  Awake()
{
      playerRigidBody2D = (Rigidbody2D)GetComponent(typeof(Rigidbody2D));
}

// Update se poziva jednom po frejmu
void Update ()
{

movePlayerVector = Input.GetAxis("Horizontal"); //uzima horizontalnu osu  zbog kretanja igrača

playerRigidBody2D.velocity = new  Vector2(movePlayerVector * speed,
playerRigidBody2D.velocity.y);

// kada će se pozvati f-ja za okretanje igrača u drugu stranu
if (movePlayerVector > 0 && !facingRight)
        {
                                Flip();
        }
        elseif (movePlayerVector < 0 && facingRight)
        {
                                Flip();
        }
```

[1]Referenca je vrsta promenljive u programiranju, koja čuva adresu neke druge promenljive i sve operacije koje se čine na njima, prosleđuje promenljivima na koje sama pokazuje.U ovom slučaju kada se kreira objekat, u okviru Unity programa dodeljuje mu se komponenta, promenljiva kojoj se dodeljuje predstavlja pokazivač (referencu) na tu komponentu. Pri prenosu u metod, prenose se reference, a ne komponenta.

```
        void  Flip()
        {
            //f-ja za transformisanje pozicije igrača i okretanje na levu/desnu stranu
            facingRight = !facingRight;

            Vector3 theScale = transform.localScale;
            theScale.x *= -1;
        transform.localScale = theScale;
        }
}
```

Kratko ćemo prokomentarisati kod, kako bismo lakše razumeli kako naša skripta funkcioniše.

- GetAxis vraća vrednosti od -1 do 1. Pritiskom na dugme (uglavnom strelicama levo i desno). Na vrednost koja se vraća utiče i to koliko se dugo drži taster. Tako da ako se duže drži to se više povećava vrednost.
- Velocitz se koristi kada koristimo fiziku u Unity-u. Ukoliko na primer dodajemo objektu rigibody i želimo da napravimo skok, tada da ne bismo računali formulom kosog citca, Unity nam pruža mogućnost lakšeg računanja, pri čemu mu dodajemo velocity.
- localScale iliti lokalno skaliranje objekta. Svaki objekat ima globalScale koji je u odnosu na scenu i localScale koji je u odnosu na objekat na kojem je ugnježden.

Poglavlje 4. Getting Animated

U ovom poglavlju ćemo proći kroz sve funkcije koje su bitne za novi 2D sistem i preći kroz nekoliko saveta i trikova koji će vam pomoći da lakše i brže napravite animaciju po želji.

Ovo poglavlje vam neće pokriti sve što se može uraditi sa mehaničkim sistemom ali teme koje ćemo preći su sledeće:

- Istorijat animacija
- Sprite animation
- State machines and Mecanim
- Curves and fine control

4.1 Istorijat animacija

Animacija potiče od latinske reči „*animatus*“ što znači dati život – oživeti. Postupak stvaranja iluzije kretanja crtežima, modelima ili beživotnim stvarima.
Najpoznatiji metod prezentovanja animacije jesu pokretne slike.
Pozicija svakog objekta u bilo kojoj slici se odnosi na položaj tog objekta na prethodnoj i narednoj slici, tako da se objekti nesmetano kreću nezavisno jedan od drugog.
Korišćeni su animatori za crtanje slika koje su stavljane na statične pozadine i tako su dobijani zasebni frejmovi.
Sličan koncept se koristi i u sprajt animacijama i da bi ovaj koncept implementirali potrebno je pratiti sledeće korake:

- Kod za praćenje niza animacije
  Ovo znači da moramo da znamo ukupan broj cel-ova, trenutni cel koji se prikazuje, vreme trajanja svakog cel-a i ukupno proteklo vreme.
- Ponavljanje kroz animaciju
  Moramo dodati proveru, koliko je vremena proslo i da uvecamo cel brojac u odgovaracujem trenutku da bi presli na sledecu sliku u animaciji
- Renderovanje odgovarajućeg okvira iz sprajt liste
  Cel-ovi su raspoređeni u sprajtu tako da se prvi cel nalazi na poziciji 0,0 drugi je na poziciji width,0 treci je na width*2,0 i tako dalje. Ovaj vid strukturisanja sprajta olakšava kretanje kroz niz celova

Trajanje jednog crteža na ekranu se može izračunati na osnovu brzine prikaza frejma (slike). Ako animacija zahteva 20 frejmova po sekundi, trajanje jednog frejma se može izračunati:

$$\frac{\text{seconds}}{20\ \text{frames}} \times \frac{1{,}000\ \text{milliseconds}}{\text{second}} = 50\ \frac{\text{milliseconds}}{\text{frame}}.$$

Ovo omogućava da animacijana svakom procesoru radi isto, zato što se brojač uvećava za odgovarajuci vremenski interval koji je nezavisan od brzine procesora.

4.2. Sprite animation

Pored 2D sprite sistema, Unity je stvorio i unapredio animacije u 2D sistemu koje su lakše za korišćenje. Uveliko je poboljšan dope sheet, animation/clip controller i sve je dobro integrisano sa Mecanim sistemom za upravljanje davajući vam veći izbor funkcija koje su lake za korišćenje kada se sa njima jednom upoznate.

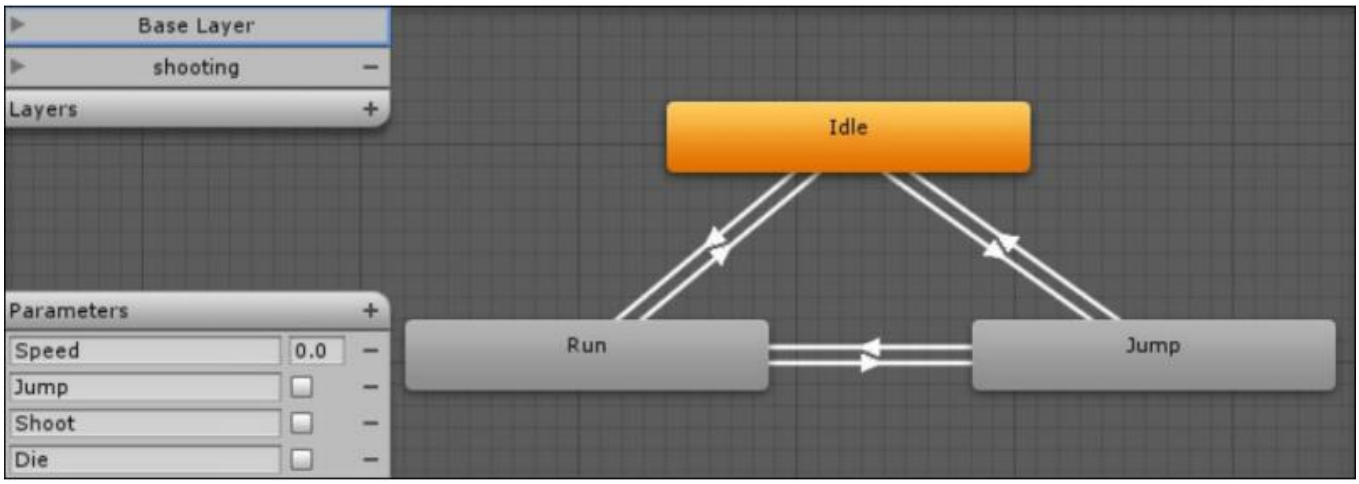
4.3. Animation components

Sve vezano za animacije u novom 2D sistemu koristi Mecanim Unity sistem za projektovanje i kontrolu. Podeljen je na tri glavna dela:

- Animation controllers
- Animation clips
- Animator components

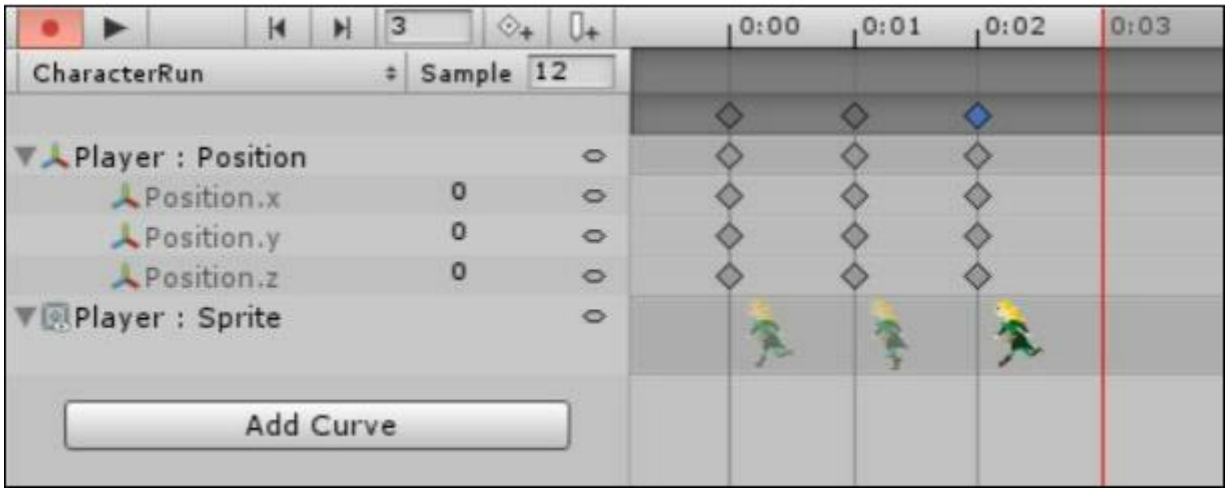
4.3.1. Animation controllers

Kontroler animacija je prosto rečeno stanje mašine koja se koristi da kontroliše kada animacija treba da počne i koliko često, uključujići i to koji uslovi moraju biti ispunjeni da bi se iz jednog stanja prešlo u neko drugo stanje. U novom 2D sistemu mora postojati makar jedan kontroler po animaciji i kontroler može sadržati više animacija, kao što možete videte ovde, tri stanja i tranzicionim linijama između njih:



4.3.2. Animation clips

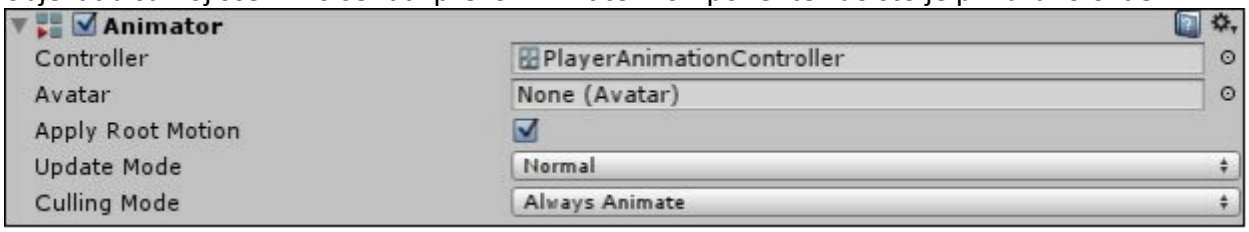
Klipovi animacija su srce animacionog sitema.



U animacionom dope sheet sistemu sa prethodne slike možete pratiti gotovo svaku promenu vezanu za sprajtove koja se dogodi u **Inspektoru**, dozvoljavajući vam da animirate gotovo sve. Možete čak kontrolisati koji sprajt iz sprajtšita se koristi za svaki frejm animacije.

4.3.3. Animator component

Da biste koristili novu animaciju pripremljenu u kontroleru, potrebno je primeniti na gejm objekat u samoj sceni. To se radi preko **Animator** komponente kao što je prikazano ovde:

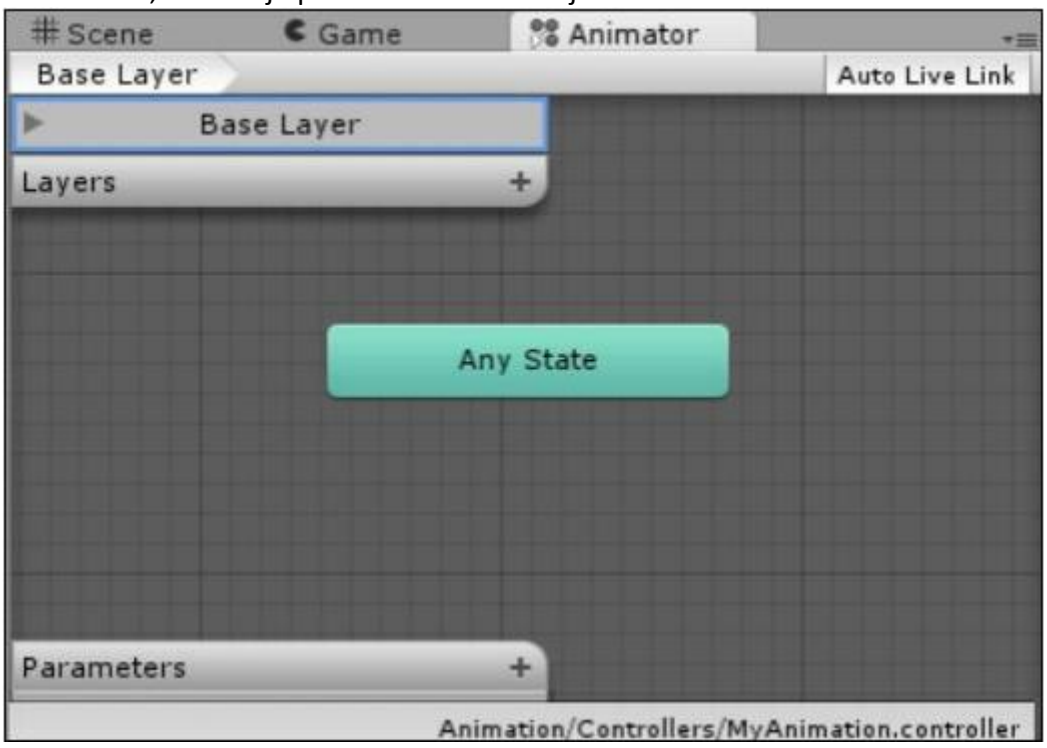


Jedina bitna stvar je **Controller** property. Ovde dodajemo kontroler koji smo prethodno kreirali.

4.4. Podešavanje kontrolera animacija

Dakle, za početak stvaranja animacija, potreban vam je Animation Controller u cilju definisanja klipa animacije. Kao što je ranije navedeno, ovo je samo stanje mašine koja kontroliše izvršavanje animacije čak i ako postoji samo jedna animacija. U ovom slučaju kontroler pokreće izabranu animaciju onoliko dugo koliko je rečeno.
Kreiranje kontrolera animacije je jednako lako kao i bilo koji drugi gejm objekat.

U **Project** view, jednostavno kliknite desnim tasterom miša na view i izaberite **Create | Animator Controller**.
Otvaranje nove animacije prikazaće vam se prazan kontroler animacija u Mecanim state prozor menadžeru, kao što je prikazano na sledećoj slici:



Kada imate kontroler možete ga dodati bilo kom objektu u igri tako što ćete kliknuti na **Add Component** u inspektoru ili prevući do **Component | Create and Miscellaneous | Animator** i izabrati ga. Zatim možete da izaberete svoj novi kontroler kao kontroler za dati properti animatora. Alternativno možete jednostavno prevući novi kontroler na objekat u igri kojem želite da ga dodate.

4.5. Podešavanje klipova animacije

Zapravo postoje dva načina kreiranja animacija (ili klipova) to je zato što Unity kontroliše kreiranje animacija, a ne neki spolji alat.

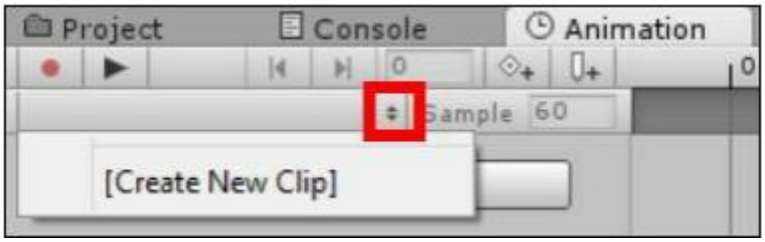
4.5.1. Manuelno kreiranje klipova animacija

Počnimo kreiranje neke animacije ručno. To je malo duži put kada se radi ručno/manuelno ali je dobar put da se razume šta automatski sistem u stvari radi za nas.
Za početak prvo moramo da stvorimo gejm objekat u hijerarhiji sa animator komponentom i kontrolerom zakačenim za dati objekat, tako da možemo dodati referencu sprajtšit-u. Pratite sledeće korake:

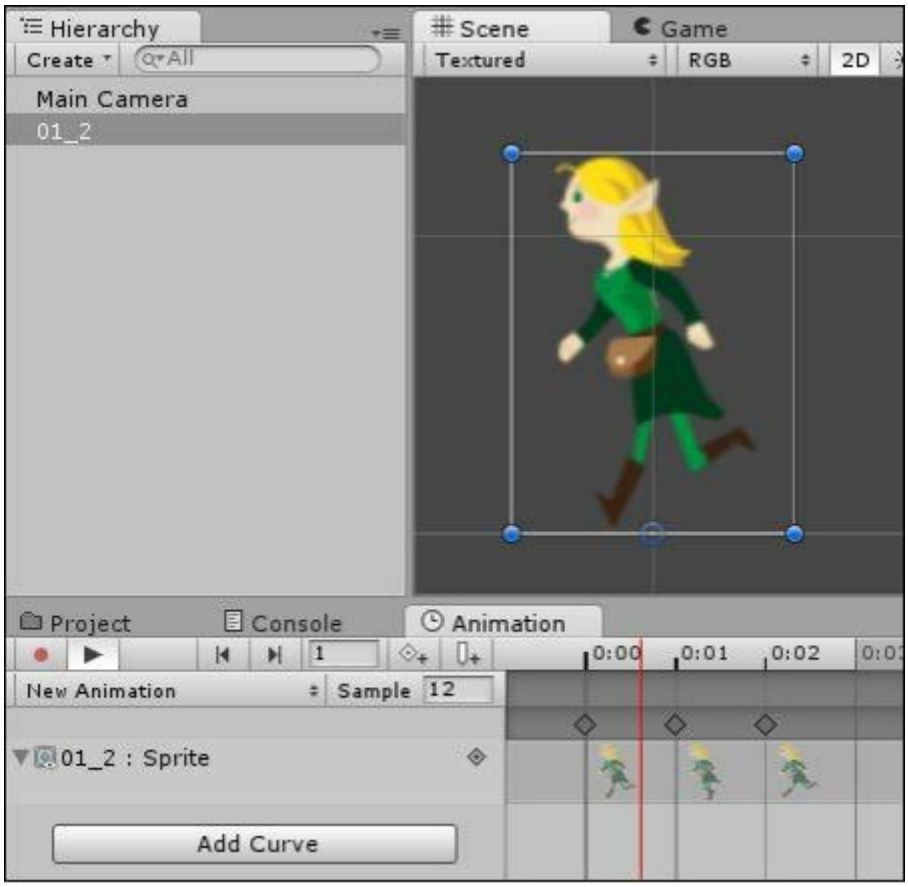
1. Prevucite ili listu sprajtova ili bilo koji sprajt na scenu. Ovo će kreirati novi gejm objekat sa dodeljenim **Sprite Renderer**-om.
2. Dodajte **Animator** komponentu na gejm objekat.
3. Kreirajte nov animator kontroler klikom na **Animator Controller** u **Project** view-u.
4. Prevucite nov animation controller na gejm objekat, ili ga postavite u **Inspektor** panelu.

Sada ako otvorite prozor **Animator(Window | Animation**) sa izabranim gejm objektom, dobićete praznu animaciju bez ikakvih definisanih klipova, kao što je prikazano na sledećoj slici:

Da biste napravili svoj prvi klip, jednostavno kliknite na klip sekciju iz padajućeg menija i izaberite **Create New Clip**, kao što je prikazano ovde:



Odavde, možete da prevlačite pojedinačne sprajtove na vremensku liniju u tačkama u kojima želite da prikažete sprajt u vašoj animaciji, ili možete da izmenite svojstvau **Inspektor** panelu, da animirate kako sprajt treba da izgleda na ekranu. Možete čak pregledati animaciju, klikom na dugme **Play**i on će vam prikazati animaciju trčanja u **Scene** view prozoru.

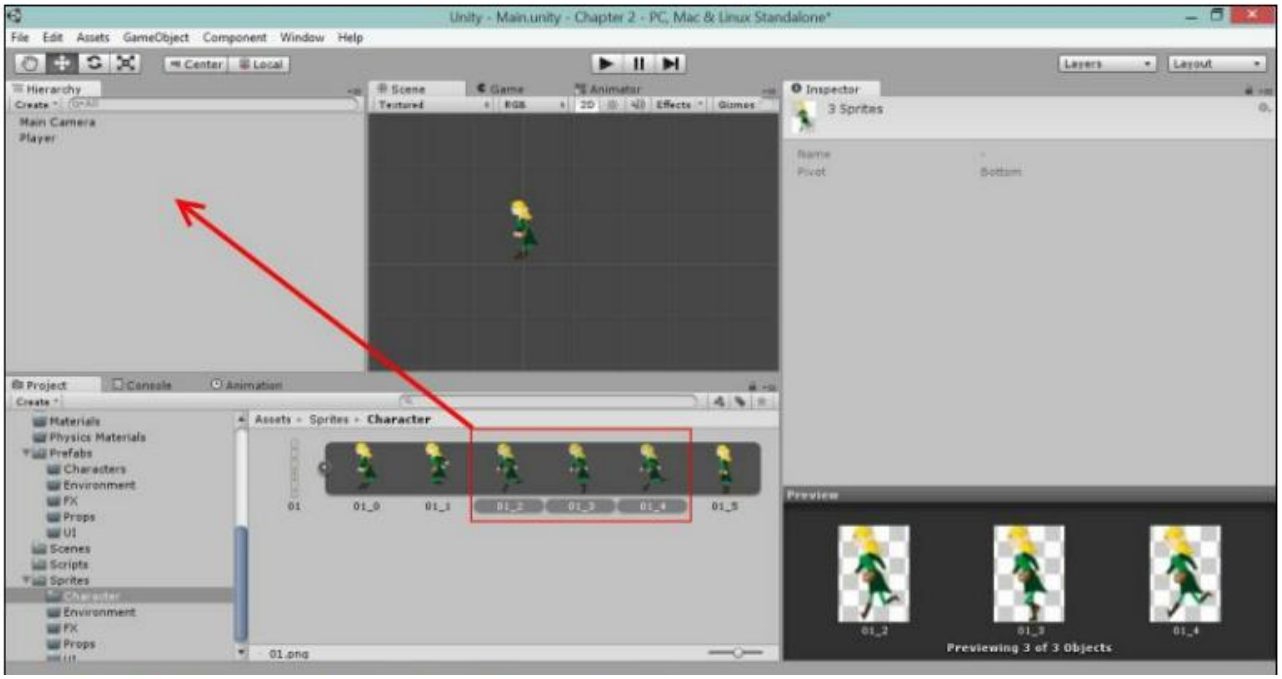


4.5.2. Automatsko kreiranje klipova animacija

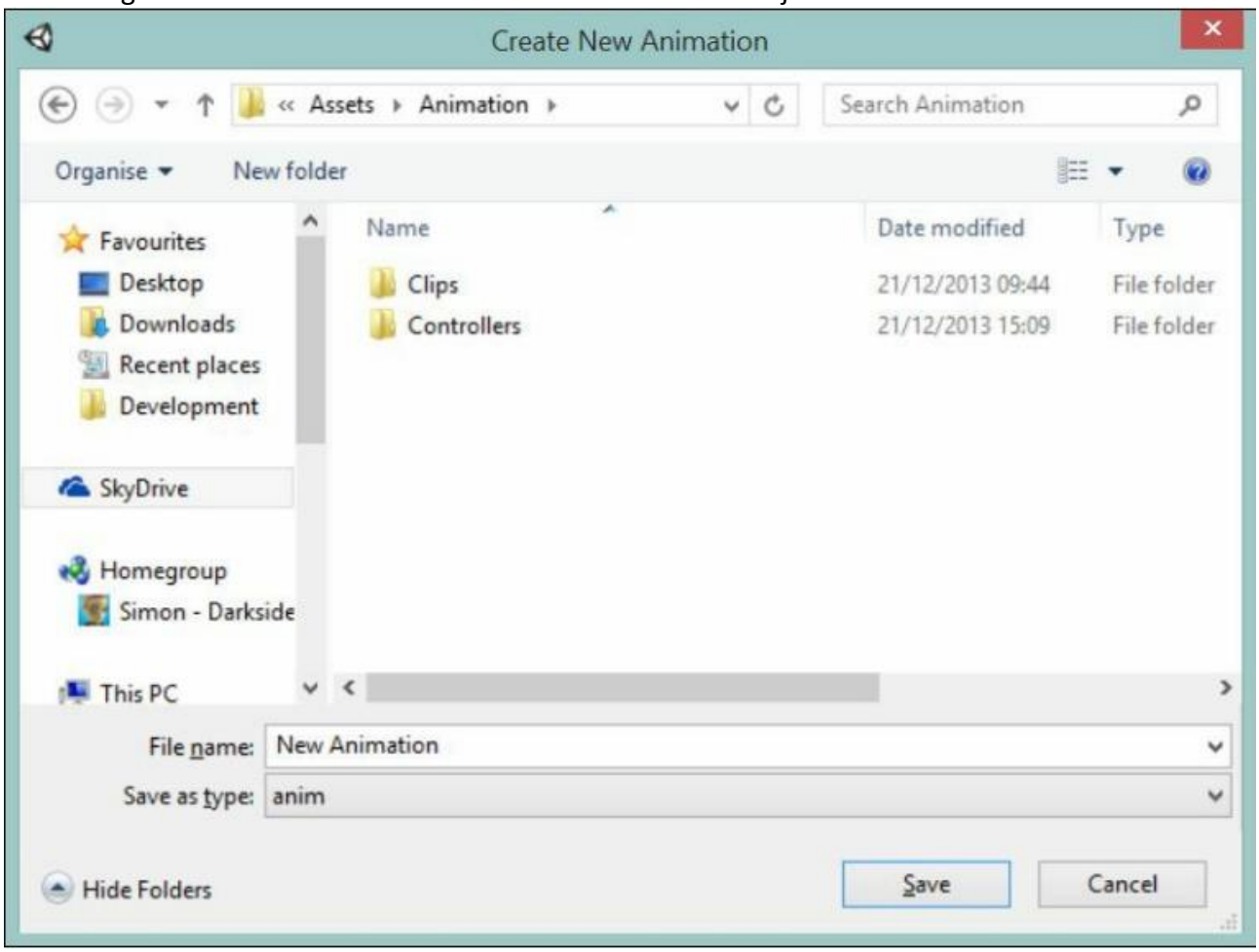
Pristup koji se preporučuje (posebno za početnike) je pustiti Unity da kreira klipove za vas. Automatsko kreiranje je napravljeno tako da razume vase animacije kao i da podesi razne propertije umesto vas, a vi uvek možete da ih podesite i promenite kasnije.

Proces je veoma jednostavan, možete kreirati podrazumevani animacioni kontroler u procesu, prateći sledeće korake:

1. Prvo, izbrišite primer koji ste prethodno napravili, samo da bi izbegli zabunu.
2. Potražite vaš sprajtšit koji je u Assets folderu.
3. Proširite sprajtšit klikom na strelicu pored slike.
4. Izaberite sve sprajtove za jednu animaciju.
5. Prevucite selektovane sprajtove na Scenu.



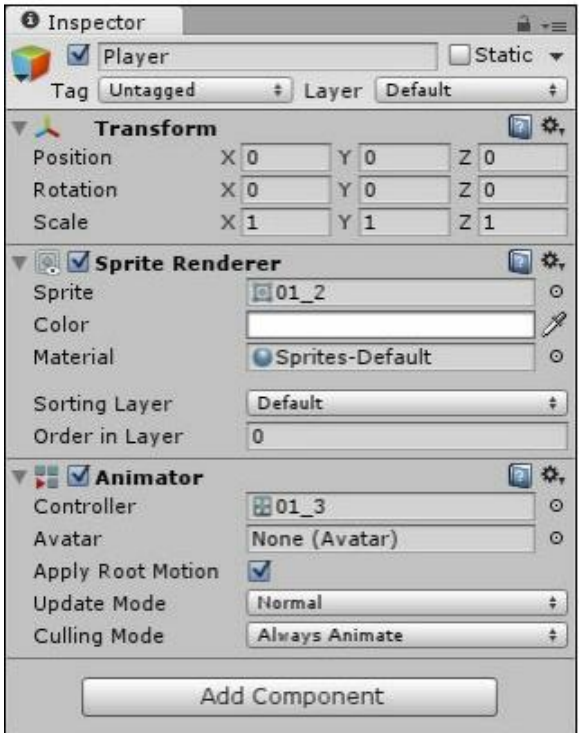
6. Nakon toga biće vam zatraženo da snimite vašu novu animaciju.



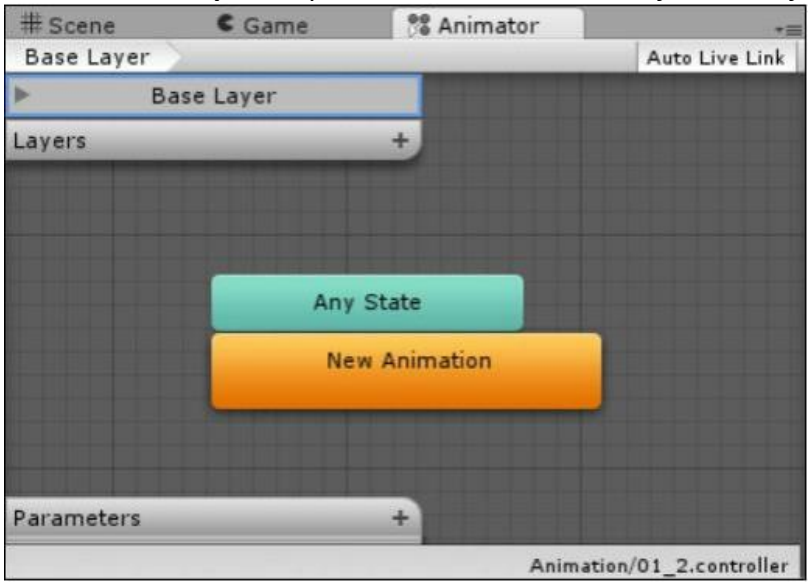
7. Snimite u Assets\Animation\Clips i gotovi ste.

Jednom kada snimite budite sigurni da ste sortirali vaše kontrolere i klipove u pravim folderima vašeg projekta radi lakšeg pronalaženja i održavanje strukture samog projekta.

Ako pogledate novi gejm objekat u **Inspektor** panelu, naći ćete novi sprajt kreiran sa animator komponentom zajedno sa novim kontorlerom koji ste snimili, prikazan u **Inspektor** panelu kako što možete videte ovde:



Sada, duplim klikom ili na novi kontroler u **Project** folderu ili na parameter **Controller**u**Animator** kontroleru u **Inspektor** panelu, videćete novo stanje animacije koje je generisano za vas:



Ako želite da proverite kako animacija izgleda, samo izaberite vaš novi objekat u **Project** hijerarhiji i otvorite **Animation** view (**Window | Animation**). Jednom kada je otvoren, trebalo bi da vidite animator **Dope Sheet** view (kao što je prikazano na sledećoj slici) za trenutno izabrani gejm objekat sa trenutnom animacijom u view-u.

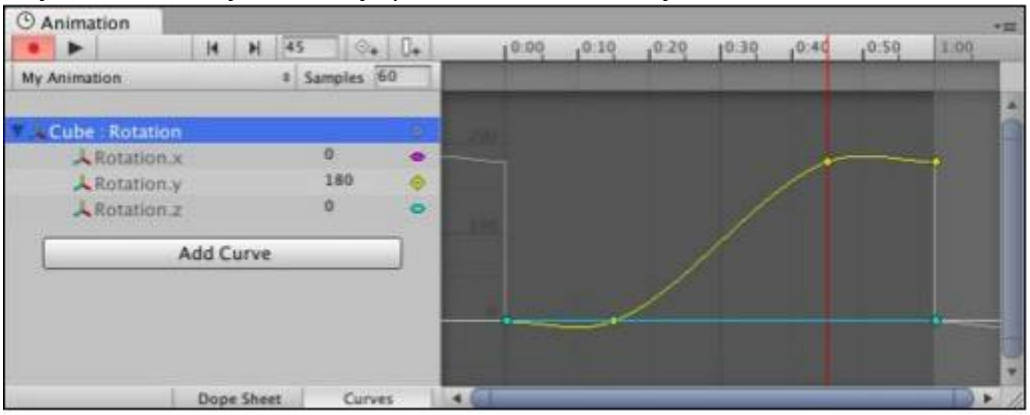
Ovde možete videte jednostavnu sprajt animaciju u samom **Dope Sheet**-u, sa terminima promena sprajt slika. To će biti petlja koja će se stalno ponavljati od početka do kraja.
Bilo koja izmena u samom **Dope Sheet**-u biće registrovana i u **Inspektor** panelu, pa budite oprezni na šta klikćete.

4.5.3. Animator Dope Sheet

Pomoću ovog animatora pravljenje animacija je lakše i same animacije postaju bolje:

Ovde imamo time/recording kontrole, izbor animation clip, sample rate (frames per second), animation properties, timeline kao i dope/curve view.

- **time/recording controls**
  Vam omogućava da vidite kako teče sama animacija
  Postoji i dugme za dodavanje novih ključnih tačaka (**KeyFrames**) ili animacionih događaja (**Animation Events**)
- **sample rate** (frames per second)
  Određuje broj frejmova po sekundi koji su dostupni na vremenskoj liniji, koje kontroliše određen broj ključnih tačaka između intervala.
- **Animation properties**
  Lista različitih propertija koja kontrolišu ovu animaciju
- **Timeline**
  Prikazuje sve ključne frejmove koji se animiraju tokom trajanja animacije.
- **Dope/curve view**
  Timeline view ima alternativni režim za finije upravljanje i kontrolu krivih između ključnih frejmova animacije kao što je prikazano na sledećoj slici:



Za uređivanje krive potrebne su male finese, koje čine da tranzicija između stanja lepše izgleda nego sa Bool ean (on/off) efektom.

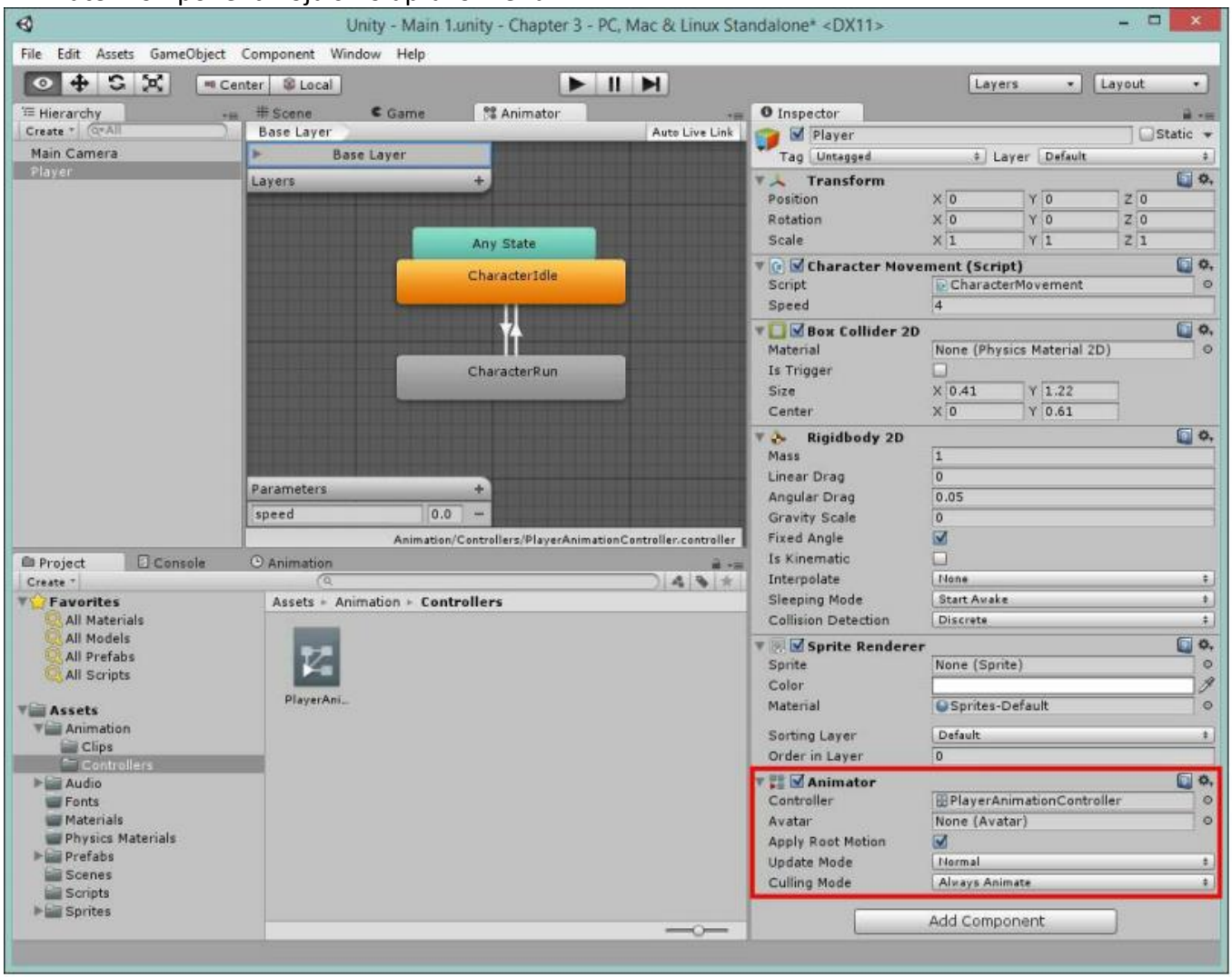
4.6. Spajanje

Do sada naš junak može da se kreće levo i desno, ali njegovo kretanje je malo ravno, ubacimo malo akcije i dinamike u njegovom kretanju.
Kako bi se dobila animacija trčanja, potrebni su nam ispunjeni sledeći preduslovi:

- Animator komponenta
- Animator kontroler
- Makar jedna animacija

4.6.1. Podešavanje animation controller-a

Da bismo pocečeli, potrebno je dodati **Animator** komponentu (**Add Component | Miscellaneous | Animator**) u **Inspektor** panelu. Zanemarite sve ostale opcije za sada. Sledeće, kreirajte nov animation **Controller**u*Assets/Animation/Controllers* (**Create | Animator Controller** u **Project** prozoru). Zatim, prevucite novi kontroler do **Controller** property u **Animator** komponenti koju smo upravo kreirali.

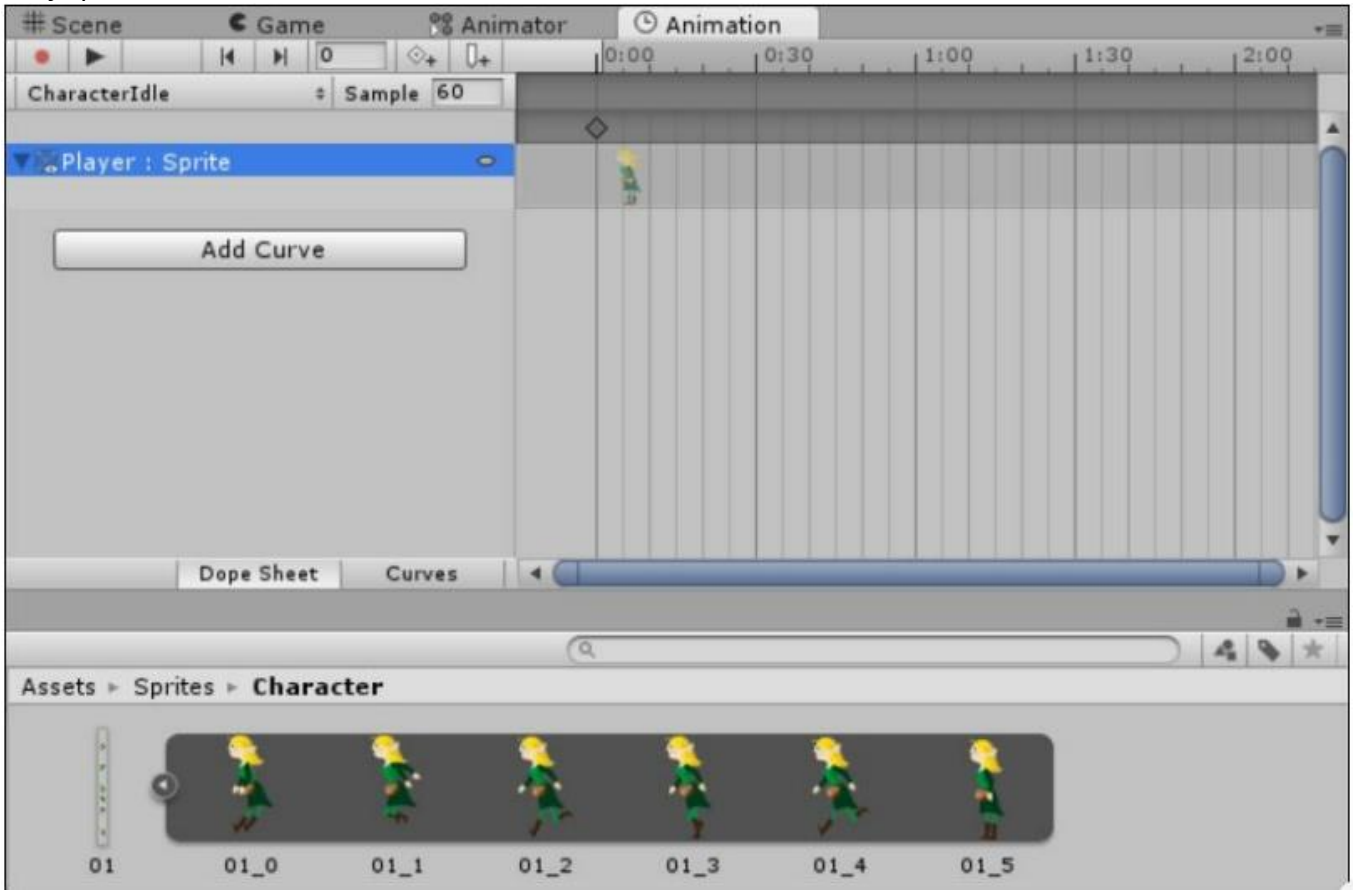


Sada, treba dodati našu animaciju. Otvorite nov animation **Controller** – duplim klikom, zatim otvorite **Animation** prozor (**Menu | Window | Animation**).

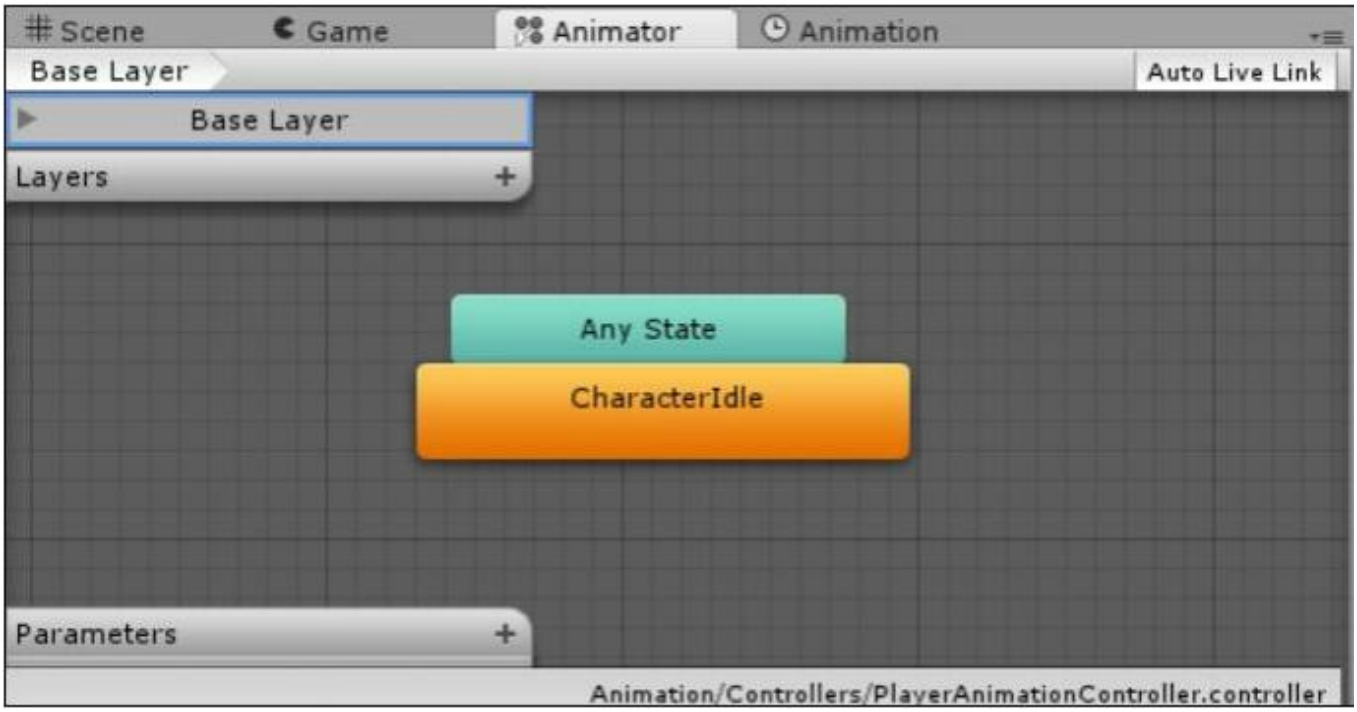
4.6.2. Dodavanje prve animacije (idle)

Da biste dodali svoju prvu animaciju, kreirajte novi klip otvaranjem padajućeg menija i izaberite opciju **Create New Clip**. Ovo će vas odvesti na deo koji smo prešli ranije u *Animator Dope Sheet* delu. Snimite pod imenom *CharacterIdle.anim* na lokaciji *Assets/Animation/Clips*.

Za animaciju u ovom stanju potreban je dodati samo jedan sprajt koji prikazuje našeg heroja u stanju mirovanja, pa prevucite sprajt pod imenom **01_5** na vremensku liniju na poziciji **0**, kao što je prikazano ovde:

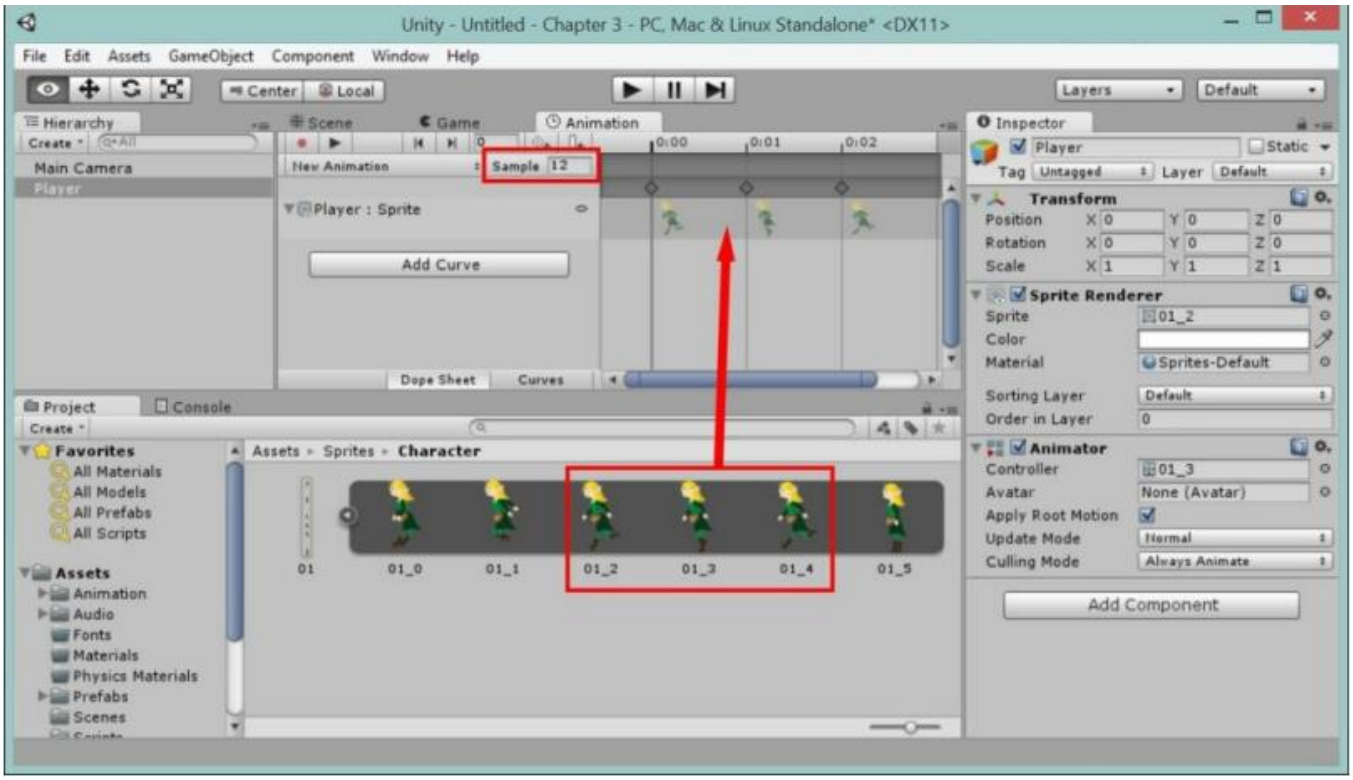


Animation controller bi sada trebalo da izgleda kao na sledećoj slici, sa jednim stanjem za animaciju u stanju mirovanja:

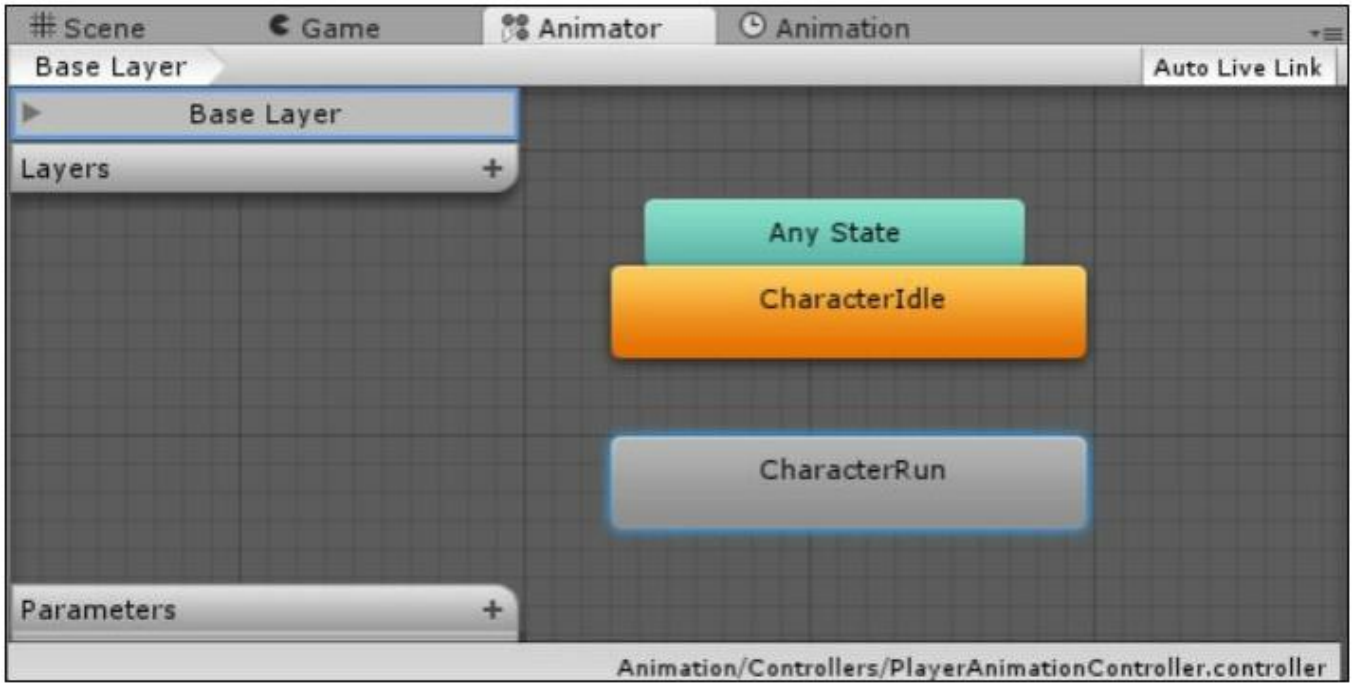


4.6.3. Dodavanje još jedne animacije (run)

Da biste dodali animaciju koja oponaša trčanje, moraćete da dodate još jedan klip. Ovaj put dajte mu naziv *CharacterRun* i umesto da prevučete jedan sprajt, dodajte tri sprajta koja će formirati jednu animaciju. Prvo podesite **Sample** rate u **Animation** prozoru na 12 frejmova po sekundi, zatim izaberite sprajtove sa nazivima **01_2**, **01_3** i **01_4** i prevucite ih u timeline prozor, koji će ih poređati na odgovarajući način na vremenskoj linijij kao što je prikazano na sledećoj slici:



Gledajući **Animation** controller sada videćete da je dodato još jedno stanje. Glavna razlika je ta što ovo stanje nije narandžasto već je sivo. Razlog je taj što ovo nije podrazumevano stanje.

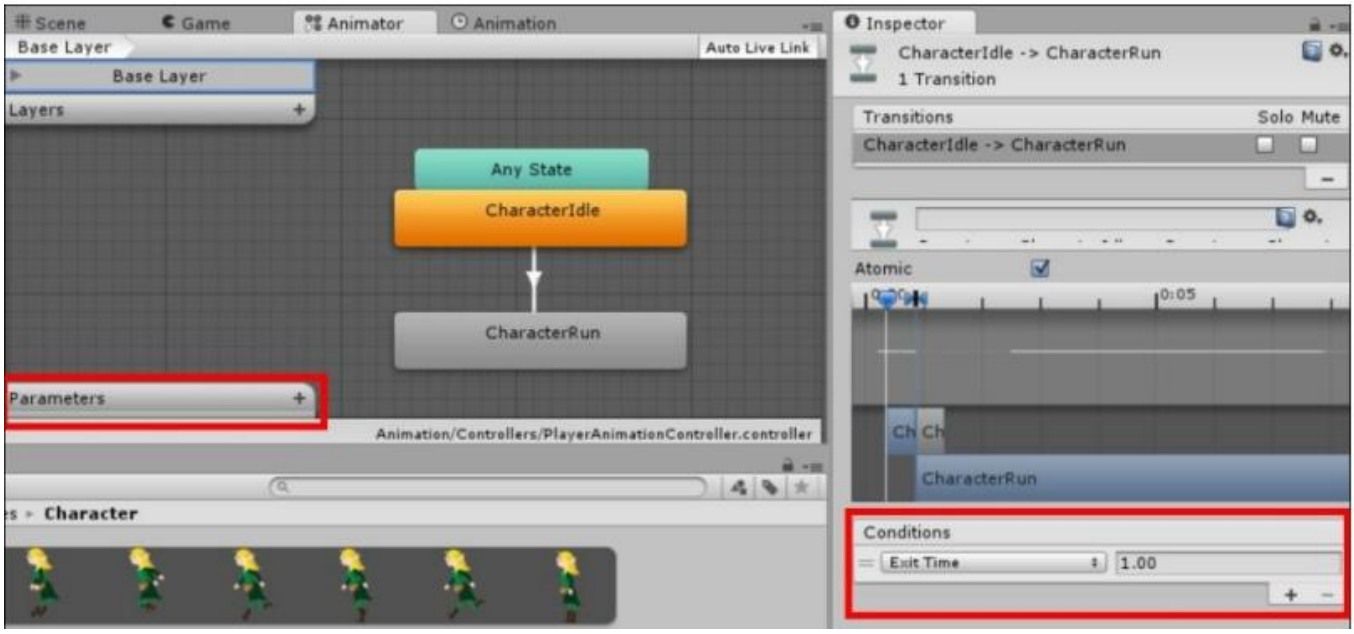


4.7. Povezivanje stanja

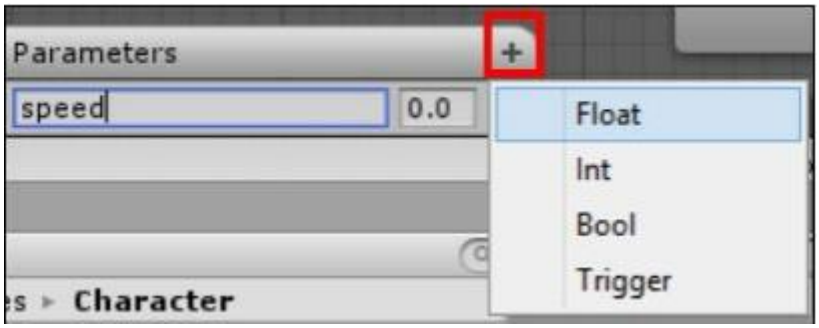
U ovom trenutku, naša stanja nisu povezana. Ako pokrenemo projekat, heroj će uvek biti u idle stanju – stanju mirovanja.
Da bismo rekli kontroleru da promeni stanje, potrebno je sledeće:

- Tranzicioni link između dva stanja
- Parametar ili događaj koji će aktivirati tranziciju
- Nešto što će promeniti vrednost parametra, obično u skripti

Dakle prvo ćemo kreirati tranziciju između podrazumevanog idle stanja, desnim klikom na CharacterIdle i klikom na **Make Transition**. Ovo će promeniti kursor miša u strelicu. Onda, kliknemo na stanje na koje želimo da promenimo, u ovom slučaju to će biti CharacterRun stanje. Klikom na novu tranziciju, prikazaće se property date tranzicije u **Inspektor** panelu kao na slici:

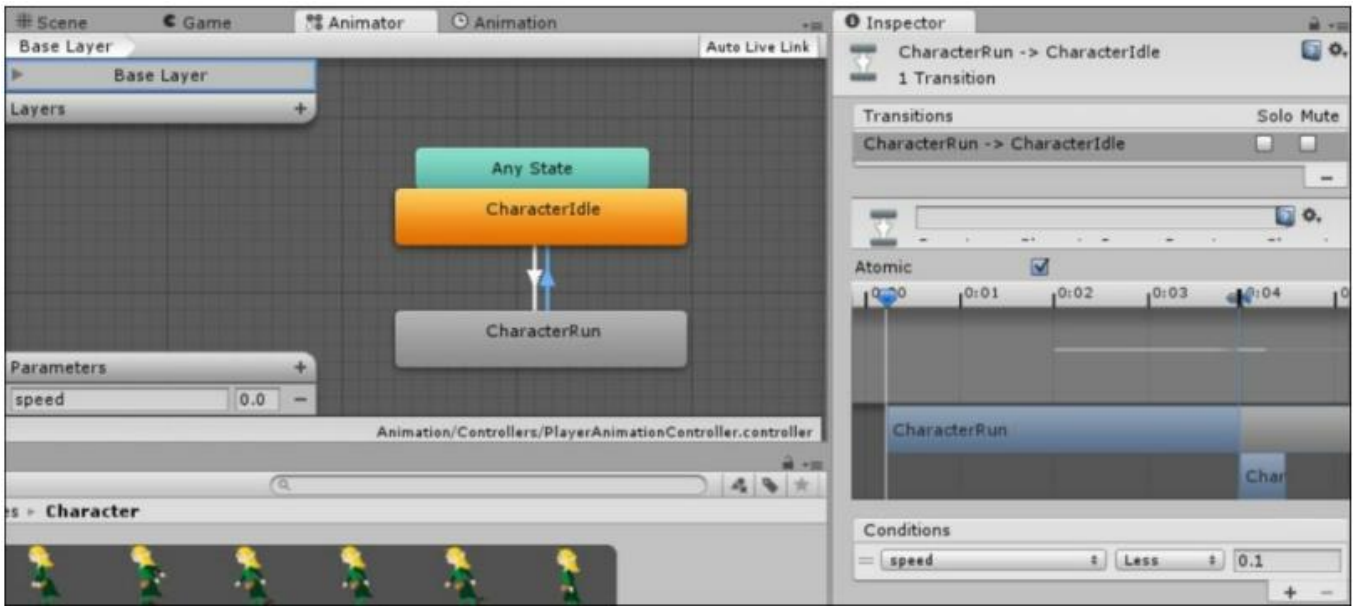


Kao što je prikazano u **Inspektor** panelu, nova tranzicija se kontroliše pomoću jednog parametra nazvanog **Exit Time**, što znači da kada se prva animacija završi, počeće se sa izvršavanjem druge animacije. Mi to ne želimo ovde, već želimo da pređemo u drugo stanje kada je Run animacija aktivirana.
Prvo, potreban nam je nov parameter kojem ćemo moći da pristupimo iz naše skripte. Kliknite na + simbol i dodajte nov **Float** parametar - *speed*.



Sada u **Inspektor** panelu, promenite **Exit Time** parametar klikom na njega i postavljanjem novog parametra – *speed*. Postavite mu vrednost na **0.1**. Ovo će obavestiti animator da kada je vrednost veća od 0.1 da izvrši tranziciju i pređe iz idle stanja u run stanje.

Zatim ponovite procesi dodajte tranzicioni link od **CharacterRun** do **CharacterIdle** stanja, koristeći isti parametar – *speed*, samo sto ćemo sada podesiti vrednost da bude manja od **0.1** kao na sledećoj slici:



Ako pokrenete sada projekat, naš junak i dalje neće trčati kada se pokrene. To je zbog toga što moramo da ažuriramo našu Character movement skriptu.

4.7.1. Pristup kontroleru iz skripte

Kao i većini stvari u Unity-u, za pristup drugoj komponenti potrebna nam je samo referenca ka datoj komponenti.

Ažurirajte vašu CharacterMovement.cs skriptu prateći sledeće korake:

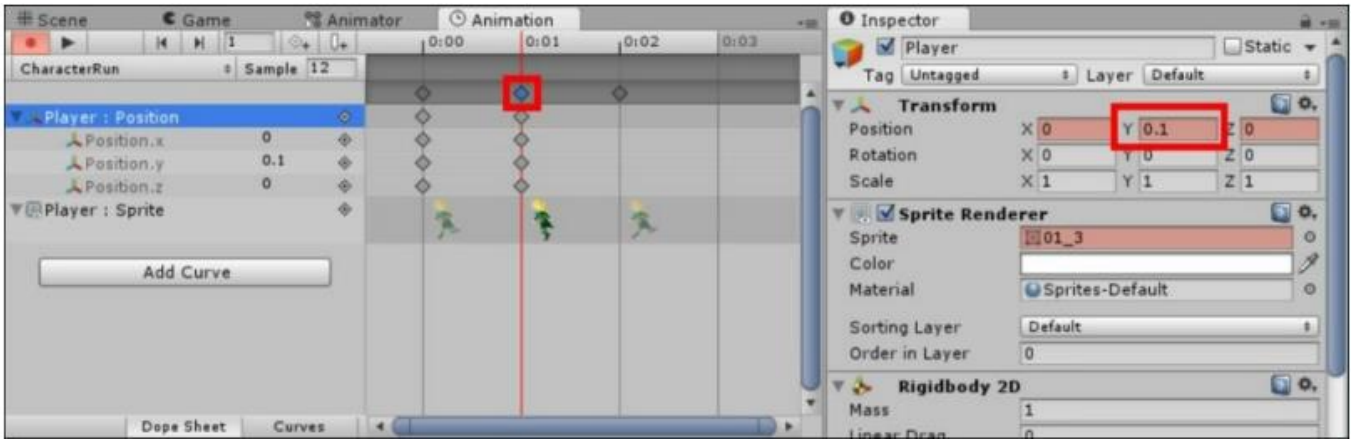
- Nova Animator referenca da održi vezu sa našim sprajtom
  ```
  //Reference to the player's animator component.
  private  Animator anim;
  ```
- Postavljanje stvarnog animatora iz sprajt objekta se vrši u Avake() funkciji
  ```
  void   Awake() {
      //Setting up references.
      playerRigidBody2D = (Rigidbody2D)GetComponent(typeof(Rigidbody2D));
      playerSprite = transform.Find("PlayerSprite").gameObject;
      anim = (Animator)playerSprite.GetComponent(typeof(Animator));
   }
  ```
- U funkciji Update() vrši se ažuriranje float parametra koji smo ranije definisali
  ```
  void   Update(){
      //Cache the horizontal input.
      movePlayerVector = Input.GetAxis("Horizontal");
      anim.SetFloat("speed", Mathf.Abs(movePlayerVector));
   }
  ```

Sada, ako pokrenete projekat, vaš heroj će početi da trči u onom pravcu u kojem ste mu rekli da trči, a onda će odmarati kada prestanete da trčite.

4.7.2. Ekstra credit

Imamo lepu animaciju koja trči i zaustavlja se, ali zar ne bi bilo bolje kada bi se naš junak kretao prirodnije?
Dakle, proširićemo našu animaciju trčanja prateći sledeće korake:

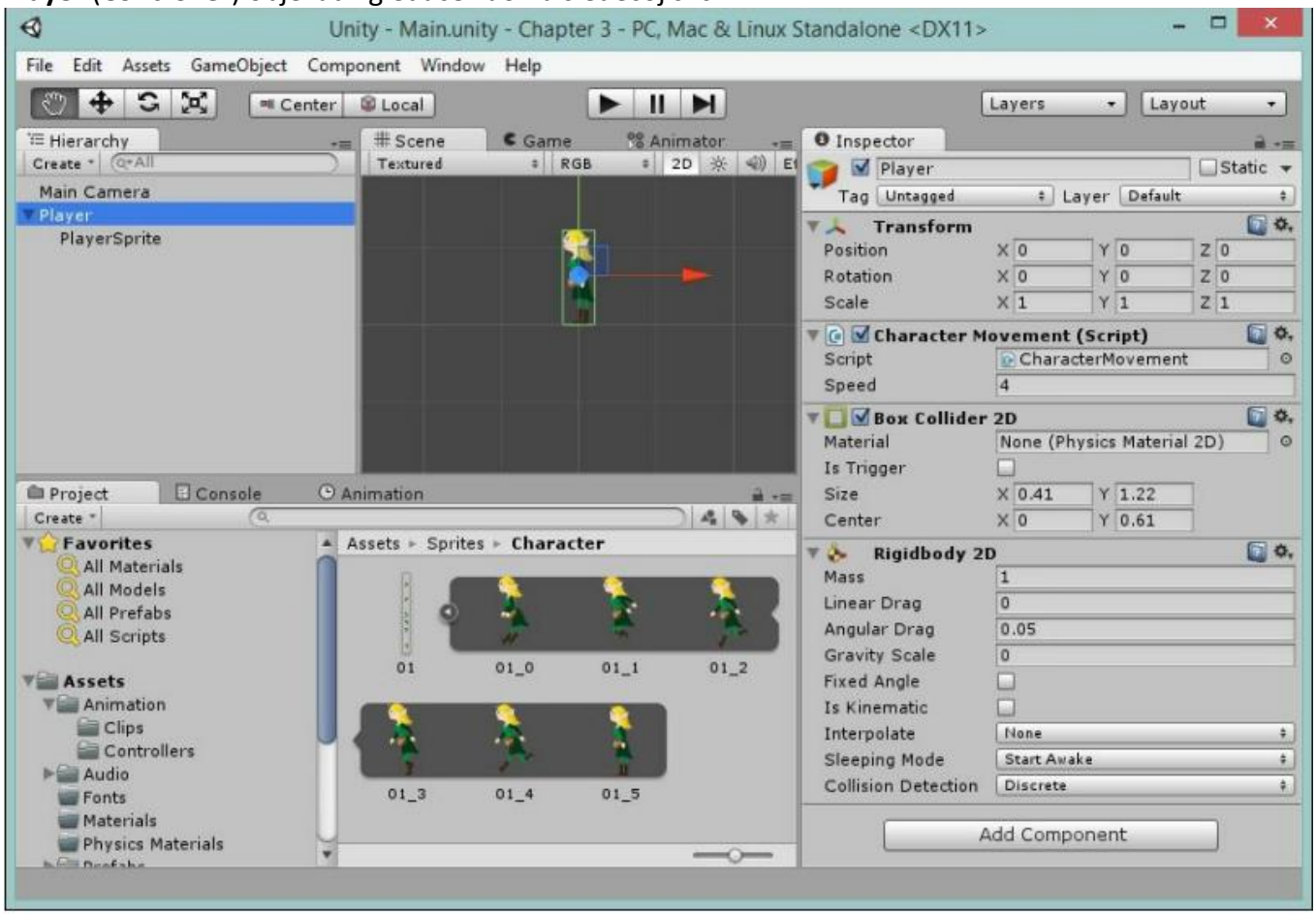
1. Izaberite **Player** gejm objekat, otvorite **Animation** prozor i izaberite **CharacterRun** animaciju.
2. Kliknide na središnji keyframe u animaciji.
3. U **Inspektor** panelu promenite vrednost **Y Transform** na 0.1.



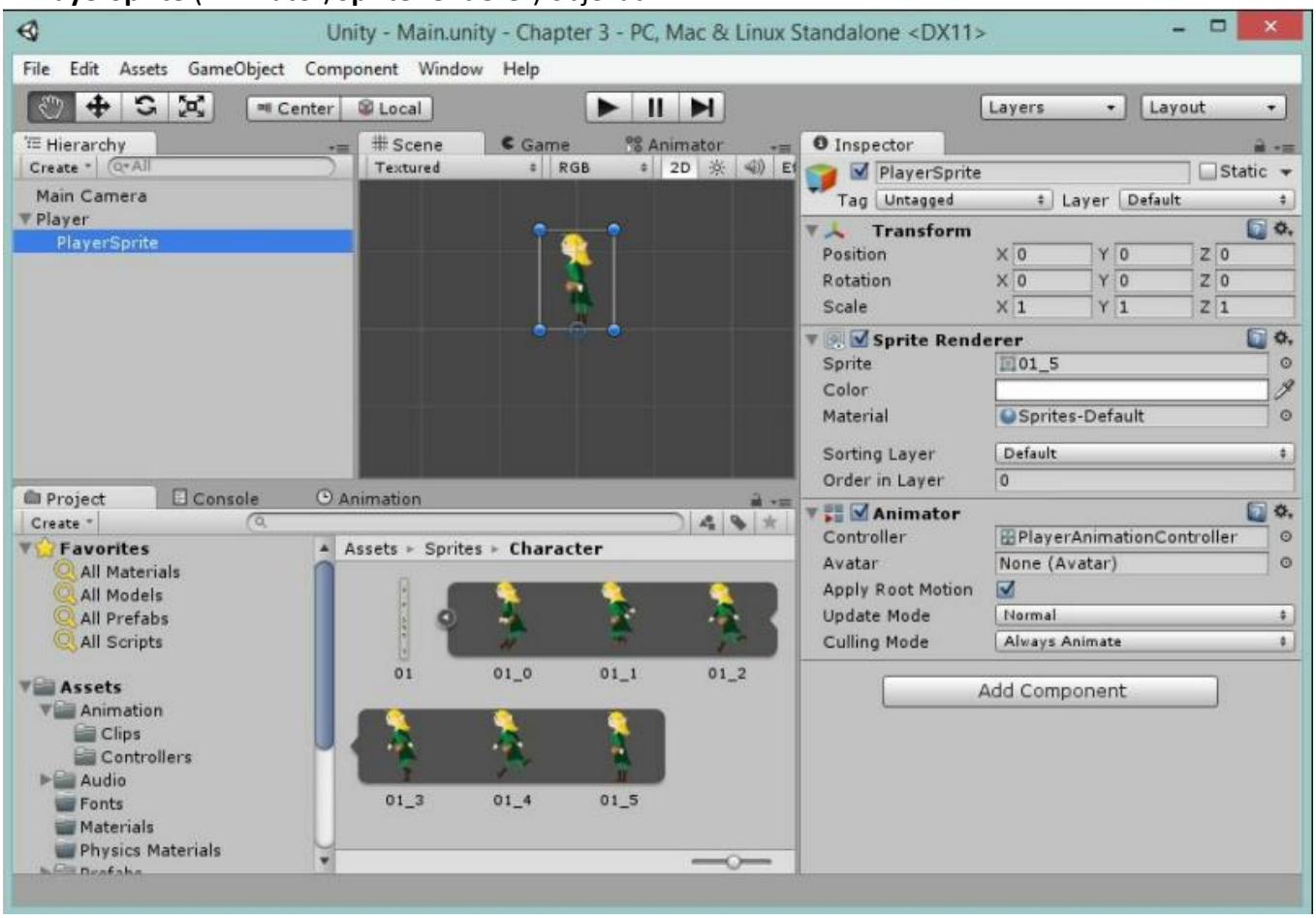
Sledeća stvar koju je potrebno uraditi je:

1. Promeniti naziv našem trenutnom gejm objektu **Player** u *PlayerSprite*.
2. Kreirati novi gejm objekat pod nazivom *Player*.
3. Prevući **PlayerSprite** na **Player**
4. Resetovati sve u **Transform** panelu za oba objekta
5. Skloniti *MovementController*, *RigidPhysics2D* i *Box Collider* skriptu iz **PlayerSprite** objekta i dodeliti ih **Player** objektu ponovo.

Player (**Controller**) objekat izgledaće kao na sledećoj slici:



A **PlayerSprite** (**Animator/SpriteRenderer**) objekat:



Sada je potrebno ažurirati našu skriptu na sledeći način:

1. Izmenite CharacterMovement.cs C# skriptu:
   - Dodajte reference na novi PlayerSprite objekat

```
//Reference to the player's sprite GameObject.
private  GameObject playerSprite;
```

   - U funkciji Awake() dodelite vrednost playerSprite promenljivi

```
void   Awake() {
        // Setting up references.
        playerRigidBody2D = (Rigidbody2D)GetComponent(typeof(Rigidbody2D));
        playerSprite = transform.Find("PlayerSprite").gameObject;
        anim = (Animator)playerSprite.GetComponent(typeof(Animator));
}
```

   - Promenite Flip() logiku da radi sa PlayerSprite objektom

```
void   Flip() {
    //Switch the way the player is labelled as facing.
    facingRight = !facingRight;
    //Multiply the player's x local scale by -1. Vector3
```

```
        theScale = playerSprite.transform.localScale;
        theScale.x *= -1;
        playerSprite.transform.localScale = theScale;
    }
```

Sada kada pokrenemo projekat, **Player** objekat se kreće pomoću skripte, a animacija ide nezavisno.

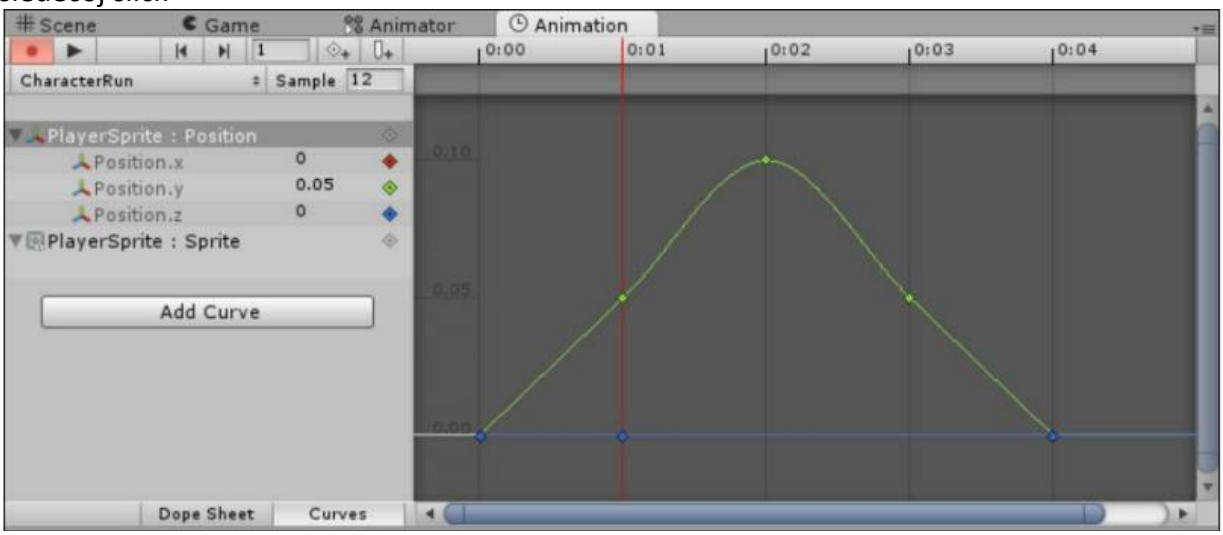
4.7.3. Getting curvy

A sada kao poslednji korak, sredimo našu animaciju trčanja da sve izgleda glatko. Trenutno se heroj kreće gore-dole sa okvirima i granicama koje ima, ali sve se to može poboljšati pomoću krivih koje se nalaze u **Animation** editoru.

Da bismo to uradili, moramo napraviti malo prostora u Animation prozoru, na vremenskoj liniji moramo dodati nekoliko dodatnih frejmova na sledeći način:

1. Pomerimo poslednji keyframe u nazad sa **0.02** na **0.04**
2. Prevucite **01_3** spraj ponovo na **0.02** i **0.03**
3. Postavite **Position.y** property sa 0.01 na 0.05.
4. Postavite **Position.y** property sa 0.02 na 0.1.
5. Postavite **Position.y** property sa 0.03 na 0.05.
6. Kliknite na **Curves** dugme

Sa dodatnim frejmovima i podesavanjima, trebalo bi da vidite krivu kao što je prikazana na sledećoj slici:



Sada prilikom pokretanja projekta, blagi skok u koraku našeg heroja bi trebalo da izgleda kao mnogo lakši pokret.

4.7.4 Zadatak za vežbu

Pokušajte da dodate još animacija iz sprajtšita našeg heroja, kao što je skakanje.

Poglavlje 5. The game world

Sa našim glavnim junakom kreiranim ostaje da mu damo dom i svet u kome će se kretati. Ovde ćemo posvetiti pažnju i tehnikama za kreiranje nekih jednostavnih osnova za dizajn.'Krenućemo od same osnove a to je importovanje (ubacivanje) slika u naš folder assets koje ćemo koristiti u našem svetu a zatim postepeno ćemo nadograđivati grad i ukrasiti ga drugim objektima i elementima sa kojima će naš junak vršiti interakciju.

Ukratko ono što ćemo preći od tema u ovom poglavlju su:

- Rad sa okruženjem
- Korišćenje slojeva sprajtova
- Pukovanje rezolucijom
- Pregled paralaksinga i efekata
- Šejderi (senčenja) u 2D - osnove

5.1. Pozadina sveta i kreiranje slojeva

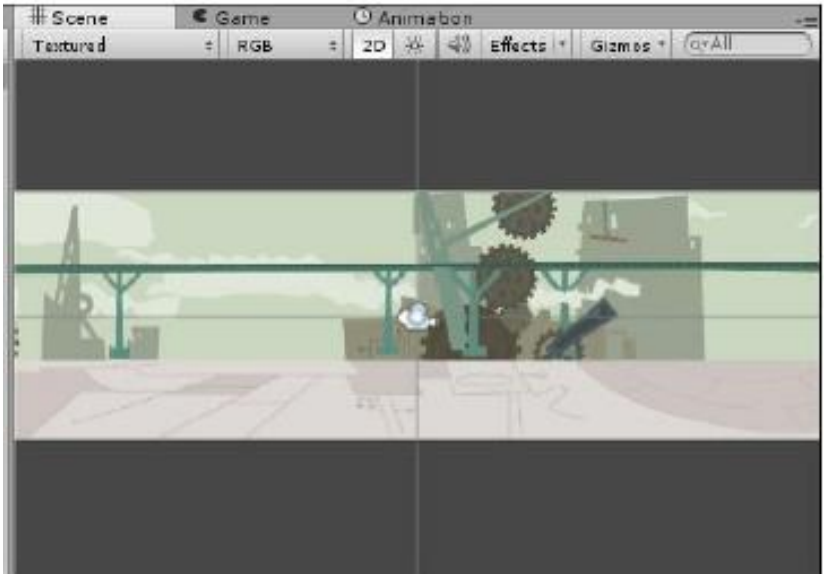
Sada nakon što smo kreirali igrača i oživeli ga uz pomoć animacije možemo da se pozabavimo svetom i okruženjem u kome će „živeti", vršiti interakciju i kretati se.

U folder *Assets\Sprites\Environment* ubaciti slike:

1. background01
2. environmentalAssets
3. environmentalAssets2

Sada ćemo dodati pozadinu sveta koji je jedan single sprajt kao na slici. Prevućičemo pozadinu našeg grada iz sprajt foldera ***background01*** u okruženje glavne kamere.

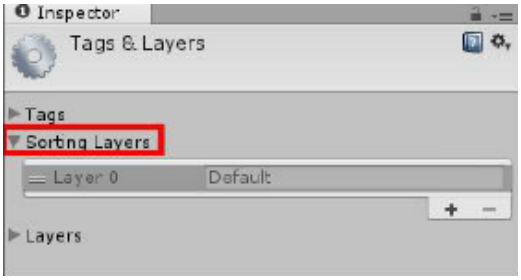
Setovaćemo X = 0,Y = 0 i Z = 0 koordinate iz razloga što Unity ima običaj da postavi pozadinu kao za 3D i to skoro nikad nije tamo gde mi zapravo želimo.



Primetićemo bitnu razliku a to je da je naš junak IŠČEZAO!! Razlog tome je zapravo ***sortiranje slojeva*** koje Unity sistem rešava na 2 načina:

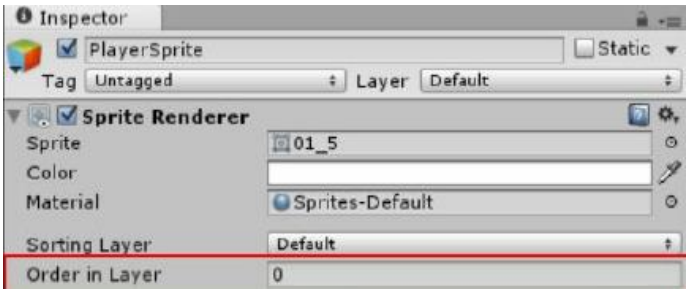
1. **Sortiranje slojeva sprajtova grupisanih u jedan sprajt.**

   Ovakvim načinom sortiranja mi grupu sprajtova pakujemo u jedan grupni sloj kojem možemo dodati više sprajtova i time lakše manipulisati njima kao što je npr prednji sloj grada (ono što će se nalaziti bliže kameri ispred našeg igrača) ili zadnji sloj (iza igrača u pozadini grada).

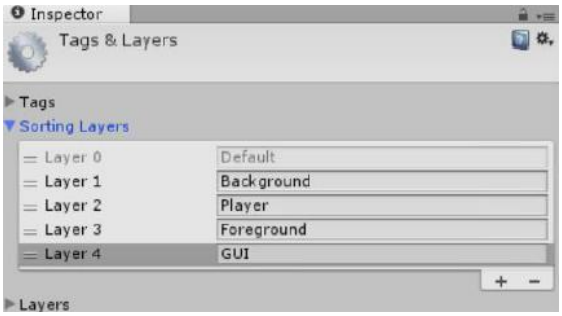


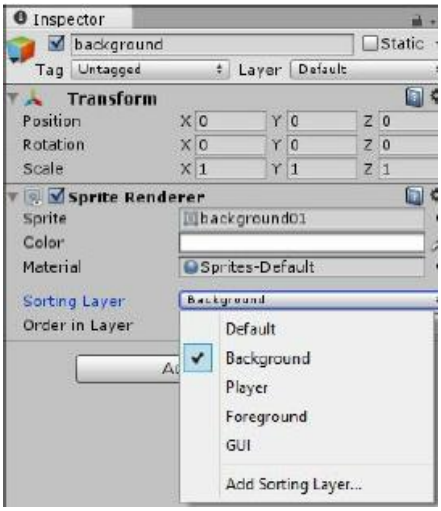
2. **Redosled prikazivanja pojedinačnih sprajtova unutar sloja**

   Ovde pak manipulišemo sa pojedinačnim sprajtovima unutar jednog sloja, odnosno redosled prikazivanja sprajta u odnosu na drugi sprajt iz istog sloja. Time se omogućava sortiranje npr stena ide ispred drveta ili kuca ispred stene itd... Prikaz načina upotrebe sortiranja slojeva je dat na slici ispod.



Unity pojedine slojeve sam kreira pa tako Default-ni sloj ostaje kao nulti jer kada nema drugih dodatnih slojeva svi se automatski pakuju u ovaj! Slojevi mogu svakog časa da se organizuju po želji jednostavnim prevlačenjem sloja iznad ili ispod. Nama trebaju sledeći slojevi:



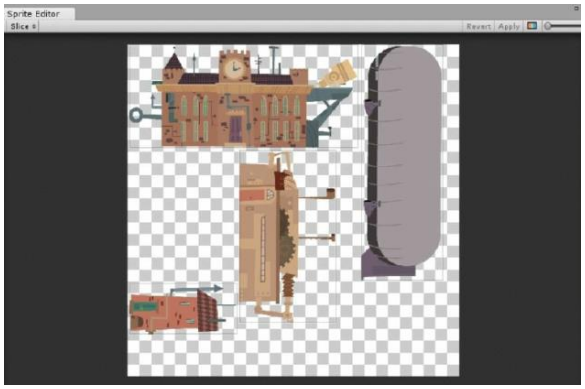


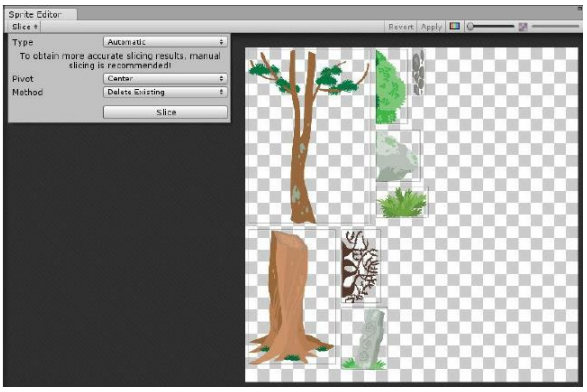
Odabirom našeg glavnog sprajta ***background01*** možemo u inspekt tabu primetiti da stoji ***Sorting Layer i*** odabraćemo da bude ***Background.***Sada se naš junak vidi lepo i dobio je svoje mesto u svetu.

Mogli smo da redosled slojeva odradimo i preko ***Order in Layer*** ali kada se doda previše sprajtova situacija može biti vrlo konfuzna. Zato order in layer koristimo za redosled objekata u zasebnom sloju.

Sada kada smo naučili kako se sortiraju sprajtovi po slojevima i kada smo dodali naš grad možemo da dodamo i ostale delove okruženja a zatim ih raspodeliti po slojevima na prikladan način.

Sprajtove ***environmentalAssets*** i ***environmentalAssets1***koje smo dodali u folder Sprites/Environment treba da isečemo u editoru kao što smo radili i u prethodnim poglavljima tako što ćemo odabrati opciju za sprajt mode ***multiple***u inspector prozorua zatim ***Slice***.



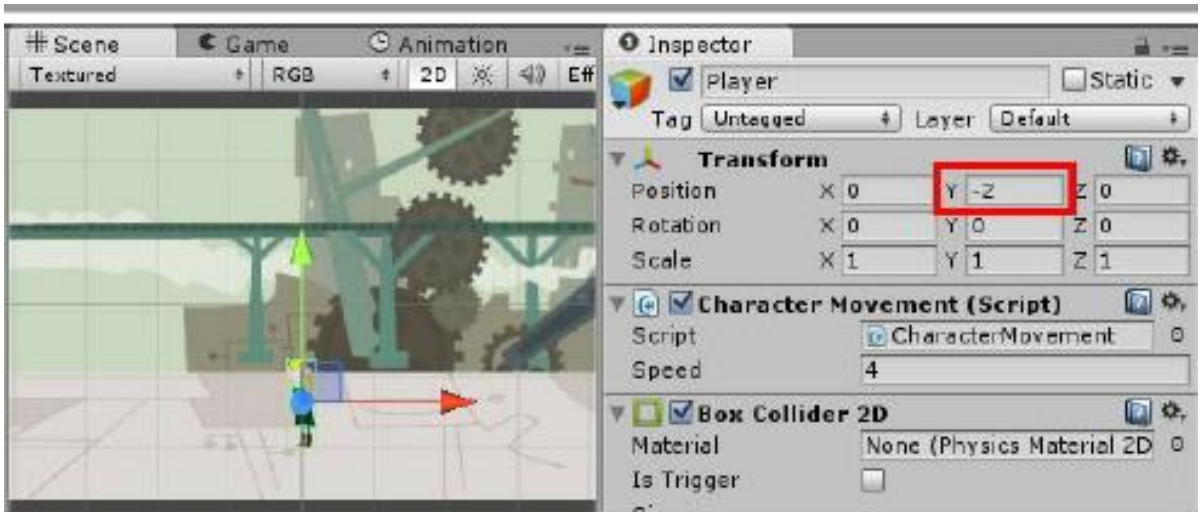


Sada isečene sprajtove (delove) zgrada i okruženja možemo dodati u naš svet po želji i rasporediti ih onako kako nam odgovara.Ako je sve odrađeno kako treba klikom na neki od sprajtova u Sprites folderu pojaviće se strelica na koju se klikom prikazuju svi sprajtovi pojedinačno.

Sada kada je naš igrač postavljen u svet treba da ga postavimo na sredinu puta. To se radi tako što postavimo u polje Y vrednost -2. Pivot smo podesili da bude bottom u prethodnim poglavljima.

Sada je najzanimljivi deo da na sličan način se igramo dodavanjem ostalih sprajtova u svet i njihovim pozicioniranjem kako bi naš junak imao lepo mesto za interakciju i kretanje.

Ukoliko treba neki isečeni sprajt rotirati to se radi u polju Rotation po Z osi.



Zadatak za vežbu 1.

Probajte da samostalno napravite sami od ponuđenih sprajtova ilipo nekom vašem ukusu drugačiji izgled grada , igranje sa ambijentom i dizajnom okruženja je najzabavniji deo kreiranja igrica jer Vam dopušta da ispoljite svoje ideje , kreativnost i maštu. Na internetu možete pronaći hiljade različitih sprajtova uvezite ih u Unity kako smo radili do sada i pustite "mašti na volju".

5.2 Rad sa kamerom

Ukoliko pokušamo da se krećemo naš igrač će ići levo ili desno u zavisnosti gde mi želimo, što je super, ali uskoro ćemo naići na novi problem ,a to je da kada dođe do kraja ekrana heroj će nestati tj. otićiće van opsega naše glavne kamere nestaje tj. više se neće videti.Ovaj problem se rešava tako što ćemo morati kamerom da pratimo igrača dok se kreće kroz našu malu oazu.

Kreirajmo novu skriptu unutar ***Assets/Scripts/FollowCamera.cs***u koju ćemo smestiti kod za kretanje kamere.

Namestićemo da kamera ne prati igrača sve vreme već samo do trenutka kada dođe do ivice. Zato postavljamo Margine na udaljenost do koje igrač može da ide a da ga kamera ne prati kao i „tečna" usklađenost kako kamera prati igrača.

Na kraju treba da porverimo svaki frejm kako bi testirali da li je igrač dostigao ivicu kamere i na taj način ažurirali njenu poziciju. U novokreiranu skriptu dodajemo sledeći kod:

```
using UnityEngine;
public   class   FollowCamera : MonoBehavior
{
        //udaljenost po x i y osi do koje igrača ne prati kamera
        public   float xMargin = 1.5f;
        public   float yMargin = 1.5f;
        //koliko brzo kamera prati igrača po x i y osi
        public   float xSmooth = 1.5f;
        public   float ySmooth = 1.5f;
        //max i min coordinate koje kamera može da ima
        public   Vector2 maxXAndY;
        public   Vector2 minXAndY;
        //za coordinate igrača
        public   Transform player;
}
```

Sada kada smo kreirali osnovne promenljive koje ćemo koristiti trebamo dodati reference na objekat igrača u funkciju ***Awake()***koja se poziva pre ***Start()***funkcije tj pri instanciranju i može služiti kao konstruktor klase.

```
void   Awake()
{
        //setovanje reference
        player = GameObject.Find("Player").transform;
        if (player == null)
        {
                Debug.LogError("Player object not found");
        }
}
```

Sledeće što treba da dodamo jesu dve metode koje će proveravati da li je igrač došao da granica kamere max X i Y.

```
bool  CheckXMargin()
{
        // Vraća true ako je razlika između udaljenosti kamere i //igrača po X osi veća od xMargin
```

```
        return  Mathf.Abs(transform.position.x - player.position.x) > xMargin;
}

bool  CheckYMargin()
{
        // Vraća true ako je razlika između udaljenosti kamere i //igrača po Y osi veća od yMargin
        return  Mathf.Abs(transform.position.y - player.position.y) > yMargin;
}

        //vrednosti target x i y ne bi trebale da budu vece od max ili //manje od min
TargetX = Mathf.Clamp(TargetX,minXandY.x,maxXandY.x);
TargetY = Mathf.Clamp(TargetY, minXandY.y, maxXandY.y);

//setovanje kamere na poziciju target
transform.position=new Vector3(TargetX,TargetY,transform.position.z);
```

Sada na kraju skripte trebamo dodati metod koji će po svakom frejmu da proverava da li je igrač došao do granice kamere kako bi ažurirao poziciju kamere i pomerio je u skladu sa igračem.

Postoji dosta debata oko toga koju metodu za ažuriranje je najbolje koristiti unutar Unity igrice. Dole navedene metode su izvedene iz klase ***MonoBehaviour*** ***(osnovna klasa iz koje su druge skripte izvedene)***.

Dakle imamo 3 vrste ***Update()***metoda:

1. **Update()**
2. **FixedUpdate()**
3. **LateUpdate()**

FixedUpdate() se poziva na regularnoj osnovi kroz sam životni vek igrice i uglavnom se primenjuje kod vremenski osetljivih kodova ili za fiziku kao što je ovde slučaj.

Update() običan se pak poziva samo jednom i to na kraju svakog frejma koji je iscrtan na ekranu, s obzirom da vreme za iscrtavanje može da varira u zavisnosti od količine objekata .

LateUpdate() se nadovezuje na Update() , takođe se poziva jednom po frejmu ali tek nakon što se završe svi proračuni, fizika i ažuriranja.

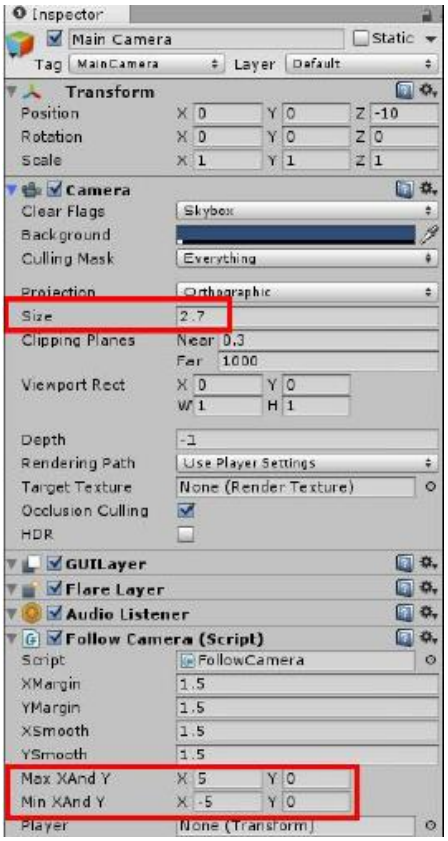
```
void   FixedUpdate()
{
        // trenutne coordinate kamere su trenutne coordinate igrača
        float targetX = transform.position.x;
        float targetY = transform.position.y;
        // ako se igrač pomerio izvan xMargine
        if (CheckXMargin())
```

```
        targetX = Mathf.Lerp(transform.position.x,
        player.position.x, xSmooth *
        Time.fixedDeltaTime );
        // ako se igrač pomerio izvan yMargine
        if (CheckYMargin())
        targetY = Mathf.Lerp(transform.position.y,
        player.position.y, ySmooth *
        Time. fixedDeltaTime );
        // x i y coordinate ne bi trebale da budu veće od Max ili //manje od minimum min
        targetX = Mathf.Clamp(targetX, minXAndY.x, maxXAndY.x);
        targetY = Mathf.Clamp(targetY, minXAndY.y, maxXAndY.y);
        // postavljanje pozicije kamere
        transform.position =
        new  Vector3(targetX, targetY, transform.position.z);
}
```

Zadatak za vežbu 2.

Namestite da kamera prati našeg heroja i ako skače… Ako ste se sami igrali iz radoznalosti i namestili igrača i da skakuće onda pokušajte i da mu dodelite poziciju kamere da ga prati uporedu sa skokom.

Nakon ovoga ostalo je samo da smanjimo veličinu kamere na veličinu visine našeg sveta i da postavimo parameter min i max kao na slici.



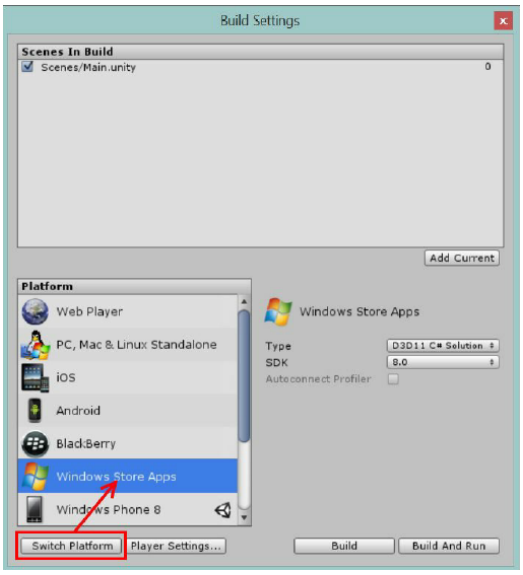
5.3. Rizici sa rezolucijama

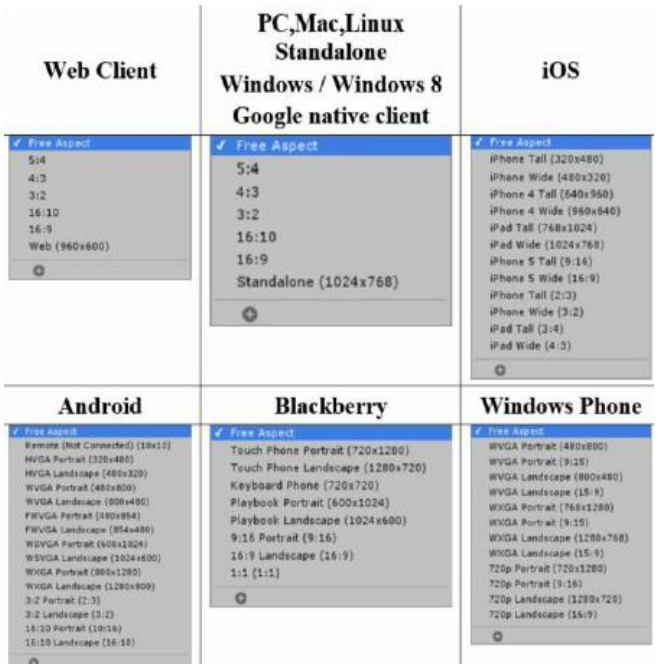
Kada radimo sa kamerama uvek će postojati problem sa rezolucijom na drugim platformama.

Po default-u u editoru Unity-a igrač će se kretati u ***Free Aspect*** modu. To može da se promeni da reprezentuje rezoluciju koja je podržana na bilo kom uređaju.

Da biste izmenili za koju platformu želite da pravite igricu ide se na ***File – Build Settings*** i zatim dugme ***Switch Platform,*** a da promenite apsekt gledanja na ***Game – Free Aspect.***

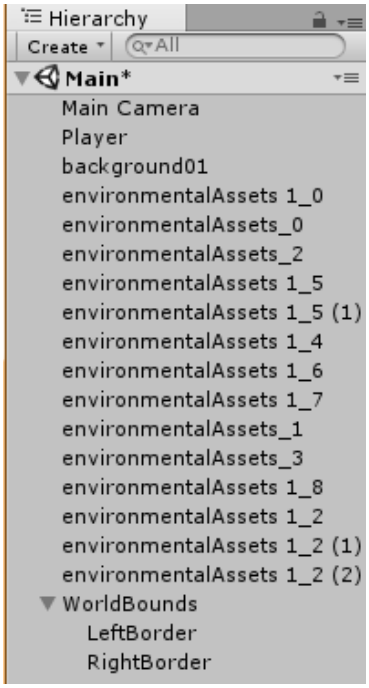
Ovde vidimo šta od mogućnosti imamo na raspolaganju ako odaberemo druge načine kreiranja igrica za određene platforme. Klikom na ***Maximaze on Play*** promene u aspect modu će se videti preko celog prozora Unitija.





5.4 Granice sveta

Iako naša kamera prati heroja ove priče, on i dalje kada stigne do granice sveta moćiće da ode izvan nje pa moramo i taj problem da rešimo!

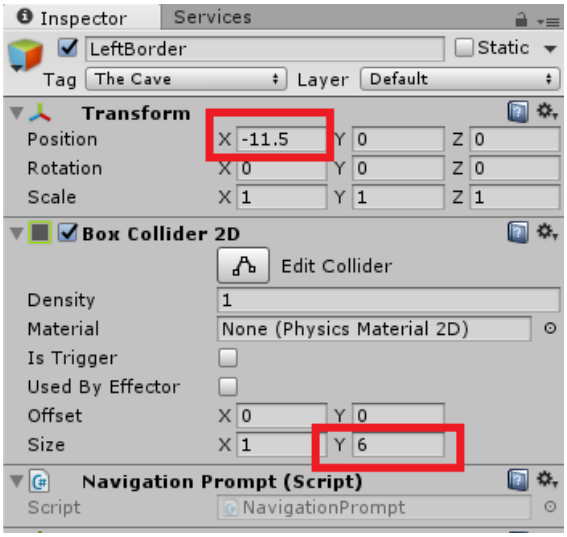


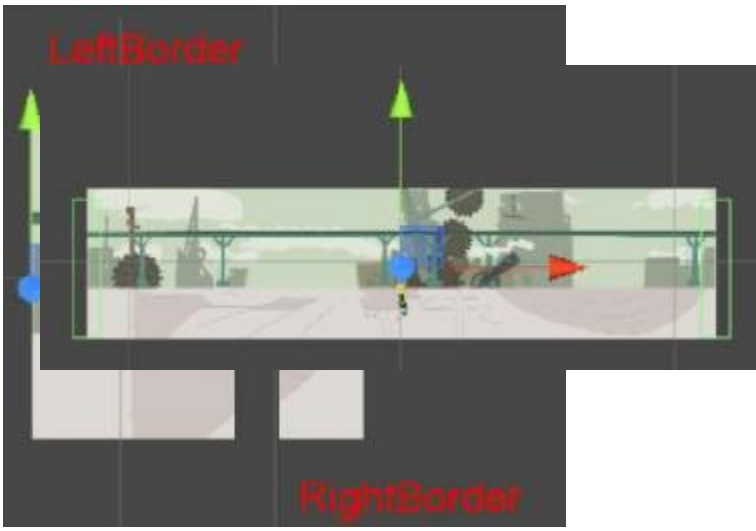
Pošto smo već dodali u poglavlju 1 kod kreiranja igrača ***BoxCollider2D*** i ***RigidBody2D*** sada ćemo njih iskoristiti i ovde.

Napravićemo 3 objekta (1 roditelj i 2 deteta) na **Create Empty**i nazvati ih kao na slici (na dnu)!

Dva objekta postavljamo sa leve i desne strane naše pozadine i setujemo im visinu kako bi osigurali da pokriva skoro celu visinu scene.

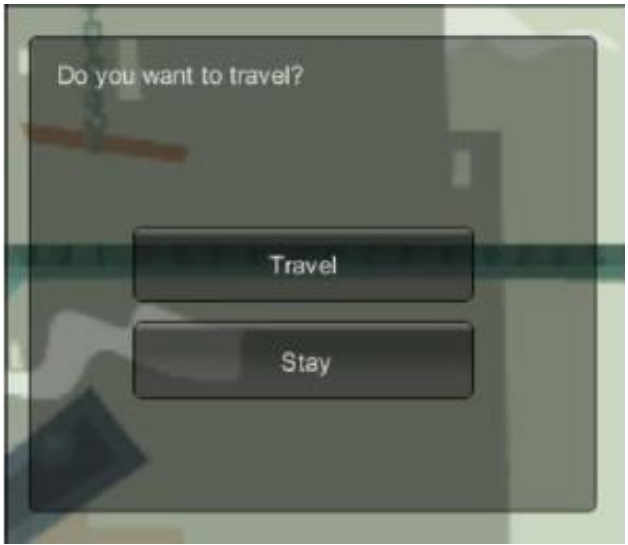
Alternativa je da se kreira jedan objekat i da se posle duplira. Sada naš heroj ne može napustiti granice njegovog sveta. Polje ***Size*** postavljamo u zavisnosti na naše koordinate sveta .





5.5. Putovanje nadalje

Kada smo postavili granice sa leve i desne strane želimo da omogućimo korisniku igrice da odlaskom na levu ili desnu stranu pri



„sudaru“ sa granicama grada odabere gde dalje želi da putuje!

Zato ćemo kreirati novu skriptu **NavigationPrompt.cs** koju ćemo kasnije dodeliti objektima ***LeftBorder***i ***RightBorder*** kako bi igrač pri interakciji sa nekim od ova dva (nevidljiva) objekta dobio GUI prozor kao na slici.

U novokreiranu skriptu dodaćemo sledeći kod:

```
using UnityEngine;

public  class  NavigationPrompt : MonoBehavior
{
        bool  showDialog;//prikazi pop up box za registrovanje //kolizije
        //pri sudaru sa granicom tj objektom iskočiće prozor
        void  OnCollisionEnter2D(Collision2D col)
        {
                showDialog = true;
        }
}

void  OnGUI()
{
        if (showDialog) //ako je pop up true onda ga trebamo //prikazati
{
//begin i end group za GUI se uvek setujex i y od kojih    //pocinje prozor , sirina i visina prozora
                GUI.BeginGroup(new Rect(Screen.width / 2 - 150, 50, 300, 250));
// slicno samo sa textom
                GUI.Box(new Rect(0,0,300,250) , "");
                GUI.Label(new Rect(15,10,300,68),"Do you want to travel?");
                if (GUI.Button(new Rect(55, 100, 180, 40), "Travel"))
                {
//ako je odabrano da se putuje prozor se zatvara i igrač se teleportuje
showDialog = false;
StartCoroutine(ChangeLevel(this.tag));
        }
if (GUI.Button(new Rect(55, 150, 180, 40), "Stay"))
        {
//ako pak zeli da ostane prozor pop up se samo zatvara
showDialog = false;
//Application.LoadLevel(1);
        }
GUI.EndGroup();
}
```

Sama funkcija je jako jednostavna:

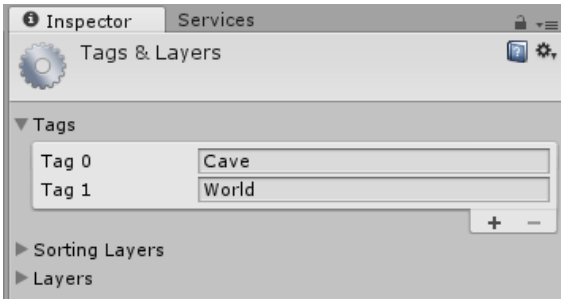
- OnGUI metoda setuje iskačući prozor za prikaz sa nekim tekstom i dva dugmeta
- Jedno dugme pita da li igrač želi da putuje , ako je odgovor potvrdan i klikom na njega treba da se učita mapa sveta gde će se izabrati dalji put, ali o tome u daljem poglavlju.
- Drugo dugme jednostavno zatvara prozor ukoliko odluči da ostane na istom nivou.

Sada prosto dodamo ovu skriptu dvema granicama tako što ih prevučemo na njih kao objekte i onda igrač kad god dođe u interakciju sa njima pojaviće se prozor koji smo kreirali.

5.6. Tagovanje

Kada smo kreirali prozor možemo da uvedemo pojam **tagovanja**.

U našem primeru možemo tagovati levu i densu granicu kako bi u zavisnosti koju igrač odabere prikazivalo konkretan tekst karakterističan za taj prozor.



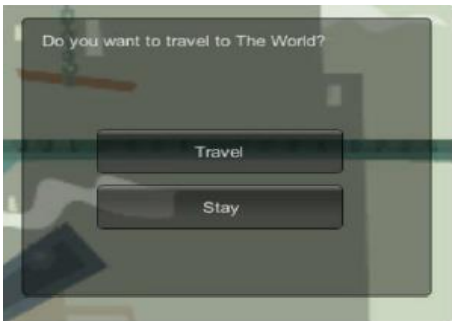
Da bismo napravili tag počećemo tako što odaberemo **Tag** polje kao na slici i u padajućem meniju Inspector prozora odaberemo postojeći ili kreiramo novi tag.

Imenovaćemo nove tagove „**Cave**“ i „**World**“. Prvi postavite na LeftBorder a World na desni.

Jedine tagove koje imamo pravo da menjamo su oni koje smo sami kreirali. Postoje tagovi koji su sistemski definisani. Njih ne možemo menjati ali možemo koristiti.

Sada ćemo ažurirati jedan red u kodu skripte **NavigationPrompt.cs:**

```
//informacija o ruti
GUI.Label(new  Rect(15, 10, 300, 68), "Do you want to travel to " + this.tag + "?");
```



Zadatak za vežbu 3.

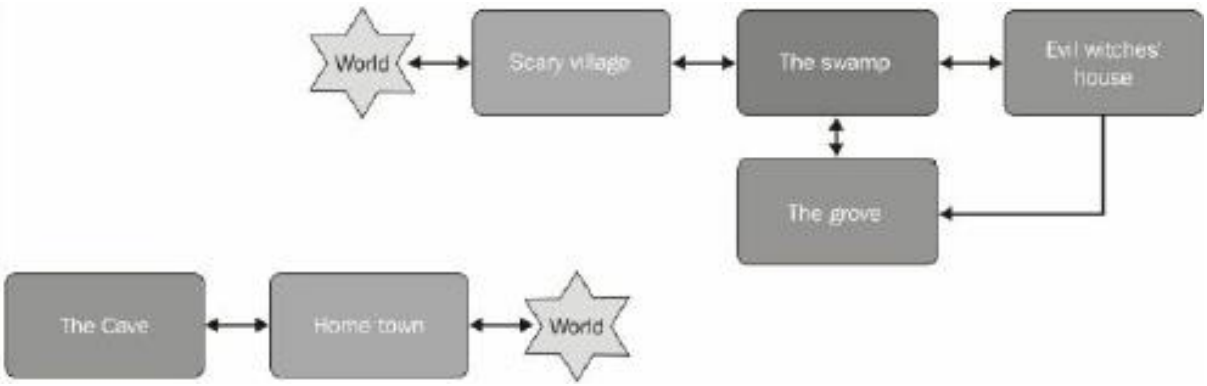
Promenite nazive tagova ili dodajte neke nove i vidite kako će onda reagovati na iskačući prozor.

5.7. Planiranje veće slike

Na slici ispod možemo da vidimo kako bi igrač putovao između delova sveta i da li može u oba smera ili samo u jednom pravcu.

Kreiraćemo skriptu ***Assets\Scripts/NavigationMenager*.cs** koja će vršiti sve prelaze između scena/svetova kao i informacije o svakoj destinaciji.

Na kraju ćemo ažurirati skriptu NavigationPromt kako bismo proverili da li igrač može da putuje.



Dodati sledeći kod u skriptu:

```
using UnityEngine;
using System.Collections;
usingSystem.Collections.Generic;
public  static class  NavigationMenager
{

// rečnik koji sadrži sve destinacije i njihove kratke opise
        public  static Dictionary<string , Route> RouteInformation = new Dictionary<string ,
Route>()
  {
                { "World", "The big bad world"},
                { "Cave", "The deep dark cave"},
  };

//prikazuje listu inventara igrača , detaljniji opis rute gde igrač putuje
public  static string GetRouteInfo(string  destination)
```

```
  {
 return  RouteInformation.ContainsKey(destination) ?
RouteInformation[destination].RouteDescription : null;
  }
// da li igračmože da putuje ili ne
public  static bool CanNavigate(string  destination)
  {
        return  true;
  }
//ukoliko može da putuje lodairamo mu novu scenu i saljemo ga teleportom
public  static void  NavigateTo(string  destination)
  {
//Debug.Log(destination);
    //Application.LoadLevel(destination);
  }
}
```

Kada smo uveli ovu skriptu sada možemo ažurirati metod za koliziju u skripti NavigationPrompt.cs:

```
void   OnCollisionEnter2D(Collision2D col)
{
        //Only allow the player to travel if allowed
        if (NavigationManager.CanNavigate(this.tag))
        {
                showDialog = true;
        }
}
```

Ponovo ćemo ažurirati skriptu samo metod OnGUI() kako bi nam sada preko menadžera prikazivao detaljniji opis rute i informacija za novu destinaciju.

```
GUI.Label(new  Rect(15, 10, 300, 68), "Do you want to travel to " +
NavigationManager.GetRouteInfo(this.tag) + "?");
```

Ako želi da putuje onda mu moramo dodeliti sledeći deo koda i ažurirati postojeći:

```
if (GUI.Button(new  Rect(55, 100, 180, 40), "Travel"))
{
        showDialog = false;
        NavigationManager.NavigateTo(this.tag);
}
```

5.8. Paralaxing

Paralaksa (odstupanje) je tremin za prividnu promenu položaja objekta u odnosu na pozadinu usled razlike u položaju posmatrača ili usled kretanja posmatrača velikim brzinama. Paralaksa je ugao između posmatranju jednog objekta iz dva različita položaja.

Bliži objekti imaju veću paralaksu od udaljenijih objekata. Na taj način je moguće koristiti paralaksu za merenje udaljenosti.

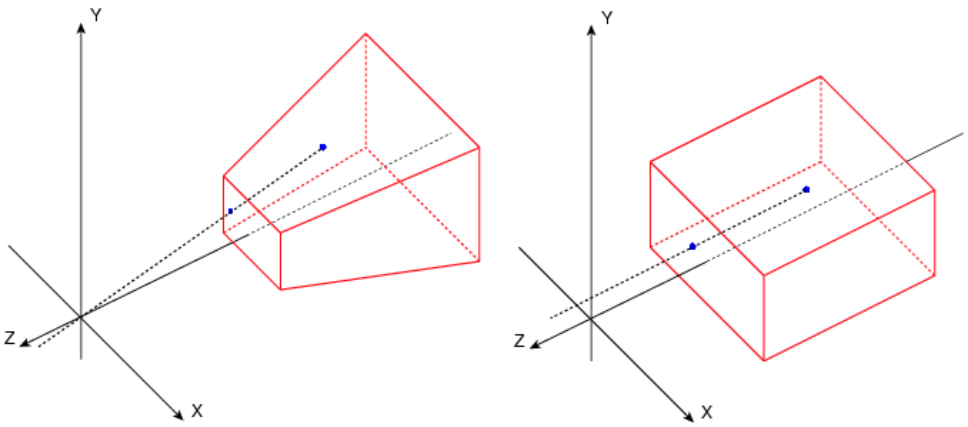
Da bismo videli na jednom prostom primeru primenu paralaksinga u igrici napravićemo dinamičan efekat kretanja oblaka iznad našeg grada.Najjednostavniji način je da koristimo odnos kamere tako što u **projection modu** promenimo način sa Ortographic na **Perspective.**

Ortografik kamera koristi ortografik veličinu koja dikitira u koliko jedinica u prostoru za kreiranje sveta je visina ekrana podeljena.

Razlika između ove dve kamere je što:

1. **Ortographic** kamera ima fiksiranu dubinu tj objekti se nalaze takvi kakvi su kreirani i kada se pokrene igrica, što nije slučaj u stvarnom svetu.
2. **Perspective** ima dubinu i objekti mogu različito da se kreću u zavisnosti od udaljenosti od kamere. Tako da je usko povezano sa paralaksingom i realističnije se omogućava kretanje objekata.

Drugi način je da napravimo preko jedne skripte , nazvaćemo je ***Parallax.cs***koja će upravljati objektima u pozadini .Kreiraćemo dva prazna objekta i njihovog roditelja:



- Backgrounds
    1. Near
    2. Far

Postupak je objašnjen u koracima ispod:

1. Zatim ćemo u njih prebaciti sprajtove koje želimo da se nađu bliže i dalje kameri.
2. Definisaćemo slojeve gde će koji objekat da se nalazi u odnosu na elemente grada.
3. Dodelićemo grupi objekata Near poziciju 8 a grupi Far 15 na primer.
4. Po završetku skriptu prevucite na **MainCameru**
5. Zatim objekte Near i Far prevucite redom u niz koji smo definisali nad skriptom u Inspector prozoru.
6. Postavite smooth na 1

Dodajte sledeći kod u skriptu:

```
using UnityEngine;
using System.Collections;

public  class  Parallaxing :MonoBehaviour
{
public  Transform[] Clouds; // niz pozicija objekata oblaka (sprajtova)
private float[] paralaxScales; //koliko svaki lejer treba da se pomeri u
//skladu sa kretanjem kamere i igrača , niz
public  float smooth; //  da se background kreće sporije ili brže
private Transform cam; // pozicija kamere
private Vector3 previousCamPos; // prethodna pozicija kamere

        // PROLAZIĆEMO KROZ SVAKI FREJM IGRE , PRI ČEMU ZNAMO POZICIJU TRENUTNU
//KAMERE i PRETHODNU POZICIJU , KOJA JE RAZLIKA i POMERIĆEMO POZADINU ZA TU RAZLIKU

        void   Start ()
        {
cam = Camera.main.transform; // naci će poziciju main kamere
```

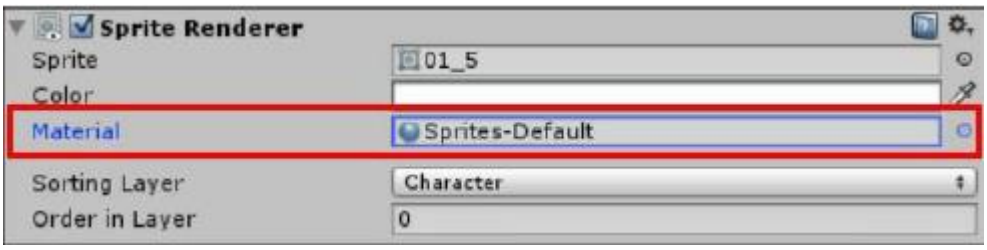
```
previousCamPos = cam.position; //trenutna pozicija postaje prethodna pozicija

paralaxScales = new float[Clouds.Length];//dužina niza oblaka
        for (int i = 0; i < Clouds.Length; i++)
                {
                paralaxScales[i] = Clouds[i].position.z * -1;
        }
        }

        // Paralax skript će biti pozvan posle kamere zato je lateUpdate
        void   LateUpdate ()
        {
for (int i = 0; i < Clouds.Length; i++)
     {
float parallax = (previousCamPos.x - cam.position.x)*paralaxScales[i]; //kolicina kretanja koje
zelimo da imamo
float backgroundTargetPosX = Clouds[i].position.x + parallax; //poyicija na koju zelimo da
pomerimo sloj , uzima poziciju objekta i
//dodaje kolicinu koju zelimo da pomerimo

Vector3  bacgroundTargetPos = new
Vector3(backgroundTargetPosX,Clouds[i].position.y,Clouds[i].position.z); //  kreiranje nove
pozicije
        Clouds[i].position = Vector3.Lerp(Clouds[i].position,bacgroundTargetPos,smooth *
Time.deltaTime); //  glatko će se pomerati duž scene
     }

previousCamPos = cam.position;//na kraju svake pozicije nasa prethodna pozicija bice jedna
trenutnoj
        }
}
```

Zadatak za vežbu 4.

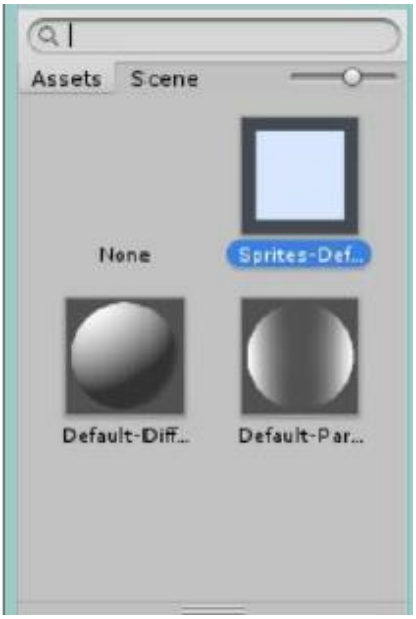
Na početku ovog poglavlja smo spomenuli dva načina za paralaksu, ja sam Vam pokazao na drugi način preko kreiranja skripte i dodeljivanja iste kameri a vi probajte sami da odradite na prvi način .

5.9. Pojam senčenja(Shaders) u 2D okruženju

Shaders (u daljem tekstu šejdersi) služe za definisanje niza osobina koje će se koristiti nad objektima i to utiče na ono što će krajnji rezultat prikazati na ekranu.Takav niz osobina kada se uskladišti naziva se ***materijal.***Veovali ili ne svi 2D objekati si iscrtani uz pomoć šejdersa.



Uglavnom se koriste kod 3D kodiranja ali i u 2D igricama se koriste prilično. Oni ne utiču na sam rad igrice već se koriste za realističnije detalje senke, osvetljenja i slično.Ako pogledate u Inspector prozoru našeg igrača videćete polje ***Material,*** koji poseduje ***Sprites-Default.***

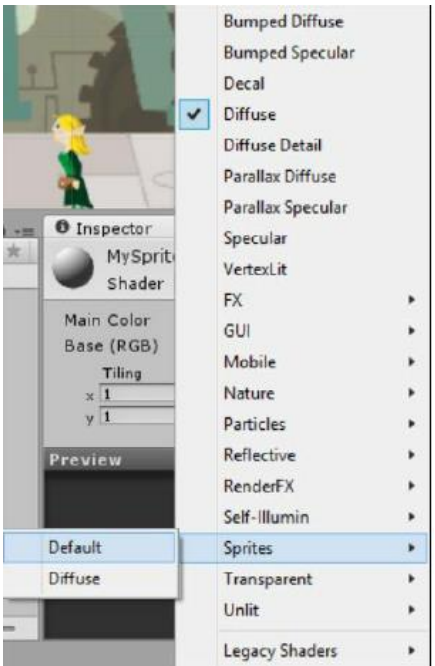


U našem konkretno primeru defoltni šejder je tipa Sprites-Default ali može biti i Diffuse.

1. **Default** – Šejder nasleđuje svoje teksture iz Sprite Renderera kako bi ga nacrtao kako treba. Ovo je osnova funkcionalnsot i sadrži sam svoje statično osvetljenje.

2. **Diffuse** – isti je kao i Default , nasleđuje teksture od Default-nog ali zahteva spoljašnje osvetljenje pošto svoje nema koje mora biti primenjeno odvojeno. Ima i neke dodatne funkcionalnosti u odnosu na Default-ni.

Materijalisu omoti koji sadrže vrednosti za osobine/svojstva objekata.Tako da različiti materijali mogu da sadrže iste šejdere ali sa drugim svojstvima.

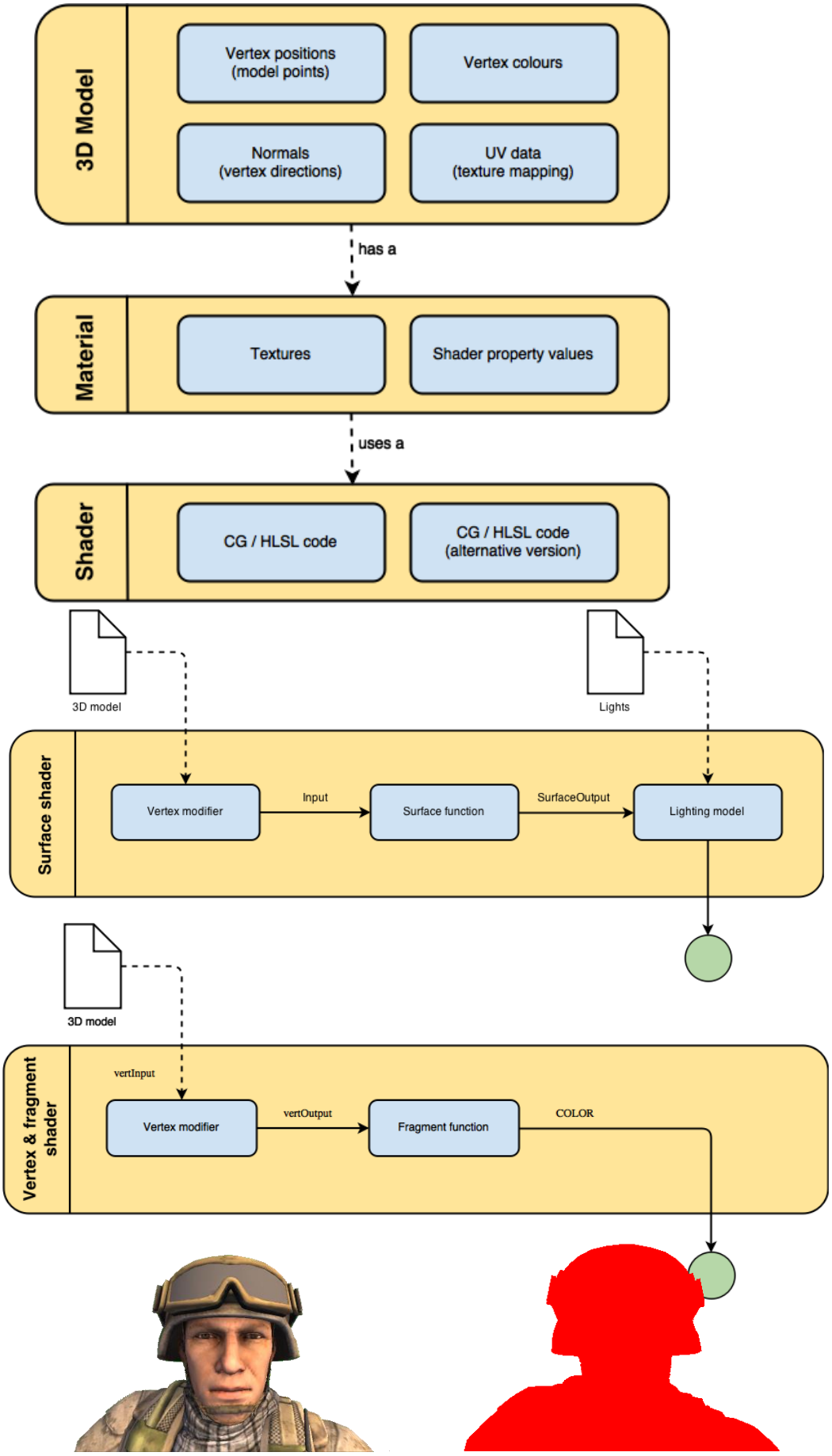


Mogu sadržasti SubShaders-e jedno ispod drugog. Oni čuvaju instrukcije za GPU a zatim će Unity da ih izvršava redom dok ne nađe onog koji je kompatibilan sa grafičkom karticom (npr za različite verzije platformi).

Postoje 3 tipa šejdera koje podržava Unity:

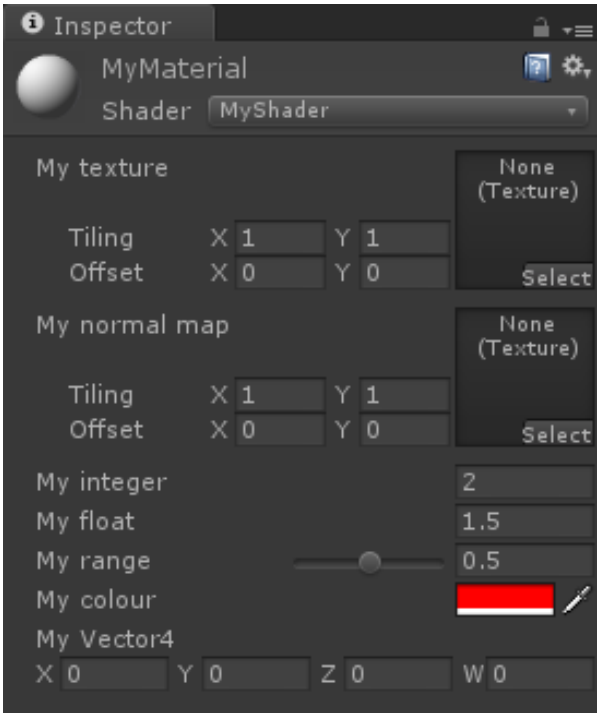
1. **Fragment shader** (teže se pišu,to je program koji koristi svaki piksel objekta, služi za računanje boje pojedinačnog piksela)
2. **Vertex shader** (program koji se pokreće na svakoj površini objekta kako bi GPU prikazala objekat na ekranu, teksture, pozicije i definisala temena objekta)
3. **Surface shader** (program koji se bavi osvetljenjem objekata, tipovima osvetljenja,njihovim opcijama i slično.)

```
Shader "MyShader"
{
    Properties
    {
        // The properties of your shaders
        // - textures
        // - colours
        // - parameters
        // ...
    }

    SubShader
    {
        // The code of your shaders
        // - surface shader
        //    OR
        // - vertex and fragment shader
        //    OR
        // - fixed function shader
    }
}
```


3D Model
Vertex positions (model points)
Vertex colours
Normals (vertex directions)
UV data (texture mapping)
has a
Material
Textures
Shader property values
uses a
Shader
CG / HLSL code
CG / HLSL code (alternative version)
3D model
Lights
Surface shader
Vertex modifier
Input
Surface function
SurfaceOutput
Lighting model
3D model
Vertex & fragment shader
vertInput
Vertex modifier
vertOutput
Fragment function
COLOR

Levo na slici je prikazan **surface** šejder dok na desnoj primer vertex/fragment šejdera.Šejdersi su uglavnom pisani u **Cg/HLSL**(High level shading language za DirectX) programskom jeziku.**Vertex i Fragment**šejdersi nemaju ugrađen koncept za manipulaciju osvetljenjem dok Surface imaju jer je njima to glavni zadatak.

Bitna razlika između vertex i fragment šejdera je da vertex može da manipuliše atributima temena objekta dok fregment se bavi pojedinačnim pikselima između temena i kalkuliše način njihovog bojenja.Zato se oni uglavnom koriste kod nerealističnih modela.



Definisana svojstva u Properties delu nisu dovoljna da bismo mogli da ih koristimo ,preko njih Unity nam daje pristup u Inspector prozoru promenljivama iz šejdera. Ove promenljive i dalje moraju biti definisane u glavnom telu šejderakoji se nalazi u Subshader sekciji.

```
SubShader
{
    // Code of the shader
    // ...
    sampler2D _MyTexture;
    sampler2D _MyNormalMap;

    int _MyInt;
    float _MyFloat;
    float _MyRange;
    half4 _MyColor;
    float4 _MyVector;

    // Code of the shader
    // ...
}
```

```
Properties
{
    _MyTexture ("My texture", 2D) = "white" {}
    _MyNormalMap ("My normal map", 2D) = "bump" {}  // Grey

    _MyInt ("My integer", Int) = 2
    _MyFloat ("My float", Float) = 1.5
    _MyRange ("My range", Range(0.0, 1.0)) = 0.5

    _MyColor ("My colour", Color) = (1, 0, 0, 1)    // (R, G, B, A)
    _MyVector ("My Vector4", Vector) = (0, 0, 0, 0) // (x, y, z, w)
}
```

Poglavlje 6. NPCs

Svet bi bio veoma usamljeno mesto kada bismo bili sami. Tako da ćemo se u okviru ovog poglavalja postarati da naš lik u igri ne bude više usamljen. Ali za početak moramo naučiti neke osnove o kumunikaciji objekata u igrama.

6.1. Singltoni i menadžeri

Svaki veliki i kompleksan projekat u kome se od početka ne vodi racučna o dizajnu može naleteti na probleme u uvođenju i brisanju objekata sa scene.

Ovaj problem se može rešiti korišćenjem jednog od dva obrasca : Menadžeri i singltoni.

Postoje 2 načina na koje možete implementirati singlton obrazac

1. Korišćenjem javne statičke klase za upravljanje glavnom klasom, što omogućava drugim klasama da joj pristupe iz bilo kog dela igre. Veoma je korisno ako želite da neki događaj izazove menadžera da uradi nešto, kao naprimer u konverzaciji.
2. Možete koristiti i prazan objekat u igri za koji je prikačen singlton obrazac, ali ovo može izazvati konflikte ako se javi više istih obrazaca.

Menadžeri su centralne skripte jedinstvene za svaku scenu, i kontrolišu i održavaju tok scene za jedan, ili više objekata.

6.1.1 Singltoni

U slučaju menadžera mora da se kotroliše gde je postavljena svaka instanca, i nije moguća interakcija sa njim bez više podešavanja. Onda moramo spajati menadžer sa drugim objektima. Najbolji način je implementacija singltona u medadžeru.

Definisanje singltona u menadžeru možete videti u nastavku (na sledećoj strani) :

```
        //Statička singlton vrednost
public  static  MySingletonManager Instance
   {
get; private set;
   }
//Javna vrednsot za menadžer
public  string  MyTestProperty = "Hello World";
void  Awake()
   {
//Čuvanje trenutne instance
      Instance = this;
   }
//Javna metoda menadžera
public  void  DoSomethingAwesome()
   { }
```

Onda možemo pristupiti vrednostima i metodama singltona jednostavnim pozivom njegovih metoda iz bilo kog dela projekta

```
//Set the public   property of the singleton
MySingletonManager.Instance.MyTestProperty = "World Hello";
//Run the public   method from the singleton
MySingletonManager.Instance.DoSomethingAwesome();
```

6.2. Komunikacija između objekata

U svakoj igri postoji planirana iterakcija između komponenti igre. Interakcije mogu biti:

- Test fizičke kolozije
- Reakcija na pucanje
- Otvaranje i zatvaranje vrata
- Trigeri, svičevi ili zamke
- Komunikacija između dva igrača

Postoji nekolino načina na koji ovo može da se izvrši, zavisno od onoga što nam treba možemo koristiti:

- Delagate
- Događaje
- Poruke

6.2.1. Delegati

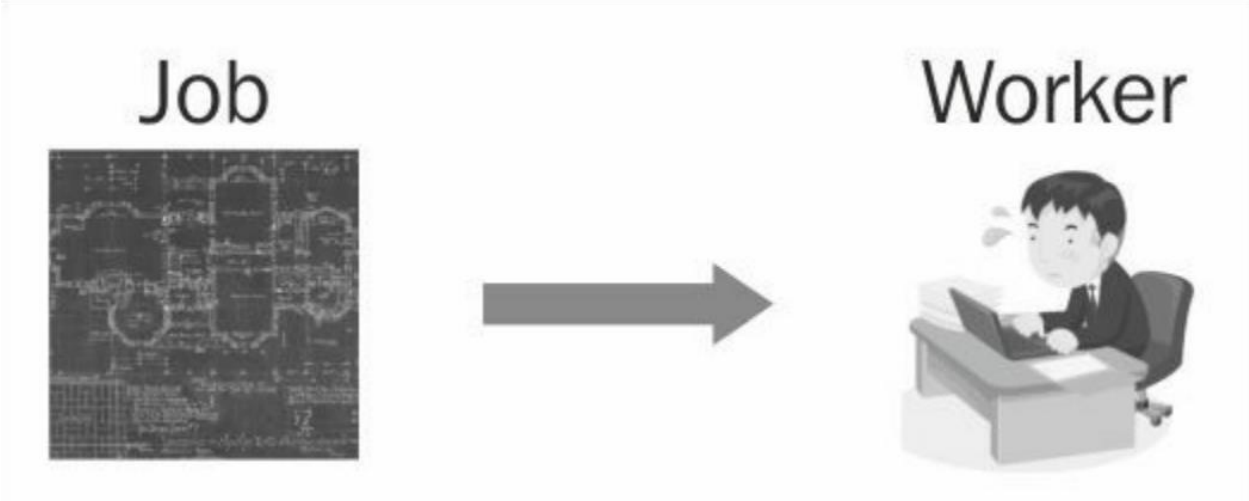
Delegate susrećemo redovno u svakodnevnom životu. Delagati su metode koje prihvataju delove posla i obavljaju umestu nekoga drugog. U C# imamo Action i Action<T> metode kao primer delegata.

Postoje dva obrasca koja se koriste za delegate

- Obrazac konfigurabilnih metoda
- Delegacioni obrazac

6.2.1.1. Obrazac konfigurabilnih metoda

Obrazac konfigurabilnih metoda imamo kada se deo posla, ili funckija prosleđuje drugoj metodi kako bi se zadatak odradio.



```
//Definisemo delegat
delegate void  RobotAction();
//vrednost delegata
RobotAction myRobotAction;
void  Start()
   {
//Pordrazumevana vrednsot delegata
       myRobotAction = RobotWalk;
   }



void  Update()
   {
//Pokreni metodu delegata
myRobotAction();
   }
//Metoda koja kaze robotu da ide
public  void  DoRobotWalk()
   {
//postavljanje delegata
       myRobotAction = RobotWalk;
   }
void  RobotWalk()
   {
Debug.Log("Robot walking");
   }
```

6.2.1.1 Delegacioni obrazac

Delegacioni obrazac se koristi kada metoda poziva pomoćnu biblioteku da odradi zadatak, i kad odradi zadatak vraća se u glavnu funkciju. Primer download sa weba, kada se download završi mi radimo nešto sa tim što smo skinuli.


```
public  class   Worker
  {
List<string > WorkCompletedfor = new List<string >();
public  void  DoSomething(
string  ManagerName,
      Action myDelegate )
      {
        WorkCompletedfor.Add(ManagerName);
//Pocinje posao
myDelegate ();
      }
  }
public  class  Manager
  {
private Worker myWorker = new Worker();
public  void   PeiceOfWork1()
      {
//radi nesto
      }
public  void   PeiceOfWork2()
```

```
        {
//radi nesto 2
        }
public  void  DoWork()
        {
 / / Posalji radniku posao 1
          myWorker.DoSomething("Manager1", PeiceOfWork1);
//Posalji radniku posao 2
           myWorker.DoSomething("Manager1", PeiceOfWork2);
        }}
```

6.2.2. Poruke

Komunikacija je ključan faktor u igrama.Unity poseduje funckije za slanje poruka, kao što su SendMessage i BrodcastMessage. Obe funckije primaju ime metode koju će pozvati prilikom kolizije.

```
void  OnCollisionEnter(Collision col)
   {
col.gameObject.SendMessage("IHitYou", SendMessageOptions.RequireReceiver);
   }
void  OnCollisionEnter(Collision col)
   {
col.gameObject.BroadcastMessage("IHitYou");
   }
```

Obe funckcije su veoma spore, jer u svaki put prvo pokusavaju da nađu da li objekat posedujetraženu metodu.Kako bismo razbili zavisnost između objekata i potrebu za čuvanje referenci, ili potrebe za pronalaženjem objekata, koristi se posrednik, a to je menadžerska klasa.

Kreiramo menadžersku klasu sa kojom možemo voditi listu objekata koji žele da prime poruku, i pružaju jednostavan način da obaveste bilo koga ko sluša. Kako bismo ovo implementirali koristićemo singltone.Menadžersku klasu kačimo na bilo koji prazan objekat.

Prvo postavljamo singlton instancu:

```
public  static MesssagingManager Instance { get; private set; }
```

Zatim listu delegata:

```
private List<Action> subscribers = new List<Action>();
```

Zatim u okviru Awake() metode inicijalizujemo singlton, uz proveru ako postoji već instanca ona se uništava.

```
void  Awake()
  {
Debug.Log(„Menadžer poruka započet");
//Proveravamo da li postoje druge instance
if (Instance != null&& Instance != this)
      {
    / / Unistavamo druge isntance ako postoje
    Destroy(gameObject);
      }

//Cuvanje trenuitnog singltona
      Instance = this;
// Opcionanlo, spreciti unistavanje trenutne instance prilikom ucitavanja
      DontDestroyOnLoad(gameObject);
  }
```

Metoda Subcribe koja dodaje prosleđeni delegat menadžeru

```
public  void  Subscribe(Action subscriber)
  {
Debug.Log(„Pretplatnik registrovan");

subscribers.Add(subscriber);
        }
```

I metoda Brodcast koja obavesteva pretplatnike da se nešto desilo.

```
public  void  Broadcast()
  {
Debug.Log(„Emitovanje zatrazeno, Broj pretplatnika = " + subscribers.Count);
foreach (var subscriber in subscribers)
      {
subscriber();
      }
  }
```

Kao agent za emitovanje poruke kreiramo skriptu MessagingClientBroadcast

```
public  class   MessagingClientBroadcast :MonoBehaviour
  {
void   OnCollisionEnter2D(Collision2D col)
     {
MessagingManager.Instance.Broadcast();
     }  }
```

Koja će reći menadžeru da emituje poruku prilikom kolizije.Reciver sada možemo dodati igraču.

Pošto niko ne sluša poruku koja se emituje moramo dodati prijemnik (Reciver)

```
public  class   MessagingClientReceiver :MonoBehaviour
  {
void  Start()
     {
MessagingManager.Instance.Subscribe(ThePlayerIsTryingToLeave);
     }
void  ThePlayerIsTryingToLeave()
     {
Debug.Log("Oi Don't Leave me!! - " + tag.ToString ());
     }
  }
```

Prilikom starta ova skripta se registruje kao pretplatnik menadžeru i daje mu do znanja da kada se događaj desi pokreni njenu metodu

ThePlayerIsTryingToLeave()

Ovu skriptu možemo dodati na jednu od granica kako bismo je testirali.

- Za početak ćemo uvesti sprite sa čarobnjakom(04 sprite) i dodati mu RigiBody2D
- Po želji radi dramatičnosti uraditi animciju od sprajtova 04_3 i 04_5

6.3. Serijalizacija

- Serijalizacija je proces prevođenja objekta u sekvencu bitova kako bi objekat mogao biti sacuvan u datoteci, memorijskom baferu ili preneti preko mreže kako bi mogao kasnije da se vrati u prvobitni oblik.

- Serijalizacija je korisna kako bi se učitavali i snimali nivoi, razgovori i slično.
- Najbolji način kako bismo to postigli u unitiu je nasleđivanje objekata ScriptableObject
- ScriptableObject nam omogućava da snimimo podatke u okviru klase koja se koristi za .assset file u projektu
- Kako bi smo postigli serijalizaciju trebamo napraviti skriptu sa svojstvima koje želimo da serijalizujemo
- Klase se moraju tagovati atributom [System.Serializable] kao bi unty znao da služe za serijalizaciju
- Nakon kreiranja serijalizovane klase desnim klikom na Asset folder moćemo naći opciju koju smo kreirali.

6.3.1 Kreiranje serijalizovanog razgovora

- Prvo kreiramo C# skriptu ConversationEntry koja nam pruža osnovne informacije za naš sistem za konverzaciju, kao što su ime lika, slika i najbitinije text. Klasa moramo tagaovatio sa [System.Serializable] kako bi unity znao sta da radim s njim.

```
        [System.Serializable]
public  class    ConversationEntry
   {
public  string  SpeakingCharacterName;
public  string  ConversationText;
public  Sprite DisplayPic;
   }
```

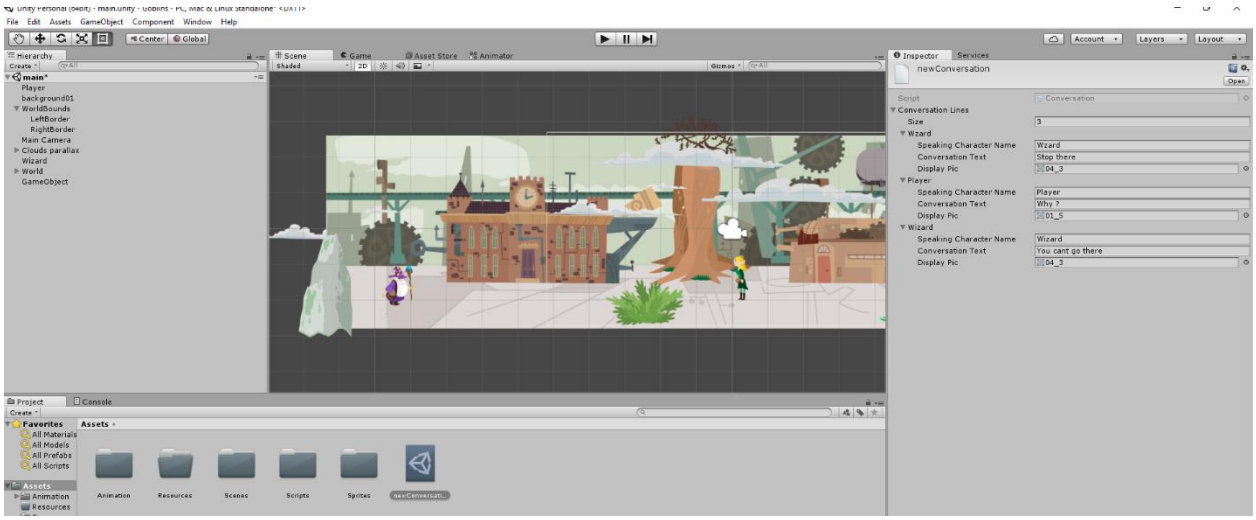
- Pošto ćemo imati više linija razgovora trebaće na objekat koji će podržati više linija razgovora .
- Kreiramo C# skriptu Conversation koja je izvedena iz ScriptableObject klase koja nam omogućava da koristimo metode serijalizacije u unity-u i pruža da uzimamo po linjiju texta iz konverzacije.

```
public  class    Conversation :ScriptableObject
        {
        public  ConversationEntry[] ConversationLines;
    }
```

Kako bismo ubacili novu opciju za razgovor u okviru dela Asset kreiramo skriptu ConversationAssetCreator .

```
public  class   ConversationAssetCreator :MonoBehaviour
   {
//Definiše opciju u editoru koja će kreirati novu opciju
      [MenuItem("Assets/Create/Conversation")]
public  static void  CreateAsset()
      {
//Kreira novu instancu ScribtibleObject
Conversation ConversationAssetCreator = ScriptableObject.CreateInstance<Conversation>();
//Kreira .asset fajl za naš novi objekat i čuva ga
         AssetDatabase.CreateAsset(ConversationAssetCreator, "Assets/new
Conversation.asset");
         AssetDatabase.SaveAssets();
//Prebacuje inspektor na naš novi objekat
         EditorUtility.FocusProjectWindow();
         Selection.activeObject = ConversationAssetCreator;
      }
   }
```

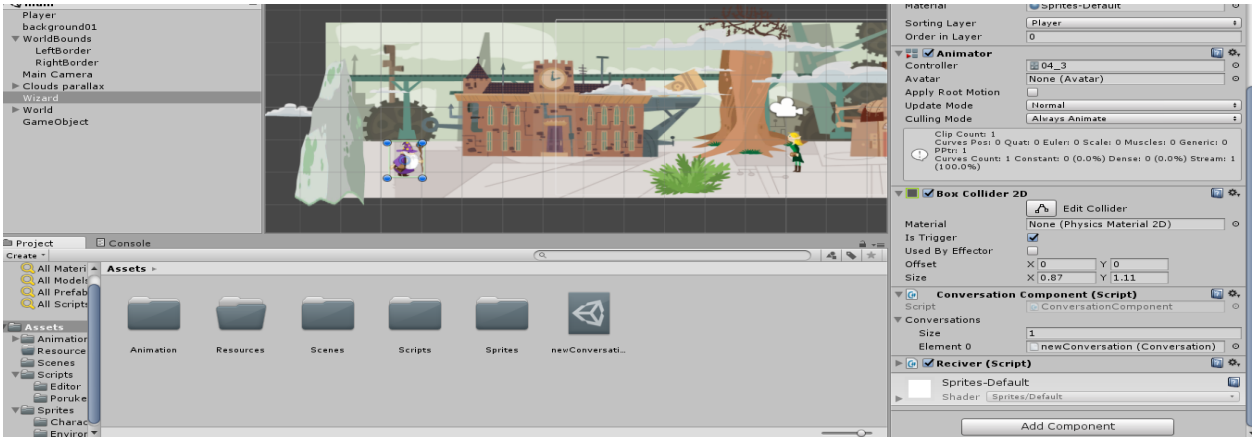
Sada možemo u okvru foldera Asset -> Resources desnim klikom kreirati novu opciju Conversation. i dodati neku konverzaciju.



Sledeći korak je kreiranje skripte ConversationComponent kako bi mogli zakačiti konverzaciju za neki objekta u igri. Na instancu ove skripte kačimo prethodno kreirano konverzaciju.

```
public  class    ConversationComponent :MonoBehaviour
  {
public  Conversation[] Conversations;
  }
```

Ovu skriptu možemo zakačiti na čarobnjaka, zajedno sa skriptom prijemnika.



Sledeće nam je potreban mehanizam za prikazivanje konverzacije.Za ovo kreiramo menadžer ConversationManager koji ći prihvatiti konverzaciju i prikazati je na ekranu

```
public  class    ConversationManager :Singleton<ConversationManager>
  {
//Garantuje nam da će biti samo singlton
protectedConversationManager() { }
  }
```

Zatim kreiramo korutinu(IEnumerator) DisplayConversation koji će uzeti svaku liniju konverzacije kroz petlju.Prvo proveravamo da li je konverzacija započeta i na kraju postavljamo da je konverzacija završena

```
IEnumerator DisplayConversation(Conversation conversation)
  {

    talking = true;
foreach (var conversationLine inconversation.ConversationLines)
    {

      currentConversationLine = conversationLine;
```

```
        conversationTextWidth = currentConversationLine.ConversationText.Length *
        fontSpacing;
yield return new WaitForSeconds(3);
    }
    talking = false;
  }
```

Za svaku liniju konverzacije postavlajamo pokazaivač na trenutni element linije currentConversationLine. Uzimamo koliko je veliki tekst zbog crtanja na ekranu. i čekamo 3 sekunde između svako dela konverzacije

Priliko starta proveramo ako nema trenutno započete konverzacije onda počinjemo prethodno definisanu korutinu kojo prosleđujemo konverzaciju.

```
public  void  StartConversation(Conversation conversation)
  {
if(!talking)
    {
      StartCoroutine(DisplayConversation(conversation));
    }
  }
```

Ostaje nam samo da iscrtamo konverzaciju na ekranu:

```
GUI.BeginGroup(new Rect(Screen.width / 2 - conversationTextWidth / 2, 50,
conversationTextWidth + displayTextureOffset + 10, dialogHeight));

// pozadinska kutija
GUI.Box(new Rect(0, 0, conversationTextWidth + displayTextureOffset + 10, dialogHeight), "");

//ime sagovornika
GUI.Label(new Rect(displayTextureOffset, 10, conversationTextWidth + 30, 20),
currentConversationLine.SpeakingCharacterName);

// tekst
GUI.Label(new Rect(displayTextureOffset, 30, conversationTextWidth + 30, 20),
currentConversationLine.ConversationText);

//kraj
GUI.EndGroup();
```

Još moramo dodati malo koda u skriptu prijemnika poruka u okviru metode ThePlayerIsTryingToLeave(). Koja će sada proslediti konverzaciju singltonu ConversationManager.

```
var dialog = GetComponent<ConversationComponent>();
if (dialog != null)
      {
if (dialog.Conversations != null&& dialog.Conversations.Length > 0)
         {
var conversation = dialog.Conversations[0];
if (conversation != null)
            {
ConversationManager.Instance.StartConversation(conversation);
            }
         }
      }
```

Zadatak za vezbu

1. Dodati novu konverzaciju, prikačiti je na novi objekat i omogućiti u MesssagingManager skripti da se prilikom Subscribe metode čuva i tag objekta koji se prijavio kako bi pustili vise konverzacija

7. Veliki svet

Veoma je važno da igrač ima osećaj velikog prostora u okviru igre. U okviru ovog dela ćemo omogućiti da naš igrač može da izabere preko mape druge mesta koje može posetiti.

7.1. Tipovi mapa

Postoje dva glavna pristupa u radu sa mapama

Fixne mape - U ovom slučaju slike su nacrate od strane umetnika i imaju veliki nivo detalja o svetu koji okružuje igrača, i postepeno otrkivaju durge delove kako igrač istražuje.

Generisane mape - U ovoj opciji se na sreću bira mesto za odlazak i događaji koji će se desiti.

7.2. Prostor ekrana i prostor sveta

Kada radimo sa inputom putem miša, ili dodira važno je voditi računa da su koordinate sa kojima Unity radi u prosotru ekrana.

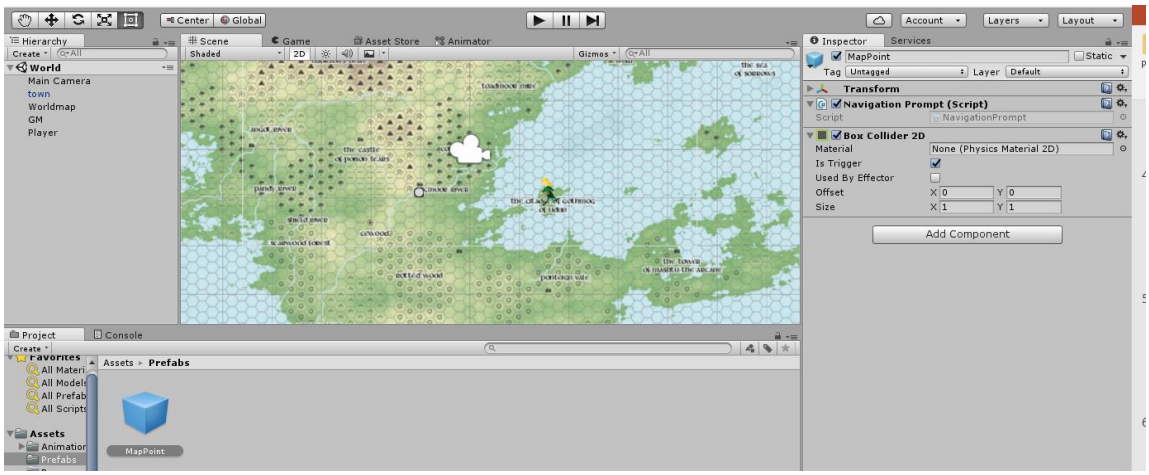
- Prosotor ekrana se odnosi na koordiante relativne u odnosu na ekran, sa početkom u gornjem levom uglu ekrana.
- Prostor sveta se odnosi na koordiante korišćene u okviru uniti-a.

Kao bismo proverili trenutne koordiante miša možemo koristiti Input.mousePosition. Za dodir možemo koristiti Input.GetTouch(<touch index>).position. S obizor da su dobijen koordinate u prostoru ekrana moramo ih prevesti u prostor igre što se može lako uraditi meotdom Camera.main.ScreenToWorldPoint(<screenCoordinate>

7.3. Kreiranje mape u igri

Sada kada smo naučili osnove o mapa možemo kreitrati našu mapu.

- Prvi korak je kreiranje nove scene u koju ćemo ubaciti mapu
- Na scenu dodajemo:
    - Sliku koja predstavlja našu mapu.
    - Sprite igrača
    - I prazan objekat MapPoint koji će predstavljati neki grad na mapi, od koga pravimo prefab kako bi mogli kreirati više mesta koje ćemo posetiti.



U NavigationManager skripti je potrebno ubaciti novu rutu .Potrebno je prikačiti skriptu NavigationPrompt na objekat na mapi.

Kreiramo klasu sa produženim metodama koje nam omogućavaju da uzmemo realne koordinate gde je kliknuto na mapi

```
// Konvertuje Vektor3 u 2D Vektor3
public  static Vector3 ToVector3_2D(thisVector3 coordinate)
   {
 return new Vector3(coordinate.x, coordinate.y, 0);
   }


// Konvertuje koordinate prosotra ekrana u Vektor3 sa vrerdnostima za 2D
public  static Vector3 GetScreenPositionFor2D(thisVector3 screenCoordinate)
   {
Vector3 wp = Camera.main.ScreenToWorldPoint(screenCoordinate);
       return  wp.ToVector3_2D();
   }
```

This označava da je ovo produžena metoda i omogućava da se pozove normlano:

```
var clickPoint = WorldExtensions.GetScreenPositionFor2D(Input.mousePosition);
```

7.4. Kretanje igrača po mapi

Kako bi omogućili igraču da se kreće po mapi kreiramo skriptu MapMovment. U kojoj proveramo uz pomoć metode Input.GetMouseButtonUp(0) da li je kliknuto dugme na mišu, i uzimao koordinate gde je klinuto uz pomoć metode Input.mousePosition

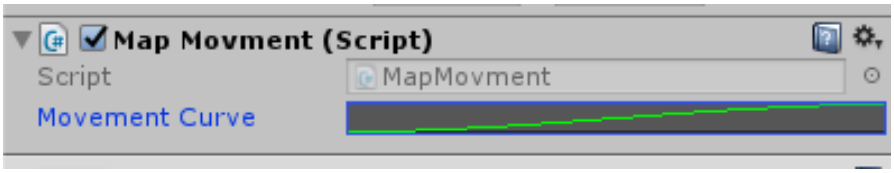
```
if (Input.GetMouseButtonUp(0))
    {
   StartLocation = transform.position.ToVector3_2D();
   timer = 0;
   TargetLocation = WorldExtension.GetScreenPositionFor2D(Input.mousePosition);
   Ready = true;
   }
```

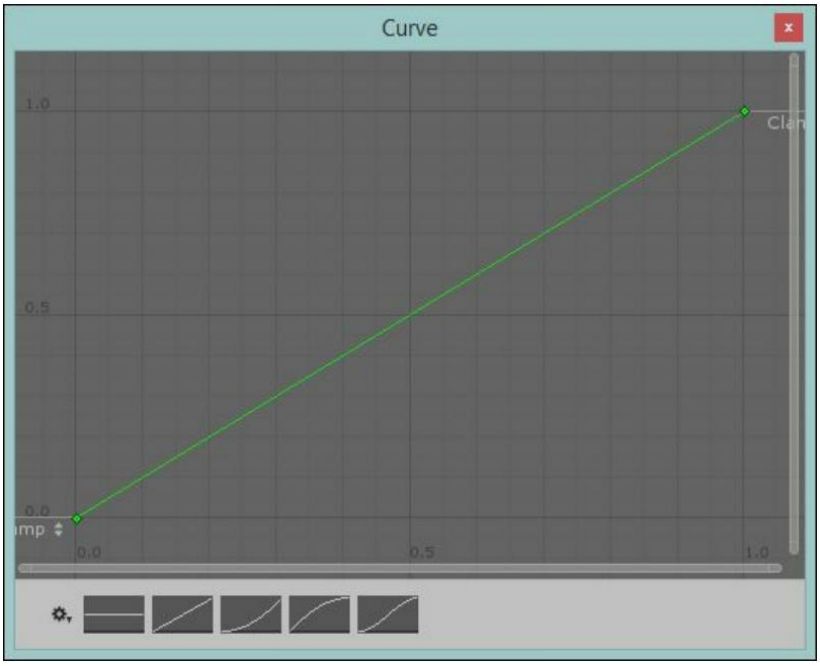
I jendostavno pomeramo igrača uz pomoć

```
if (TargetLocation != Vector3.zero && TargetLocation != transform.position && TargetLocation
!= StartLocation &&  Ready)
   {
transform.position = Vector3.Lerp(StartLocation, TargetLocation,
          timer);
 timer += Time.deltaTime;     }
```

Animacione krive mogu promeniti kako se sprite menja tokom vremena. Tako da ćemo ih ubaciti u naš MapMovement za kretanje igrča na mapi:

public AnimationCurve MovementCurve;

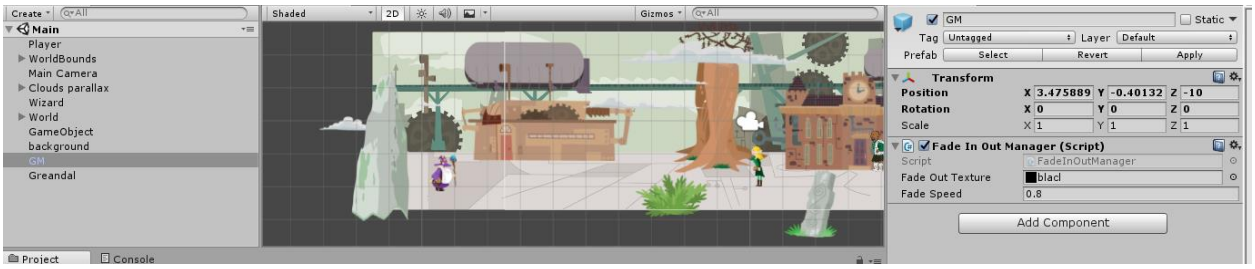




7.5. Postepena tranizcija između nivoa

Kako bi omogućili opstepenu promenu scena potrbna nam je skripta u obliku menadžera koja će voditi računa o promeni scena.

- Prvo postavljamo prazan objekat „GM“ za koji kačimo skriptu „FadeInOut“ koja nam omogućava postepenu traniziciju između scena. Ova skripta mora implementirati singleton.
- Metoda OnGUI koja iscrtava teksturu koja će biti prikazan između scena (Tekstura može biti samo crna pozdina)



Kako bi mogli koristit ovo na svakoj sceni prvo postavljamo singlton instancu:

```
public  static  FadeInOutManager Instance { get; private set; }
```

Zatim u okviru Awake() metode inicijalizujemo singlton, uz proveru ako postoji već instanca ona se uništava.

```
void  Awake()
   {
//Proveravamo da li postoje druge instance
if (Instance != null&& Instance != this)
      {
    / / Uništavamo druge isntance ako postoje
       Destroy(gameObject);
      }
//Cuvanje trenuitnog singltona
      Instance = this;
   }
```

Sledeće je potrebno korišćenje metode OnGUI() kako bi iscrtali tranziciju.

```
void  OnGUI()
   {
// fade na osnovu smera,brzine i brzine promene frejma
      alpha += fadeDir * fadeSpeed * Time.deltaTime;

// setovanje alfa vrednosti na silu između 0 i 1
      alpha = Mathf.Clamp01(alpha);

// podesava boju GUI-a (u nasem slucaju teksture) sve vrednosti ostaju iste samo se alfa setuje
na alfa vrednost
GUI.color = new Color(GUI.color.r, GUI.color.g, GUI.color.b, alpha);

// crdna tekstura na vrhu
GUI.depth = drawDepth;

// postaviti tekstuiru preko celog ekrana
GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height), fadeOutTexture);
// poziva se nakon ucitavanja nove scene
void  OnLevelWasLoaded(int level)
   {
// poziva metodu koja zapocinje traniziciju
BeginFade(-1);
      }
// postavlja smer tranizicije između scena
public  floatBeginFade(int direction)
   {
      fadeDir = direction;
 return  fadeSpeed;
   }
```

Zatim je potrebno postaviti korutinu koja će se aktivirati prlikom promene scena i izvršiti traniziciju

```
IEnumerator ChangeLevel()
   {
// pronalzi objekat za koji je zakacena tekstura i skripta za tranziciju između scena
float fadeTime = GameObject.Find("GM").GetComponent<FadeInOutManager>().BeginFade(1);
yield return new WaitForSeconds(fadeTime);
// promena scene
UnityEngine.SceneManagement.SceneManager.LoadScene(level);
   }
```

Korutina će biti pokrenuta pozivom metode Fading kad god je potrebno promeniti scenu.

```
public  void  Fading(string  location)
  {
     StartCoroutine(ChangeLevel(location));
  }
```

I poziva je NavigationManager u metodi NavigateTo:

FadeInOutManager.Instance.Fading(destination);

Zadatak za vežbu

1. Postaviti neku fixnu mapu, i tačku na mapi koja se može posetiti.

Poglavlje 8. Neprijatelji

8.1 Kreiranje protivnika i scene za borbu

- Napraviti scenu **Battle** (File->New Scene), dodati pozadinu **background02.png** na nju. Dodati još neke sličice na pozadinu, po izboru, voditi računa o layerima.

- Podeliti sličicu goblina 05.png na sprajtove pomoću **Sprite Editor-a** (označiti sliku pa u Inspectoru prvo označiti SpriteMode: Multiple pa kliknuti na Sprite Editor)**,** prevući dobijeni sprajt **05_03** na scenu i zatim preimenovati novi objekat u **Goblin** (goblin će izgledati kao na sledećoj slici):

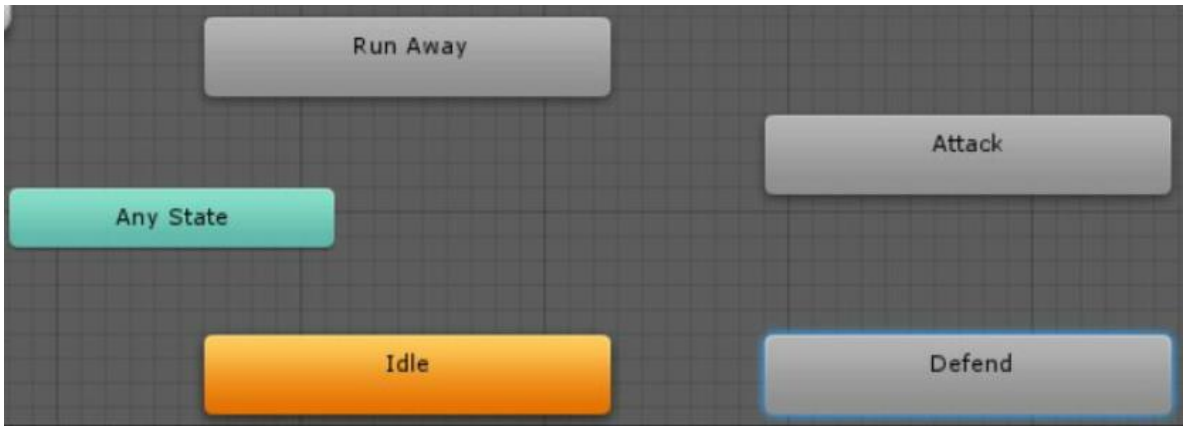
- Potrebno je dodati neku logiku ovom goblinu, to se neće koristiti u ovom poglavlju (tek u poglavlju 10) ali je dobro uraditi ovo na samom početku: unutar foldera Assets\Animation\Controllers napraviti novi **AnimatorController** (rmb->create->Animator Controller) - ovo zapravo predstavlja mašinu stanja za goblina. Dati mu naziv **GoblinAI**.

- Otvoriti GoblinAI. Dodati parametre kako bismo kontrolisali mašinu stanja (kliknuti na + unutar Parameters dela):

Tipovi ovih parametara:

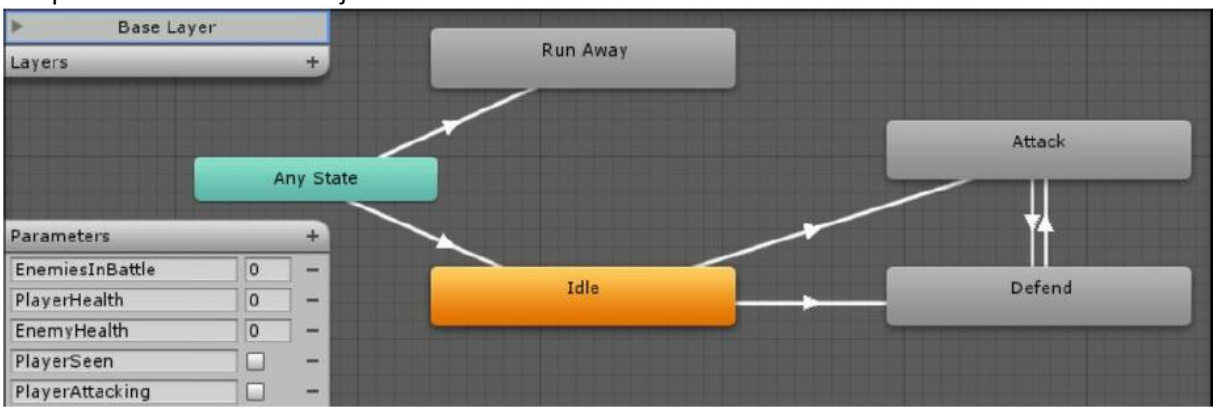
- EnemiesInBattle: Int
- PlayerHealth: Int
- EnemyHealth: Int
- PlayerSeen: Bool
- PlayerAttacking: Bool

- Dodajemo stanja kao na slici (rmb->Create State->Empty):



Stanje koje je obojeno u narandžasto je default tj. startno stanje. Moguće je promeniti default stanje klikom na njega pa Set As Default.

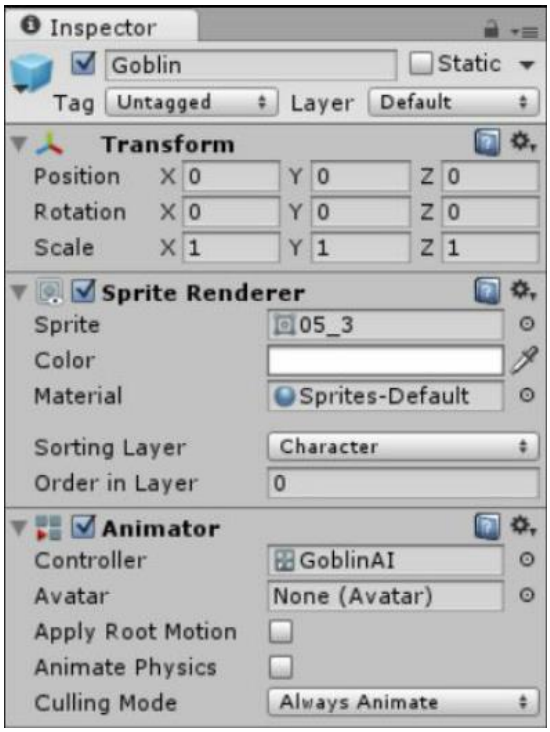
- Napraviti veze između stanja kao na slici:



Za svaku vezu je potrebno podesiti **Conditions** (u Inspectoru) na sledeći način:

- Idle | Attack, PlayerSeen - true (napadni igrača kada ga vidiš)
- Idle | Defend, PlayerSeen - true, PlayerAttacking - true (ako igrač prvi napadne, brani se)
- Attack | Defend, PlayerAttacking - true (ako igrač napadne, prebaci se iz stanja napada u stanje odbrane)
- Defend | Attack, PlayerAttacking - false (kada igrač prestane sa napadom, prebaci se iz stanja odbrane u stanje napada)
- Any State | Idle, PlayerSeen - false (ako izgubiš igrača iz vida, prebaci se u stanje mirovanja)
- Any State | Run Away, EnemyHealth > 0, EnemyHealth < 2, PlayerHealth > 2 (beži ako imaš zdravlje <2, a igrač ima zdravlje >2)

- Selektovati objekat Goblin unutar Hierarchy dela (leva strana); unutar Inspector dela dodati novu komponentu **Animator**; prevući fajl GoblinAI iz foldera Controllers na Animator komponentu u Inspector delu. Trebalo bi da izgleda kao na sledećoj slici:

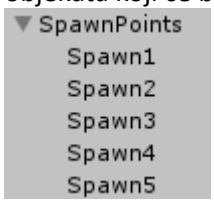


- Sada je potrebno od ovog goblina napraviti prefab - to se radi tako što prevučemo objekat Goblin iz Hierarchy dela u folder Assets\Prefabs\Characters. Obrisati objekat Goblin iz Hierarchy dela.

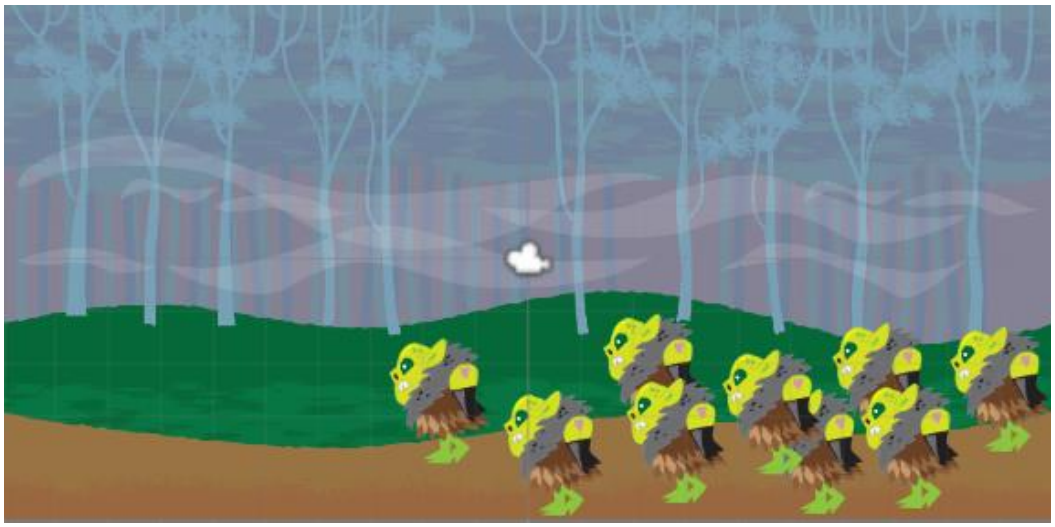
Ukoliko želimo da editujemo prefab, to radimo tako što selektujemo objekat u Characters folderu i zatim izvršimo izmene u Inspector delu - promene će se reflektovati na sve instance objekta na svim scenama gde se objekat nalazi. Ukoliko selektujemo objekat na samoj sceni pa vršimo izmene u Inspector-u, tada će se promene reflektovati samo na taj konkretan objekat.

8.2. Postavljanje spawn pozicija i upravljanje borbom

- Potrebno je da postavimo spawn (startne) pozicije na sceni na kojima će se pojaviti neprijatelji. Prvo pravimo novi objekat kog nazivamo SpawnPoints unutar Hierarchy dela (služiće kao kontejner koji će čuvati sve spawn-ove) i postavimo njegove Position koordinate na 0, 0, 0. Zatim pravimo 5 novih objekata koji će biti deca objektu SpawnPoints i nazivamo ih na sledeći način:



- Svakom spawnu dodeliti x i y koordinatu mesta gde će se nalaziti Goblin (to je lako moguće uraditi prevlačenjem prefaba Goblin na mapu i čitati njegove koordinate i zatim upisivati u spawn):



- Pošto smo poređali gobline, potrebno je da im naredimo da se na tim mestima i pojave preko skripte **BattleManager.cs** (napraviti unutar Assets/Scripts foldera). Svrha ove skripte je upravljanje svim aspektima borbe - od postavljanja scene i protivnika pa do same borbe. Ona radi samo kada smo u borbi. Trenutno, naša skripta će popuniti scenu sa protivnicima, a naš heroj će moći samo da pobegne.

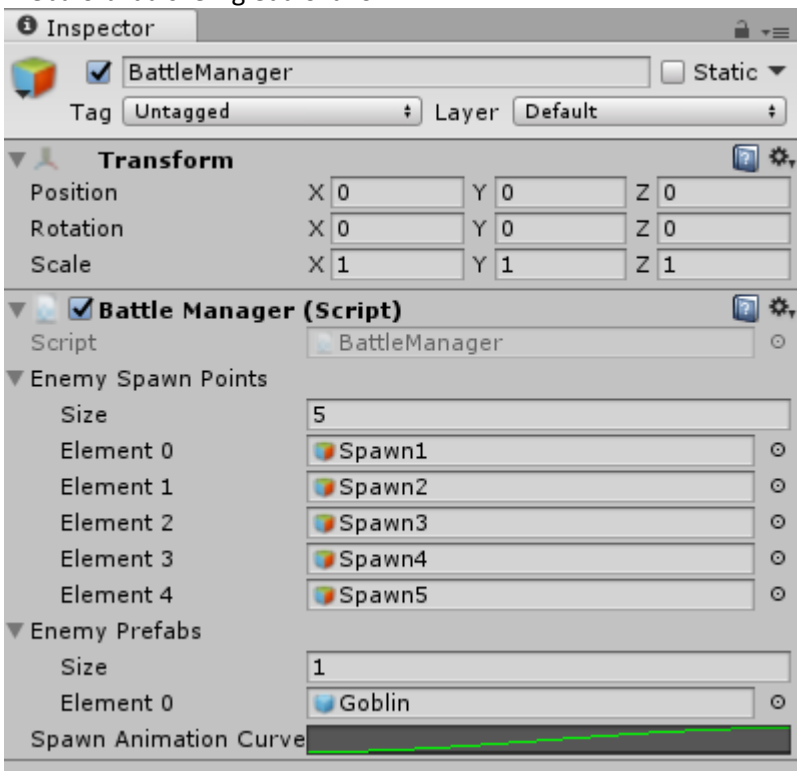
BattleManager.cs:

```
using System.Collections;
using UnityEngine;

public  class    BattleManager :MonoBehaviour {

public  GameObject[] EnemySpawnPoints; //spawn pozicije
public  GameObject[] EnemyPrefabs; //protivnici
public  AnimationCurve SpawnAnimationCurve; //koristimo AnimationCurve kako bismo
omogućili goblinima da (tecno) dosetaju od pozicije van ekrana do spawn pozicije

private int enemyCount; //broj aktivnih protivnika na sceni
enumBattlePhase
  {
    PlayerAttack,
    EnemyAttack
  }
private BattlePhase phase;//fleg koji nam govori ko je u fazi napada

// inicijalizacija scene za borbu - generise se slucajan broj goblina na mapi
// zatim se oni postavljaju na spawn pozicije preko Coroutine poziva i zapocinje se bitka tako sto
igrač napadne prvi
void  Start ()
  {
```

```
// broj generisanih protivnika po borbi
      enemyCount = Random.Range(1, EnemySpawnPoints.Length);

// postavlja protivnike na spawn pozicije
      StartCoroutine("SpawnEnemies");

// postavlja fleg za početno stanje bitke
      phase = BattlePhase.PlayerAttack;
   }

IEnumerator SpawnEnemies()
   {
// pravi gobline koji se krecu ka svojim spawn pozicijama, dolazeci van ekrana
for (int i = 0; i < enemyCount; i++)
      {
varnew  Enemy = (GameObject)Instantiate(EnemyPrefabs[0]); // novi protivnik
new  Enemy.transform.position = new  Vector3(10, -3, 0); // setuje se pocetna pozicija van
ekrana
yield return  StartCoroutine(MoveCharacterToPoint(EnemySpawnPoints[i], new  Enemy)); //
goblin dolazi na spawn poziciju
new  Enemy.transform.parent = EnemySpawnPoints[i].transform;
      }
   }

IEnumerator MoveCharacterToPoint(GameObject destination, GameObject character)
   {
float timer = 0f;
var StartPosition = character.transform.position; //startna pozicija protivnika
if (SpawnAnimationCurve.length > 0)
      {
//protivnik se kreće sve dok ne dođe do krajnje pozicije
while (timer < SpawnAnimationCurve.keys[SpawnAnimationCurve.length - 1].time)
         {
//Lerp omogucava postepeno pomeranje objekta od startne pozicije do destinacije
character.transform.position = Vector3.Lerp(StartPosition, destination.transform.position,
SpawnAnimationCurve.Evaluate(timer));
            timer += Time.deltaTime;
yield return new  WaitForEndOfFrame(); //ceka se da sledeci frejm bude spreman (inače bi se
animacija izvrsila odednom)
         }
      }
else
      {
character.transform.position = destination.transform.position;
```

```
            yield return null;
        }
    }

// dugme se pojavljuje ukoliko se igrač nalazi u fazi napada
void  OnGUI()
    {
if (phase == BattlePhase.PlayerAttack)
        {
if (GUI.Button(new  Rect(10, 10, 100, 50), "Run Away"))
            {
NavigationMenager.NavigateTo("World");
            }
        }
    }
}
```

- Napraviti prazan objekat u Hierarchy delu i nazvati ga **BattleManager**. Prevući skriptu BattleManager.cs na njega. Unutar Inspector dela, Enemy Spawn Points->Size staviti na 5, pojaviće se 5 novih elemenata ispod, pa prevući sve Spawn Pointe iz Hierarchy dela na svaki element. U Enemy Prefabs->Size ukucati 1 pa ispod toga prevući prefab Goblin. Selektovati Spawn Animation Curve i tu označiti poslednju krivu. Trebalo bi da sve izgleda ovako:



- Dodati prefab **GM** u hijerarhiju koji će omogućiti prelaz izmeću scena.

8.3. Čuvanje poslednje pozicije igrača na mapi

- Kada kliknemo na Run Away, vidimo da se igrač uvek nalazi na istom mestu na mapi. Da bismo to promenili, potrebno je da čuvamo poslednju poznatu poziciju igrača na mapi. Pravimo skriptu unutar Assets\Scripts koja se naziva **GameState.cs**.

GameState.cs:

```
usingSystem.Collections.Generic;
using UnityEngine;

public  static class    GameState
{
public  static Player CurrentPlayer = ScriptableObject.CreateInstance<Player>();

//rečnik koji cuva scene i poslednju poziciju u sceni gde je bio igrač
public  static Dictionary<string , Vector3> LastScenePositions = new  Dictionary<string ,
Vector3>();

public  static Vector3GetLastScenePosition(string  sceneName)
   {
//kada tražimo vrednost iz rečnika, prvo se proverava da li vrednost postoji...
if (GameState.LastScenePositions.ContainsKey(sceneName))
      {
var lastPos = GameState.LastScenePositions[sceneName];
 return  lastPos;
      }
//...inače se vraca default vrednost
else
      {
 return Vector3.zero;
      }
   }

public  static void  SetLastScenePosition(string  sceneName, Vector3 position)
   {
//kada dodajemo novu vrednost u rečnik, prvo se proverava da li vrednost vec postoji
//u rečniku. Ako postoji, onda se vrsi update...
if (GameState.LastScenePositions.ContainsKey(sceneName))
      {
GameState.LastScenePositions[sceneName] = position;
      }
//...ako ne postoji, onda se jednostavno pravi novi unos u rečnik
else
```

```
    {
GameState.LastScenePositions.Add(sceneName, position);
    }
  }
}
```

- Nakon ovoga, potrebno je ažurirati skriptu **MapMovement.cs** kako bi mapa učitavala poslednju lokaciju igrača ukoliko takva postoji i da čuva poslednju poziciju igrača kada izlazi sa scene.

MapMovement.cs:

```
//traži poslednju poziciju igrača za trenutnu scenu. Ukoliko takva postoji - pomera igrača na nju
void  Awake()
{
var lastPosition = GameState.GetLastScenePosition(Application.loadedLevelName);
if (lastPosition != Vector3.zero)
  {
transform.position = lastPosition;
  }
}

// kada se napusta scena, cuva se poslednja poznata pozicija igrača
void  OnDestroy()
{
GameState.SetLastScenePosition(Application.loadedLevelName, transform.position);
}
```

8.4. Nasumično generisanje borbi

- Potrebno je odraditi da igrač, dok je na glavnoj mapi, može naleteti na nasumično generisanu borbu dok putuje po mapi. Ažuriraćemo **MapMovement.cs**:

MapMovement.cs:

```
// promenljive koje se tičuverovatnoće desavanja borbe na mapi
int EncounterChance = 30;
float EncounterDistance = 0;
```

ažuriramo metodu Update:

```
if (Input.GetMouseButtonUp(0))
{
... stari kod ...
//određuje da li će biti borbe. Ako će biti, onda se određuje
        //distanca putanje koju će preci igrač do trenutka borbe
var EncounterProbability = Random.Range(1, 100);
```

```
if (EncounterProbability < EncounterChance)
        {
            EncounterDistance = (Vector3.Distance(StartLocation, TargetLocation) / 100) *
Random.Range(10, 100);
        }
else
        {
            EncounterDistance = 0;
        }
}
elseif (Input.touchCount > 0)
{

 ... stari kod ...

//copy-paste odozgo
  var EncounterProbability = Random.Range(1, 100);
  if (EncounterProbability < EncounterChance)
  {
     EncounterDistance = (Vector3.Distance(StartLocation, TargetLocation) / 100) *
Random.Range(10, 100);
  }
  else
  {
     EncounterDistance = 0;
  }
}

if (TargetLocation != Vector3.zero && TargetLocation != transform.position && TargetLocation
!= StartLocation)
      {
// pokretanje igrača na mapi
transform.position = Vector3.Lerp(StartLocation, TargetLocation,
MovementCurve.Evaluate(timer));

          timer += Time.deltaTime;
      }

//ako je daljina postavljena, borba se mora odigrati.
//Prema tome, kada je igrač putovao dovoljno daleko, ulazi se na ekran za
      //borbu
if (EncounterDistance > 0)
{
if (Vector3.Distance(StartLocation, transform.position) > EncounterDistance)
```

```
  {
    TargetLocation = Vector3.zero;
NavigationMenager.NavigateTo("Battle");
  }
}
```

- Dodati scenu **Battle** u Build Settings.

Zadatak za vežbu

Ograničiti igrača da može ući u samo jednu borbu dok je na glavnoj mapi, sve dok ne uđe u grad. Kada izađe iz grada, ponovo može ući u samo jednu borbu. (trenutna postavka je da može ući u neograničen broj borbi u bilo kom trenutku)

Poglavlje 9. Inventar

9.1 Kreiranje scene

- Pravimo novu scenu koju čuvamo pod nazivom **Shop** (unutar Scenes foldera)

- Prevući sliku **ShopScreen** iz foldera Sprites na glavni ekran scene i proveriti da se novi game objekat naziva ShopScreen (unutar Hierarchy prostora)

- Podeliti sliku **Weapon Icons 1** preko Sprite Editora na delove

- U Hierarchy delu, napraviti nove objekte tako da izgledaju kao na slici:

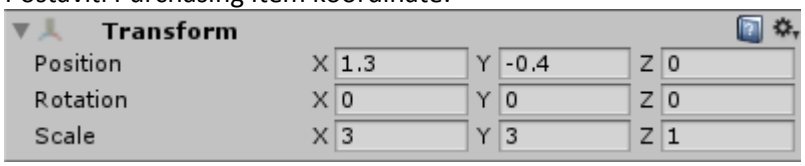
- Dodati komponentu **Sprite Renderer** na objekte BackButton i BuyButton. Prevući sličice BackButton i BuyButton u odgovarajući Sprite Renderer svakog objekta. Scena bi trebalo da izgleda ovako:

- Dodati komponentu **Box Colider 2D** sledećim objektima Slot1-Slot6, BuyButton. Za objekat BackButton dodati komponentu **Circle Colider 2D**. Za Slot1-Slot6 postaviti Scale za X i Y koordinatu na 0,32 i postaviti Position koordinate na sledeći način:

Slot1 X: -1.24, Y: 0.5
Slot2 X: -0.24, Y: 0.5
Slot3 X: -1.24, Y: -0.24
Slot4 X: -0.24, Y: -0.24
Slot5 X: -1.24, Y: -0.94
Slot6 X: -0.24, Y: -0.94

(za BuyButton i BackButton nije potrebno podešavati koordinate Box/Circle Colider-a jer će se one postaviti automatski kako bi popunile sličicu)

Postaviti Purchasing Item koordinate:



Inventar bi trebalo da izgleda ovako:

Svakom objektu dodati komponentu Sprite Renderer i postaviti Sorting Layer svakog objekta na **GUI**, osim za objekat ShopScreen, koga treba postaviti na **Background**.

9.2. Kreiranje predmeta za skladište (inventory)

- Pravimo novu C# skriptu (Assets/Scripts) pod nazivom **InventoryItem,** obrisati sav kod iz nje i napisati sledeće:

InventoryItem.cs:

```
using UnityEngine;
public   class   InventoryItem : ScriptableObject
{
        public   Sprite Sprite;
        public   Vector3 Scale;
        public   string  ItemName;
        public   int Cost;
        public   int Strength;
        public   int Defense;
}
```

Ovim smo definisali pojedina svojstva naših predmeta, kao što su cena, snaga, odbrana...

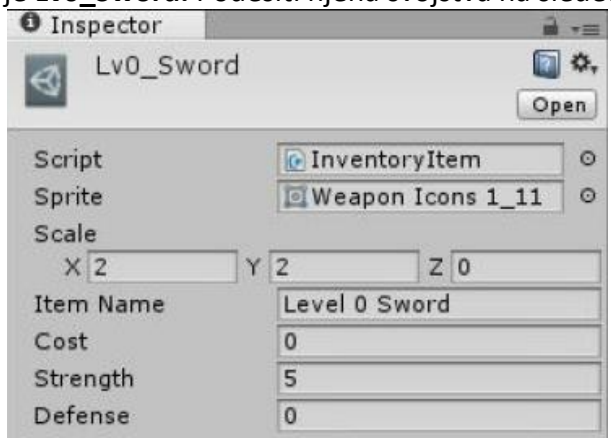
- Napraviti novu C# skriptu u (Assets/Scripts/Editor) pod nazivom **InventoryItemAssetCreator.**

InventoryItemAssetCreator.cs

```
using UnityEngine;
using UnityEditor;
public  class   InventoryItemAssetCreator :MonoBehaviour
{
  [MenuItem("Assets/Create/Inventory Item")]
public  static void  CreateAsset()
  {
//Kreira novu instancu ScribtibleObject
InventoryItem InventoryItemAssetCreator =
                                        ScriptableObject.CreateInstance<InventoryItem>();

        //Kreira .asset fajl za naš novi objekat i čuva ga
AssetDatabase.CreateAsset(InventoryItemAssetCreator, "Assets/new  Item.asset");
AssetDatabase.SaveAssets();

        //Prebacuje inspektor na naš novi objekat
EditorUtility.FocusProjectWindow();
Selection.activeObject = InventoryItemAssetCreator;
  }
}
```

Ovim smo dodali stavku u meni pod nazivom **Inventory Item**, sada možemo praviti objekte tog tipa.

- Sada je potrebno da napravimo neki Inventory Item. Napraviti folder **Inventory Items** unutar **Assets/Resources**.Napraviti novu InventoryItem klasu (RMB->Create->Inventory Item). Nazvati je **Lv0_Sword.** Podesiti njena svojstva na sledeći način:



Ponoviti postupak za još jedan item koji će se zvati Lv0_Axe.

9.3. Upravljanje prodavnicom

Pošto imamo izgled inventara i 2 predmeta, vreme je da ih spojimo. (S obzirom da će folderi ShopSlot i ShopManager zavisiti jedan od drugog, u toku kucanja koda pojavljivaće se greške, što je normalno).

- U folderu Assets/Scripts napraviti folder **Shop**, zatim u istom folderu napraviti C# skript **ShopManager**.

ShopManager.cs:

```
using UnityEngine;
public  class   ShopManager :MonoBehaviour
{
public  Sprite ShopOwnerSprite;
public  Vector3 ShopOwnerScale;
public  GameObject ShopOwnerLocation;
public  GameObject PurchasingSection;
public  SpriteRenderer PurchaseItemDisplay;
public  ShopSlot[] ItemSlots;
public  InventoryItem[] ShopItems;
private static  ShopSlot SelectedShopSlot;
private int nextSlotIndex = 0;

//kada igrač uđe na ekran za kupovinu, prikazuje se vlasnik prodavnice i njegova roba
void  Start()
   {
var OwnerSpriteRenderer = ShopOwnerLocation.GetComponent<SpriteRenderer>();
      OwnerSpriteRenderer.sprite = ShopOwnerSprite;
OwnerSpriteRenderer.transform.localScale = ShopOwnerScale;

//pakujemo iteme u inventar
if (ItemSlots.Length > 0 && ShopItems.Length > 0)
      {
for (int i = 0; i < ShopItems.Length; i++)
         {
if (nextSlotIndex > ItemSlots.Length) break;
            ItemSlots[nextSlotIndex].AddShopItem(ShopItems[i]);
            ItemSlots[nextSlotIndex].Manager = this;
            nextSlotIndex++;
         }
      }
   }

//mogucnost selektovanja itema u prodavnici
```

```
public  void  SetShopSelectedItem(ShopSlot slot)
   {
      SelectedShopSlot = slot;
      PurchaseItemDisplay.sprite = slot.Item.Sprite;
      PurchasingSection.SetActive(true);
   }

//mogucnost brisanja itema iz prodavnice
public  void  ClearSelectedItem()
   {
Debug.Log("Clearing Shop Purchase area");
      SelectedShopSlot = null;
      PurchaseItemDisplay.sprite = null;
      PurchasingSection.SetActive(false);
   }

//kupovina trenutno selektovanog itema
public  static void  PurchaseSelectedItem()
   {
      SelectedShopSlot.PurchaseItem();
   }
}
```

- Pravimo novi C# skript **ShopSlot** u folderu Shop. Taj skript će definisati slotove u prodavnici koji pamte šta se drži na policama.

ShopSlot.cs:

```
using UnityEngine;

public  class   ShopSlot :MonoBehaviour
{
public  InventoryItem Item;
public  ShopManager Manager;

//omogucava dodavanje itema na trenutni slot i prikazuje ga
public  void  AddShopItem(InventoryItem item)
   {
var spriteRenderer = GetComponent<SpriteRenderer>();
      spriteRenderer.sprite = item.Sprite;
spriteRenderer.transform.localScale = item.Scale;
      Item = item;
   }

//kontrolise kako je item kupljen
```

```
public  void  PurchaseItem()
   {
GameState.CurrentPlayer.Inventory.Add(Item);
      Item = null;
var spriteRenderer = GetComponent<SpriteRenderer>();
      spriteRenderer.sprite = null;
      Manager.ClearSelectedItem();
   }

//omogucava kliktanje po itemima na slotovima
void  OnMouseDown()
   {
if (Item != null)
      {
         Manager.SetShopSelectedItem(this);
      }
   }
}
```

- Napraviti novi C# skript **BuyButton** i smestiti ga u folder Shop.

BuyButton.cs:

```
using UnityEngine;

public  class   BuyButton :MonoBehaviour
{
void  OnMouseDown()
   {
ShopManager.PurchaseSelectedItem();
   }
}
```

- Napraviti novi C# skript **BackButton** i smestiti ga u folder Shop.

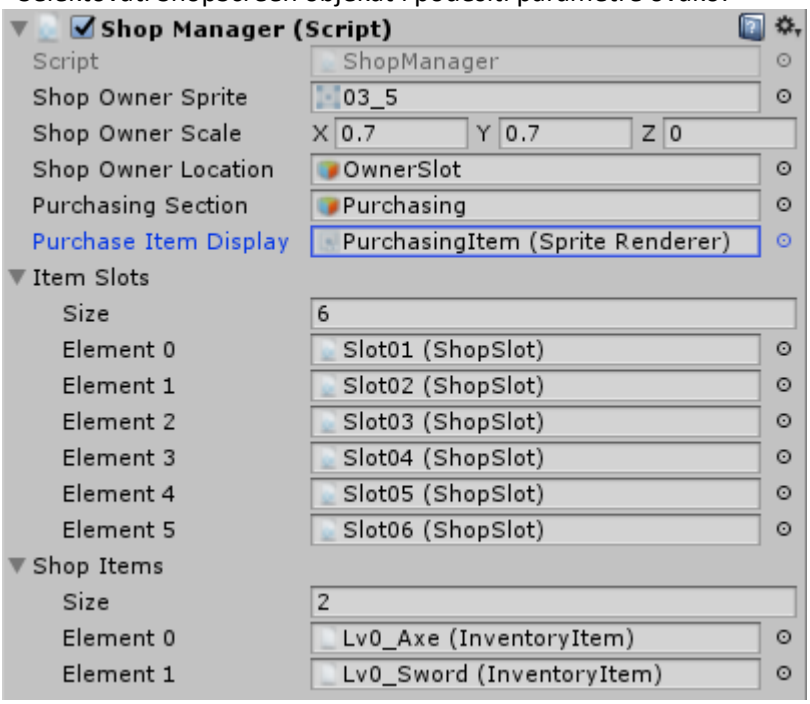
```
using UnityEngine;

public  class  BackButton :MonoBehaviour
{
void  OnMouseDown()
   {
NavigationMenager.GoBack();
   }
}
```

- Otvoriti klasu **Player** unutar foldera (Assets/Scripts) i ažurirati je da izgleda ovako:

```
usingSystem.Collections.Generic;

public  class  Player :Entity
{
   //definisemo one atribute koji su konkretni za igrača i
//sto ga razlikuje od ostalih entiteta u igrici
public  List<InventoryItem> Inventory = new
                                        System.Collections.Generic.List<InventoryItem>();
public  string [] Skills;
public  int Money;
}
```

- Prevući **ShopManager** skriptu (Assets/Shop) na **ShopScreen** objekat u Hierarchy delu.
- Prevući **BuyButton** skriptu (Assets/Shop) na **BuyButton** objekat u Hierarchy delu.
- Prevući **BackButton** skriptu (Assets/Shop) na **BackButton** objekat u Hierarchy delu.

- Selektovati ShopScreen objekat i podesiti parametre ovako:



(kliknuti na tačkicu pored svakog polja i izabrati odgovarajuće nazive)

- Moramo dodati mogućnost da se igrač, kada izađe iz prodavnice, nađe na mestu gde je u nju i ušao. Modifikovaćemo NavigationManager kako bismo zapamtili poslednje mesto na kojem je igrač bio. Otvoriti **NavigationManager** (Assets/Scripts):

- dodati linije

```
using UnityEngine;
using UnityEngine.SceneManagement;
```

- unutar klase dodati liniju `private static string  PreviousLocation;`
- modifikovati NavigateTo metodu, dodati na vrh:

```
PreviousLocation = SceneManager.GetActiveScene().name; //cuva poslednju scenu na kojoj je bio igrač
```

- dodati novu funkciju:

```
//vracanje na prethodnu scenu
public  static void  GoBack()
   {
var backlocation = PreviousLocation; //cuva prethodnu lokaciju
      PreviousLocation = SceneManager.GetActiveScene().name; //postavlja trenutnu scenu na mesto prethodne
Debug.Log(backlocation);
FadeInOutManager.Instance.Fading(backlocation);
   }
```

- Igrač će ući u prodavnicu tako što će stati ispred nje i pritisnuti taster UP kako bi ušao.
Pravimo novu C# skriptu unutar foldera Assets/Scripts pod nazivom **ShopEntry:**

ShopEntry.cs

```
using UnityEngine;
public  class  ShopEntry :MonoBehaviour
{
bool  canEnterShop; //kontrolise da li možemo uci u prodavnicu ili ne

//promena flega se vrsi preko ove funkcije
void  DialogVisible(bool  visibility)
   {
      canEnterShop = visibility;
   }

//detektuje da li je igrač ispred prodavnice
void  OnTriggerEnter2D(Collider2D col)
   {
      DialogVisible(true);
   }
void  OnTriggerExit2D(Collider2D col)
   {
      DialogVisible(false);
```

```
    }

//detektuje da li je igrač pritisnuo UP za ulazak
void  Update()
    {
if (canEnterShop &&Input.GetKeyDown(KeyCode.UpArrow))
        {
if (NavigationManager.CanNavigate(this.tag))
            {
NavigationManager.NavigateTo(this.tag);
            }
        }
    }
//igrač dobija obavestenje da može uci u prodavnicu kada se nađe ispred nje
void  OnGUI()
    {
if (canEnterShop)
        {
//layout start
GUI.BeginGroup(
new Rect(
Screen.width / 2 - 150,
            50,
            300,
            50));
//the menu background box
GUI.Box(new Rect(0, 0, 300, 250), "");
//Dialog detail—updated to get better detail
GUI.Label(
new Rect(15, 10, 300, 68),
"Do you want to Enter the Shop? (Press up)");
//layout end
GUI.EndGroup();
        }
    }
}
```

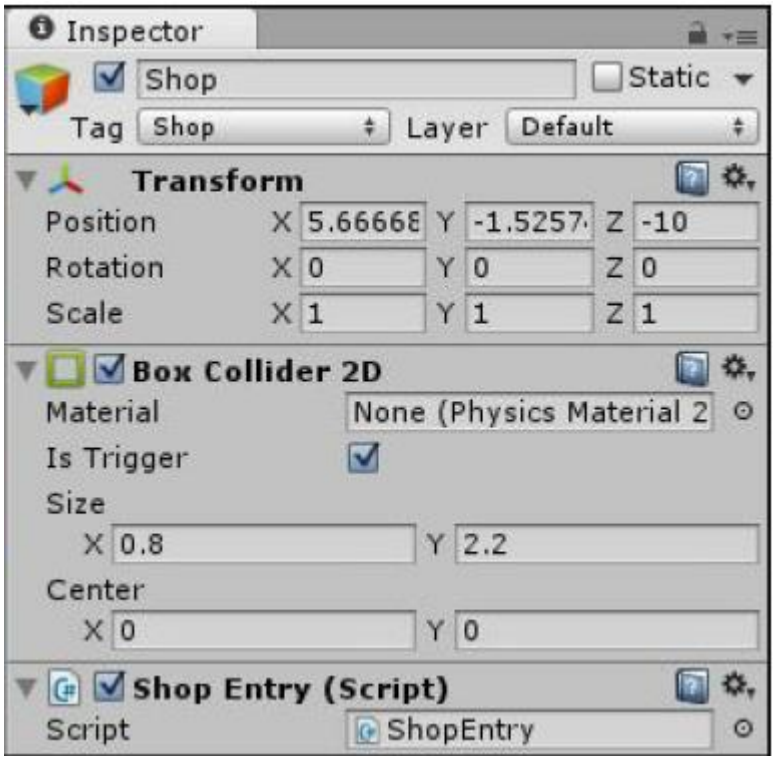
- Ići u **File->Build Settings** i tamo kliknuti na dugme **Add Open Scenes**, kako bi se pojavila nova stavka Scenes/Shop.

- Ažurirati skriptu **NavigationManager**. Dodati u rečnik (Dictionary) sledeći red:

```
{ "Shop", new Route {CanTravel = true}},
```

- Otvaramo sačuvanu scenu unutar Assets/Scenes pod nazivom Main. Pravimo novi objekat kao dete objektu WorldBounds, pod nazivom **Shop**. Prevući ShopEntry (Assets/Scripts) na

novokreirani objekat Shop. Dodati **Box Collider 2D** komponentu objektu Shop. Štiklirati **Is Trigger**. Podesiti ostala svojstva da izgledaju ovako:



- Dodati novi tag Shop (klikne se na padajući meni pa stavka Add Tag unutar Inspector dela) i dodeliti tag Shop objektu Shop.

- Nakon ovoga, moguće je pokrenuti projekat i dovesti igrača ispred prodavnice, gde će se ispisati poruka.

9.4. Prikaz inventara igrača

- Napraviti novu skriptu unutar Scripts foldera pod nazivom **PlayerInventoryDisplay**.

PlayerInventoryDisplay.cs

```
using UnityEngine;

public  class  PlayerInventoryDisplay :MonoBehaviour
{
bool  displayInventory = false; //određuje da li je prozor inventara prikazan ili ne
Rect inventoryWindowRect;
private Vector2 inventoryWindowSize = new Vector2(150, 150);
Vector2 inventoryItemIconSize = new Vector2(130, 32);
float offsetX = 6;
float offsetY = 6;

void  Awake()
  {
    inventoryWindowRect = new Rect(
Screen.width - inventoryWindowSize.x,
Screen.height - 40 - inventoryWindowSize.y,
    inventoryWindowSize.x,
    inventoryWindowSize.y);
  }

//prikaz dugmeta za pristup igračevom inventaru
void  OnGUI()
  {
if (GUI.Button(new Rect(Screen.width - 40, Screen.height - 40, 40, 40), "INV"))
    {
      displayInventory = !displayInventory;
    }
if (displayInventory)
    {
      inventoryWindowRect = GUI.Window(0, inventoryWindowRect,
DisplayInventoryWindow, "Inventory");
      inventoryWindowSize = new Vector2(inventoryWindowRect.width,
inventoryWindowRect.height);
    }
  }

//prolazi kroz sve iteme u igračevom inventaru (ako ih ima) i prikazuje ih
void  DisplayInventoryWindow(int windowID)
  {
var currentX = 0 + offsetX;
var currentY = 18 + offsetY;
```

```
foreach (var item inGameState.CurrentPlayer.Inventory)
      {
Rect texcoords = item.Sprite.textureRect;
         texcoords.x /= item.Sprite.texture.width;
texcoords.y /= item.Sprite.texture.height;
texcoords.width /= item.Sprite.texture.width;
texcoords.height /= item.Sprite.texture.height;
GUI.DrawTextureWithTexCoords(new Rect(
         currentX,
         currentY,
item.Sprite.textureRect.width,
item.Sprite.textureRect.height),
item.Sprite.texture,
         texcoords);
         currentX += inventoryItemIconSize.x;
if (currentX + inventoryItemIconSize.x + offsetX > inventoryWindowSize.x)
         {
            currentX = offsetX;
            currentY += inventoryItemIconSize.y;
if (currentY + inventoryItemIconSize.y + offsetY > inventoryWindowSize.y)
            {
 return ;
            }
         }
      }
   }
}
```

- Prevući skriptu **PlayerInventoryDisplay.cs** na objekat Player u Hierarchy delu (objekat iz scene Main)

- Kada igrač kupi oružje potrebno je da omogućimo da to oružje i koristi kada je potrebno - otvoriti **Player** klasu unutar (Assets/Scripts). Dodati novu funkciju:

```
//promena statistike igrača u zavisnosti od novog itema
public  void  AddinventoryItem(InventoryItem item)
   {
this.Strength += item.Strength;
this.Defense += item.Defense;
      Inventory.Add(item);
   }
```

- Ažurirati ShopSlot skriptu (Assets/Scripts/Shop) promeniti prvi red unutar metode PurchaseItem() sa ovom linijom:

```
GameState.CurrentPlayer.AddinventoryItem(Item);
```

Zadatak za vežbu

Prikazati inventar igrača u donjem levom uglu ekrana.

Poglavlje 10. Spremanje za borbu

Kako stižemo do kraja putovanja nase RPG igrice, ulazimo u poslednji aspekt framework-a. U “Spotlight“ dolazi jedan od najtežih delova bilo kojeg razvijanja igrice, a to je: BITKA. Zašto najteži? U zavisnosti od toga na koji način pravite borbu, određuje da li ćete privući igrače da igraju ili ne, jer ako je borba dobro odrađena i zanimljiva, ako je uvek izazovna, to će definitivno privući veći broj igrača.

Šta ćemo pokriti u ovom poglavlju:

- Priprema statistike za borbu i pravljenje UI-a igre
- Implementiranje potez-basiran sistem borbe
- Radićemo sa Mecanim u kodu
- Pravljenje naprednijih sprajtova u 2D GUI sistemu

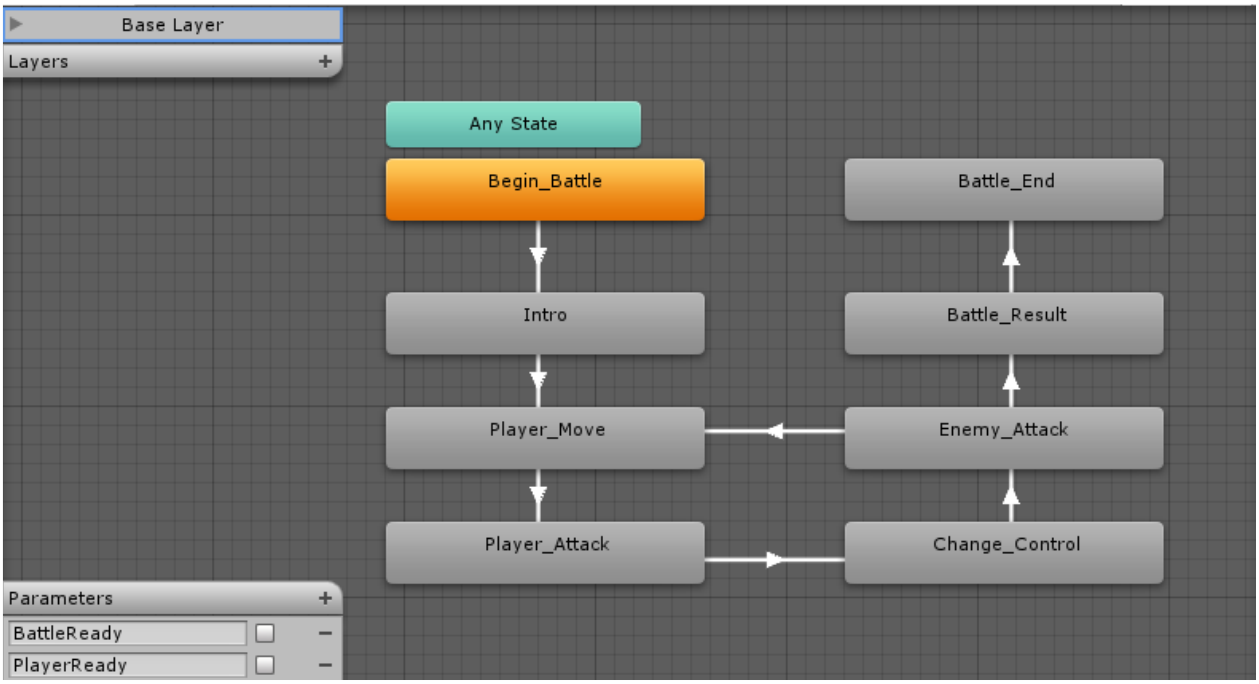
Počećemo sa “battle state machine“, spremanja igrača za borbu i onda da ih prati neke GUI interakcije za igrača da moze da bude u mogućnosti da počne sa napadom u borbi. U sledećem poglavlju ćemo završiti ceo proces borbe samog igrača.

Počnimo!

10.1 Battle state manager

Prvo što treba da uradimo jeste da zamenimo našu privremenu *state machine* sa odgovarajućom, koristeći sva odgovarajuća Mecanim-ova sredstva koja su nam potrebna.

U Assets\Animation\Controllers napravite novi **Animator Controller** i nazovite ga BattleStateMachine i onda možemo da počnemo da sastavljamo stanja borbe.



Kao što vidimo, imamo osam stanja s kojim možemo da kontrolišemo tok borbe I dva Boolean parametra da kontrolišemo tranzicije.

Tranzicije su definisane na sledeći način:

- Iz **Begin_Battle** u **Intro**
  - BattleReady = true (Transition Duration = 0)
- Iz **Intro u Player_Move**
  - Exit Time = 0.9 (Transition Duration = 2)
- Iz **Player_Move** u **Player_Attack**
  - PlayerReady = true (Transition Duration = 0)
- iz **Player_Attack** u **Change_Control**
  - PlayerReady = false (Transition Duration = 2)
- Iz **Change_Control** u **Enemy_Attack**
  - Exit Time = 0.9 (Transition Duration = 2)
- Iz **Enemy_Attack** u **Player_Move**
  - BattleReady = true (Transition Duration = 2)
- Iz **Enemy_Attack** u **Battle_Result**
  - BattleReady = false (Transition Time = 2)
- Iz **Battle_Result** u **Battle_End**
  - Exit Time = 0.9 (Transition Time = 5)

Sada, kada smo napravili animator za stanja, sve što treba da uradimo jeste da ga privučemo našem battle manager-u kako bi bilo dostupan kada se pokrene borba, a to ćemo uraditi ovako:

1. Otvoriti **Battle** scenu
2. Selektovati BattleManager objekat iz hiararhije i dodati **Animator** komponentu
3. Sada prevući **BattleStateMachine** animator kontroler koji smo napravili u *Controller* properti Animator komponente

Sada da bi mogli da referenciramo **BattleStateMachine** otvorićemo *BattleManager* skriptu i dodati na vrhu:

```
private Animator battleStateManager;
```

I da bi ga uhvatili, u start funkciji dodajemo

```
void Start()
{
    battleStateManager = GetComponent<Animator>();
```

10.2 Pristupanje state manager-u u kodu

Sada, prvo što ćemo uraditi jeste obrisati prethodno definisani enum stanje masine:

```
enum BattlePhase
{
    PlayerAttack,
    EnemyAttack
}
private BattlePhase phase;
```

i takođe obrisati iz start metode:

```
phase = BattlePhase.PlayerAttack;
```

I dalje postoji referenca u OnGUI metodi koju ćemo uskoro zameniti, ali slobodno je možete obrisati sada.

Sada, umesto starog enum-a, stavljamo novi s kojim ćemo raditi:

```
public enum BattleState
{
    Begin_Battle,
    Intro,
    Player_Move,
    Player_Attack,
    Change_Control,
    Enemy_Attack,
    Battle_Result,
    Battle_End
}
```

Dodaćemo na vrh klase

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

A onda, dodajemo na vrh klase:

```
private Dictionary<int, BattleState> battleStateHash = new Dictionary<int,
BattleState>();
private BattleState currentBattleState;
```

I konačno da bi mogli da prođemo kroz sva stanja animatora koji smo napravili, trebaće nam metoda `GetAnimationStates` u `BattleManager` klasi:

```
void GetAnimationStates()
    {
        foreach (BattleState state in
(BattleState[])System.Enum.GetValues(typeof(BattleState)))
        {
            battleStateHash.Add(Animator.StringToHash("Base Layer." +
state.ToString()),state);
        }
    }
```

Ova metoda samo generise hash-ove za odgovarajuća stanja animiranja u Mecanim-u i pakuje hash rezultate u rečnik tako da ne moramo da računamo u kada treba da razogavarmo sa state mašinom u realnom vremenu.

Da bi se to izvršilo, samo pozovemo napravljenu metodu `GetAnimationStates()` u `Start` funkciji u `BattleManager` skripti.

```
      GetAnimationStates();
```

Sada kada imamo naša stanja, možemo da ih koristimo u igri i da kontrolišemo logiku koja je primenjana i na GUI elemente koji su nacrtani na ekran.

Sada, dodajemo Update funkciju u BattleManager klasi:

```
void Update()

{
    currentBattleState =
battleStateHash[battleStateManager.GetCurrentAnimatorStateInfo(0).nameHash];

        switch (currentBattleState)
        {
            case BattleState.Intro:
                break;
            case BattleState.Player_Move:
                break;
            case BattleState.Player_Attack:
                break;
            case BattleState.Change_Control:
                break;
            case BattleState.Enemy_Attack:
```

```
                break;
            case BattleState.Battle_Result:
                break;
            case BattleState.Battle_End:
                break;
            default:
                break;
        }
    }
```

Ovaj kod uzima renutno stanje iz animatora jednom po frejmu i postavlja neki izbor (switch naredba) šta da se desi u zavisnosti od trenutnog stanja.

Sada menjamo OnGUI metodu:

```
    void OnGUI()
    {
        switch (currentBattleState)
        {
            case BattleState.Begin_Battle:
                break;
            case BattleState.Intro:
                break;
            case BattleState.Player_Move:
                break;
            case BattleState.Player_Attack:
                break;
            case BattleState.Change_Control:
                break;
            case BattleState.Enemy_Attack:
                break;
            case BattleState.Battle_Result:
                break;
            default:
                break;
        }
    }
```

10.3 Početak bitke

Kao što znamo, state mašina ceka `Begin_Battle` stanje da bi započeli borbu, naravno mi to želimo da uradimo kada se sve postavi na scenu što nam je potrebno.

U trenutnoj **Battle** sceni koju smo dodali u 7. poglavlju, ucita se igrač i nekoliko random neprijatelja tako što se poziva funkcija `SpawnEnemies`. Znaci, kad postavimo sve gobline na scenu, tek onda možemo da započnemo borbu.

Da bi rekli state mašiini da započne borbu, samo treba da dodamo na kraju for petlje u SpawnEnemies Ienumerator funkcije:

```
battleStateManager.SetBool("BattleReady", true);
```

10.4 Ulepšavanje početka scene

Sada ćemo na početku scene da dodamo intro, tj na Intro stanje da ispise poruka o našoj borbi.

Trenutno, state mašina pauzira na intro stanju par sekundi, pa dok je pauzirano, hajde da dodamo neki GUI dialog da kažemo igraču nešto o predstojećoj borbi. Jednostavno ćemo dodati na `case` liniji `Intro` stanja u `OnGUI` funkciji:

```
case BattleState.Intro :
                GUI.Box(new Rect((Screen.width / 2) - 150 , 50, 300, 50),
                "Battle between Player and Goblins");
                break;
```

Sada kada je igrač obavešten da će doći do borbe, želeo ne želeo, moraće da stupi u borbu!

Međutim, možemo ostaviti da u slučaju da želi igrač da pobegne, da mu ostavimo varijantu da se vrati nazad u svet.

```
case BattleState.Player_Move:
                if (GUI.Button(new Rect(10, 10, 100, 50), "Run Away"))
                {
                    GameState.playerReturningHome = true;
                    NavigationMenager.NavigateTo("World");
                }
                break;
```

Kako sada već igrač treba da se aktivira i borba je u progresu, trebaće nam neke interakcije za usera, pa hajde da mu damo nešto da može da klikće.

10.5 Krećemo sa GUI-om

Prvo, želimo da našem igraču dodelimo neko oružje koje je već mogao da kupi, ali hoćemo mu dozvoliti da izabere oružje. Da bi to uradili, ubacićemo nekoliko dugmića koja mogu da se klikću i u zavisnosti da li smo kliknuli na oružje ili nismo, moći ćemo da napadamo s njim.

CommandBar (kontejner) ćemo koristiti da bi napravili dugmiće za naša oružja. Krećemo sa implementacijom CommandBar klase i dodavanje definicije dugmeta na njoj.

9.5.1. The command bar

Napravite C# skriptu i nazovite je CommandBar, a onda zamenite kod sa ovim:

```
using UnityEngine;
using System.Collections;


public class CommandBar : MonoBehaviour {
    public bool anchor = true;
    public Vector2 anchorOffset = Vector2.zero;
    public ScreenPositionAnchorPoint anchorPoint =
ScreenPositionAnchorPoint.BottomCenter;

    private CommandButton [] commandButtons;

    public float buttonSize = 1.28f;
    public float buttonRows = 1;
    public float buttonColumns = 6;
    public float buttonRowSpacing = 0;
    public float buttonColumnSpacing = 0;
    public Sprite DefaultButtonImage;
    public Sprite SelectedButtonImage;
    private float ScreenHeight;
    private float ScreenWidth;

    public int Layer
    {
        get { return gameObject.layer; }
    }
    float Width
    {
    get
        {
            return (buttonSize * buttonColumns) +
Mathf.Clamp((buttonColumnSpacing * (buttonColumns - 1)), 0, int.MaxValue);
        }
```

```
    }
    float Height
    {
        get
        {
            return (buttonSize * buttonRows) + Mathf.Clamp((buttonColumnSpacing *
(buttonRows - 1)), 0, int.MaxValue);
        }
    }

}
```

Da bi tačno obezbedili gde će nam se nalaziti bar sa komandama, tj oružjem moramo to omogućiti preko fiksiranih pozicija, najlakše ćemo to uraditi tako što ćemo napraviti C# skriptu i nazvati je `ScreenPositionAnchorPoint` i onda zameniti njen kod sa sledećim:

```
public enum ScreenPositionAnchorPoint
{
    TopLeft,
    TopCenter,
    TopRight,
    MiddleLeft,
    MiddleCenter,
    MiddleRight,
    BottomLeft,
    BottomCenter,
    BottomRight
}
```

Sada u CommandBar skripti, možemo samo dodati varijable na vrh klase da izaberemo odgovarajuće mesto gde želimo da postavimo nase barove koristeći postojeći enum:

```
public bool anchor = true;
public Vector2 anchorOffset = Vector2.zero;
public ScreenPositionAnchorPoint anchorPoint =
ScreenPositionAnchorPoint.BottomCenter;
```

A sada da bi tačno odredili pozicije, samo dodajte sledeću funkciju u `CommandBar` skripti:

```
Vector2 CalculateAnchorScreenPosition()
    {
        Vector2 position = Vector2.zero;
        switch (anchorPoint)
        {
            case ScreenPositionAnchorPoint.BottomLeft:
                position.y = -(ScreenHeight / 2) + Height;
```

```
                position.x = -(ScreenWidth / 2) + buttonSize;
                    break;
            case ScreenPositionAnchorPoint.BottomCenter:
                position.y = -(ScreenHeight / 2) + Height;
                position.x = -(Width / 2);
                    break;
            case ScreenPositionAnchorPoint.BottomRight:
                position.y = -(ScreenHeight / 2) + Height;
                position.x = (ScreenWidth / 2) - Width;
                    break;
            case ScreenPositionAnchorPoint.MiddleLeft:
                position.y = (Height / 2);
                position.x = -(ScreenWidth / 2) + buttonSize;
                    break;
            case ScreenPositionAnchorPoint.MiddleCenter:
                position.y = (Height / 2);
                position.x = -(Width / 2);
                    break;
            case ScreenPositionAnchorPoint.MiddleRight:
                position.y = (Height / 2);
                position.x = (ScreenWidth / 2) - Width;
                    break;
            case ScreenPositionAnchorPoint.TopLeft:
                position.y = (ScreenHeight / 2) - Height;
                position.x = -(ScreenWidth / 2) + buttonSize;
                    break;
            case ScreenPositionAnchorPoint.TopCenter:
                position.y = (ScreenHeight / 2) - Height;
                position.x = -(Width / 2);
                    break;
            case ScreenPositionAnchorPoint.TopRight:
                position.y = (ScreenHeight / 2) - Height;
                position.x = (ScreenWidth / 2) - Width;
                    break;
        }
        return anchorOffset + position;
    }
```

A onda dodamo Awake() metodu:

```
void Awake()
{
    ScreenHeight = Camera.main.orthographicSize * 2;
    ScreenWidth = ScreenHeight * Screen.width / Screen.height;
}
```

Ovo koristimo da bi uzeli tačnu veličinu ekrana, što će nam biti korisno da bi lakse skalirali command bar.

Sada treba napraviti funkciju koja će napraviti dugmiće od naseg CommandBara na sledeći način:

```
CommandButton CreateButton()
    {
        // Create our new game object
        GameObject go = new GameObject("CommandButton");
        // Add components
        go.AddComponent<SpriteRenderer>();
        go.AddComponent<BoxCollider2D>();
        go.transform.parent = transform;
        // Init
        CommandButton button = go.AddComponent<CommandButton>();
        button.Init(this);
        return button;
    }
```

Ova funkcija radi sledeće:

- Pravi prazan objekat i naziva ga CommandButton
- Dodaje SpriteRenderer2D i BoxCollider2D komponente
- Postavlja ovo dugme dete command bara tako što setujemo transform parent
- Pravi i dodeljuje CommandButton skriptu na naše dugme
- Poziva Init fukciju dugmeta
- Kada se sve završi, vraća `CommandButton` objekat kome god da ga je pozvao

Sada kada imamo mogućnost da pravimo dugmiće, takođe nam treba mogućnost da ih pomeramo unutar comandnog bara. Dakle pravimo sledeću funkciju:

```
void InitButtonPositions()
    {
        int i = 0;
        float xPos = 0;
        float yPos = 0;
        for (int r = 0; r < buttonRows; ++r)
        {
            xPos = 0;
            for (int c = 0; c < buttonColumns; ++c)
            {
                commandButtons[i].transform.localScale = new
Vector3(buttonSize,buttonSize, 0);
                commandButtons[i].transform.localPosition = new Vector3(xPos,
yPos, 0);
```

```
                i++;
                xPos += buttonSize + buttonColumnSpacing;
            }
            yPos -= buttonSize + buttonRowSpacing;
        }
    }
```

Ova funkcija će da prođe kroz sve dugmiće koji su dodeljeni komandnom baru i da ih skalira i pozicionira onako kako treba. Sledećom funkcijom ćemo završiti inicijalizaciju ComandBar klase:

```
void InitCommandButtons()
    {
        commandButtons = new CommandButton[(int)buttonRows * (int)buttonColumns];
        for (int i = 0; i < commandButtons.Length; i++)
        {
            var newButton = CreateButton();
            if (i < GameState.CurrentPlayer.Inventory.Count)
            {
                newButton.AddInventoryItem(GameState.CurrentPlayer.Inventory[i]);
            }
            commandButtons[i] = newButton;
        }
        InitButtonPositions();
    }
```

Ovom funkcijom se pravi niz već konfigurisanih dugmića i dodajemo nove prazne command bar dugmiće na svaki element koristeći `CreateButton` funkciju. Kada ih sve napravimo, samo pozovemo `InitButtonPosition` funkciju da ih lepo posicionira.

Sve što ostaje jeste da komandni bar pozicioniramo kako treba, da bi to uradili dodaćemo funkciju:

```
void SetPosition(float x, float y)
    {
        transform.position = new Vector3(x, y, 0);
    }
```

Sada samo dodamo update funkciju sledećom:

```
    void Update () {

        Vector2 position = Vector2.zero;
        if (anchor)
        {
            position = CalculateAnchorScreenPosition();
        }
        else
```

```
        {
            position = transform.position;
        }
        SetPosition(position.x, position.y);
    }
```

Za kraj, treba da pozovemo funkciju `InitCommandButtons` u `Start()` metodi:

```
    void Start ()
    {
        InitCommandButtons();
    }
```

10.5.2. The command button

Počećemo tako što ćemo definisati CommandButton klasu

```
using UnityEngine;
using System.Collections;
[RequireComponent(typeof(SpriteRenderer))]
[RequireComponent(typeof(BoxCollider2D))]
public class CommandButton : MonoBehaviour
{
    private CommandBar commandBar;
    public InventoryItem Item;
    bool selected;
}
```

Primećujemo komande
```
[RequireComponent(typeof(SpriteRenderer))]
[RequireComponent(typeof(BoxCollider2D))]
```
to samo znaci da su obavezne komponente za nas CommandButton i da će nas Unity upozoriti ako ih nemamo.

Sledeće, treba dodati Init funkciju koju smo već zvali u CommandBar skripti:

```
public void Init(CommandBar commandBar)
{
    this.commandBar = commandBar;
    gameObject.layer = commandBar.Layer;
    var collider = gameObject.GetComponent<BoxCollider2D>();
    collider.size = new Vector2(1f, 1f);
    var renderer = gameObject.GetComponent<SpriteRenderer>();
    renderer.sprite = commandBar.DefaultButtonImage;
    renderer.sortingLayerName = "GUI";
    renderer.sortingOrder = 5;
}
```

Kada inicijalizujemo sadržaj dugmeta, moramo postaviti sve komponente koje objekat treba da ima:

- Definišemo referencu na CommandBar zato što svako dugme komunicira sa njom, tj sa roditeljom
- Postavljamo layer za novi button isti kao za command bar
- Uzimamo BoxCollider2D objekat i postavljamo određenu veličinu
- Uzimamo SpriteRenderer i dajemo joj difoltni sprajt

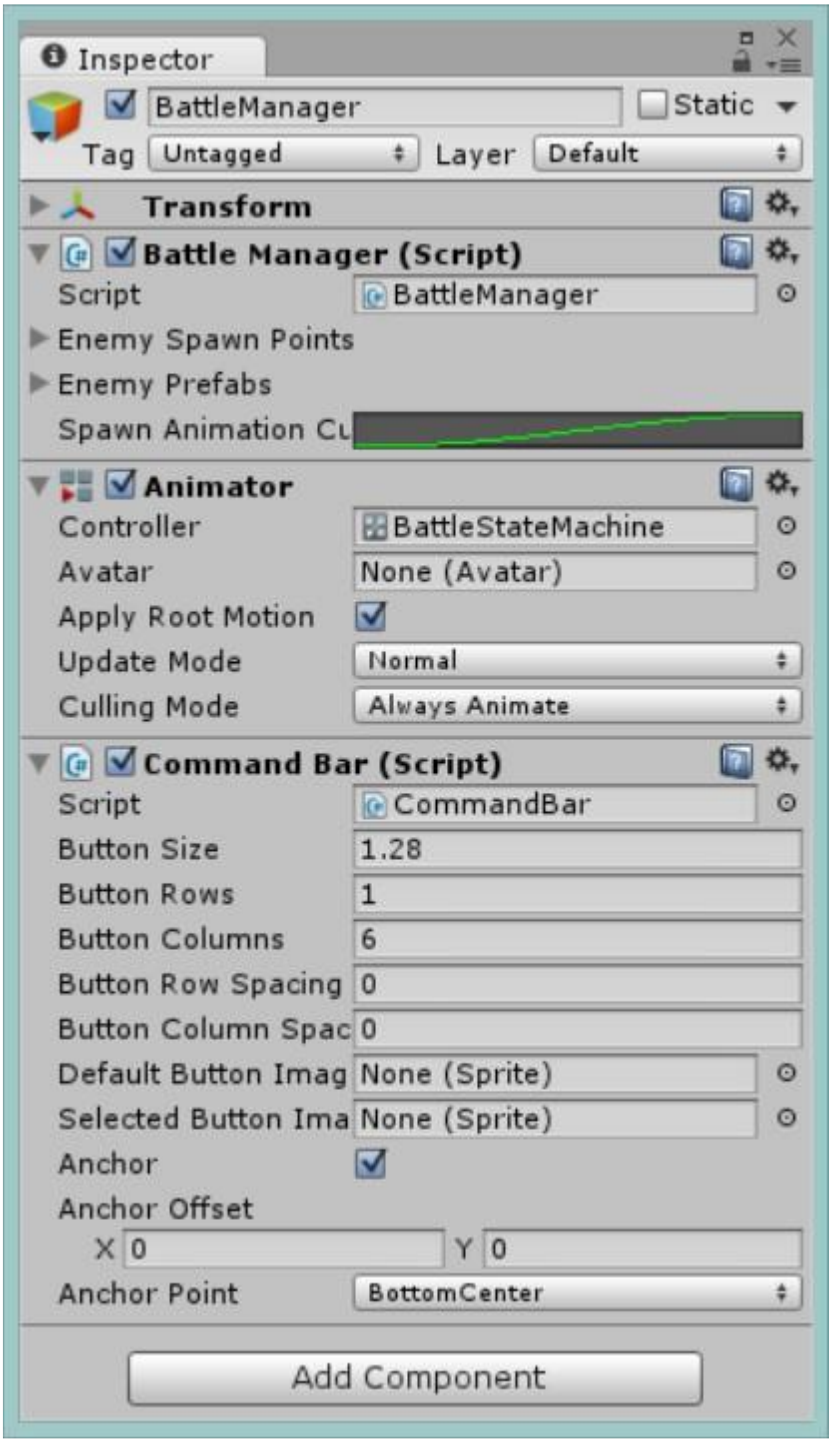
Sada dodajemo funkciju koja će dodati naše oružje u Inventory:

```
public void AddInventoryItem(InventoryItem item)
{
    this.Item = item;
    var childGO = new GameObject("InventoryItemDisplayImage");
    var renderer = childGO.AddComponent<SpriteRenderer>();
    renderer.sprite = item.Sprite;
    renderer.sortingLayerName = "GUI";
    renderer.sortingOrder = 10;
    renderer.transform.parent = this.transform;
    renderer.transform.localScale *= 4;
}
```

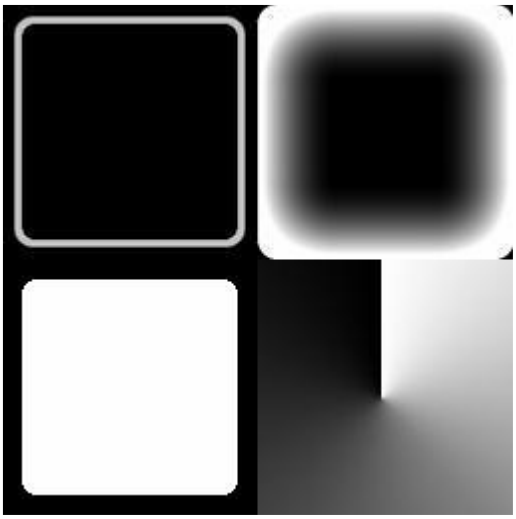
Znači kada CommandBar klasa zahteva da stavi InventoryItem, on pravi prazan objekat i dodaje sprite na njega koji će da bude prikazan.

10.5.3. Dodavanje command bar-a na scenu

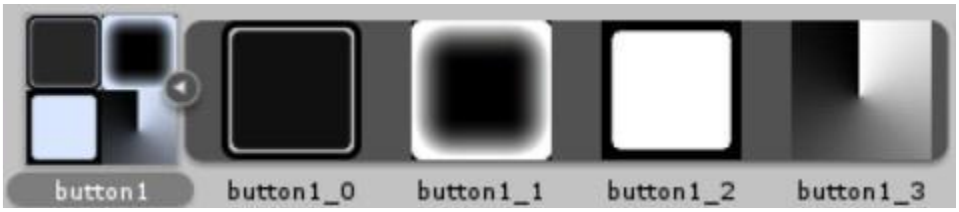
Otvorite Battle scenu ako već niste da bi mogli da dodamo command bar na scenu. Command bar koji smo napravili je deo kontrolera za borbu, pa ćemo dodati našu skriptu na BattleManager objekat.



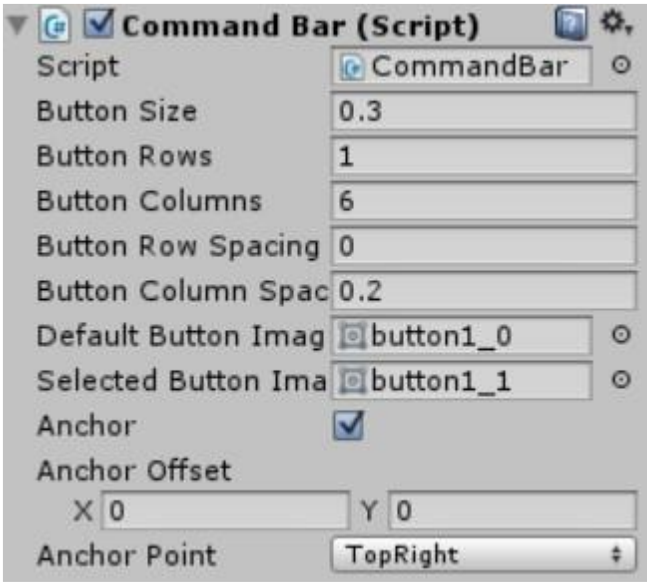
Sledeće će nam trebati slike za naše dugmiće, nađite button1 sliku u sprite-ovima, isecite je i postavite button1_0 za Defoltni sprajt, i button1_1 za selected sprite.



Isecite sprite-ove tako što ćete izabrati opciju **Grid slicing mode** i postaviti piksel velicinu za X i Y na 128, dobićete nešto ovako:



Onda se vratite na BattleManager objekat i podesite ove vrednosti:



Možete malo menjati vrednosti, videti kako to sve funkcionise.

10.6. Biranje oružja

Ukratko, šta ćemo odraditi:

- Use the OnMouseDown funkciju da detektuje klik korisnika
- Ovo će izazvati poruku da se ispise
- BattleManager će tada uzeti koje god je oružje izabrano i koristiti ga kao aktivno oružje
- Ako korisnik klikne na neko drugo dugme, pregaziće prethodno, cak iako je kliknuo na prazno dugme

10.6.1. Selektovanje komandnog dugmeta

Da bi kompletirali pricu sa dugmetom, moraćemo da dodamo funkcije koje će dozvoliti da bude kliknuto, da promeni kliknuto stanje, kao i sprajt dugmeta kada nije kliknuto i kada jeste i da se naravno skine selekcija.

Prvo ćemo dodati UpdateSelection funkciju u CommandButton skripti:

```
void UpdateSelection()
    {
        var renderer = gameObject.GetComponent<SpriteRenderer>();
        renderer.sprite = selected ? commandBar.SelectedButtonImage :
commandBar.DefaultButtonImage;
    }
```

Dakle postavljamo odgovarajući sprajt u zavisnosti da li je selektovano dugme ili nije. Sledeće ćemo dodati ClearSelection funkciju:

```
    public void ClearSelection()
    {
        selected = false;
        UpdateSelection();
    }
```

Ovde samo postavlja selected na false i poziva UpdateSelection funkciju koja će da postavi odgovarajući sprajt.

Konačno dodajemo OnMouseDown funkciju koja će reagovati na BoxCollider2D komponentu koju smo stavili na nase dugme i da primi *clicked* događaje:

```
    void OnMouseDown()
    {
        if (commandBar.CanSelectButton)
        {
            selected = !selected;
            UpdateSelection();
            commandBar.Selectbutton(selected ? this : null);
        }
    }
```

10.6.2. Upravljanje selekcijama iz Command bar-a

Sada želimo da napravimo da može samo jedno dugme da bude selektovano, a ne da možemo da selektujemo sve dugmiće. Pošto dugmići međusobno ne znaju jedan za drugog, a Command Bar zna za svu svoju decu onda ćemo morati u `CommandBar` skripti da odradimo takvu vrstu provere:

```
private CommandButton selectedButton;
public bool CanSelectButton
{
    get
       {
           return canSelectButton;
       }
}
```

Sada možemo da praitmo koje je dugme trenutno selektovano.

A pošto nam treba način da obriše trenutnu selekciju:

```
public void ResetSelection(CommandButton button)
    {
        if (selectedButton != null)
        {
            selectedButton.ClearSelection();
        }
        selectedButton = button;
    }
```

Ovo će jednostavno obrisati trenutnu selekciju i postaviti novu selekciju. U slučaju da je korisnik samo obrisao selekciju, onda će ovo biti null.

10.6.3. Updateovanje BattleManager stanja sa selekcijama

Sada kada komandni bar može da selektuje i deselektuje dugmiće, treba da kažemo BattleManager-u da je igrač izabrao oružje. Za to, koristićemo poruke. Prvo treba da dodamo u `MessagingManager` skripti novi događaj koji želimo da objavimo isto kao i pre što smo radili za Dialog i UI događaje.

Dakle otvorite `MessagingManager` skriptu i dodajte sledeći kod:

```
private List<Action<InventoryItem>> inventorySubscribers = new
List<Action<InventoryItem>>();
    // Subscribe method for Inventory manager
    public void SubscribeInventoryEvent(Action<InventoryItem> subscriber)
    {
        if (inventorySubscribers != null)
        {
            inventorySubscribers.Add(subscriber);
        }
    }
    // Broadcast method for Inventory manager
    public void BroadcastInventoryEvent(InventoryItem itemInUse)
    {
        foreach (var subscriber in inventorySubscribers)
        {
            subscriber(itemInUse);
        }
    }
    // Unsubscribe method for Inventory manager
    public void UnSubscribeInventoryEvent(Action<InventoryItem> subscriber)
    {
        if (inventorySubscribers != null)
        {
            inventorySubscribers.Remove(subscriber);
        }
    }
    // Clear subscribers method for Inventory manager
    public void ClearAllInventoryEventSubscribers()
    {
        if (inventorySubscribers != null)
        {
            inventorySubscribers.Clear();
        }
    }
```

Kada smo ovo odradili, samo treba da bradcastujemo novu poruku kada igrač selektuje neko dugme.

Dodaćemo Selectbutton metodu u CommandBar skripti da završimo:

```
    public void Selectbutton(CommandButton button)
    {
        if (selectedButton != null)
        {
            selectedButton.ClearSelection();
        }
        selectedButton = button;
        if (selectedButton != null)
        {
            MesssagingManager.Instance.BroadcastInventoryEvent(
selectedButton.Item);
        }
        else
        {
            MesssagingManager.Instance.BroadcastInventoryEvent(null);
        }
    }
```

Da završimo sa CommandBar skriptom, hajde da dodamo poruku sa UI događaja i povežemo sa canSelectbutton propertijem koji smo dodali ranije. Prvo ćemo napraviti delegat metodu:

```
    void SetCanSelectButton(bool state)
    {
        canSelectButton = !state;
    }
```

A onda prepraviti Start funkciju iz CommandBar skripte:

```
    void Start ()
    {
        InitCommandButtons();
        MesssagingManager.Instance.SubscribeUIEvent(SetCanSelectButton);
    }
```

I da ne zaboravimo da unsubscribe-ujemo događaj kada je uništen komandni bar dodavanjem funkcije:

```
    void OnDestroy()
    {
        if (MesssagingManager.Instance != null)
        {
            MesssagingManager.Instance.UnSubscribeUIEvent(SetCanSelectButton);
        }
    }
```

Sada, kada je UI zaključan, sigurno neće moći da se klikne na komandni bar, npr kada igrac napada goblina.

10.6.4. Menjanje BattleManager-ovog stanja sa oružjem

Konačno, možemo da vidimo šta ćemo sada kada izaberemo oružje. Prvo nam treba varijabla za odabrano oružje, a onda funkcija da sačuvamo to oružje koje smo izabrali:

```
    private InventoryItem selectedWeapon;

    private void InventoryItemSelect(InventoryItem item)
    {
        selectedWeapon = item;
    }
```

Onda treba da povežemo ovu funkciju sa događajem koji broadcastujem iz `BattleManager` stanja u `Start` metodi:

```
MesssagingManager.Instance.SubscribeInventoryEvent(InventoryItemSelect);
```

Sada kada znamo s čime će igrač da se bori, hajde da promenimo malo UI sa sledećim instrukcijama: Kada igrač treba da napadne, prvo ćemo tražiti da odabere oružje:

```
if (selectedWeapon == null)
{
        GUI.Box(new Rect((Screen.width / 2) - 50, 10, 100, 50), "Select Weapon");
}
break;
```

Dakle u `Intro` delu, prikazujemo igraču informacije o borbi, a kada je red na igrača da igra, onda se ispisuje **Select Weapon** poruka sve dok ne selektuje jedno oružje.

Zadatak za vežbanje: Probajte da dodate igračev i neprijateljov bar sa životom negde na ekranu koristeći trikove naučene iz command bara

Poglavlje 11. Borba

Poglavlje borbe je poprilično gomazno i moralo je da se podeli na dva dela. U prethodnom poglavlju smo ostavili igrača da je već spreman za borbu, sa oružjem u ruci (odabrano), gledajući ka goblinima i spreman da napadne. Da bi završili ovu rundu borbe, treba odabrati kojeg ćemo goblina napasti.

U ovom poglavlju, prećićemo:

- Šta znači boriti se
- Planiranje za dalje (dugovečnosti)
- Poboljšanje neprijateljske veštačke inteligencije
- “Particle systems“ u 2D-u
- Jos po nešto o animacijama

11.1. Poligon

Borba može biti raznolika, u realnom vremenu, hack and slash, konstantne akcije za strategijske bazirane sisteme borbe i slično. U zavisnosti od pravca koji smo izabrali u kom žanru će naša igrica biti će različito uticati na različitu publiku. U većini slučajeva, borba će biti ta koja će biti glavni deo vaše igrice (pored priče naravno), što znači, ako napravimo borbu da bude zanimljiva, sama igrica će biti zanimljiva.

Bez obzira na pravac koji smo izabrali, neprijatelji koje će igras sresti moraju biti izazovni za igrača u svakom trenutku. To samo znači da igra mora biti osmišljena na taj način da se igrač uvek oseća izazvanim dok igra. Dakle, ako je igrica previse teška, verovatno neće imati dovoljno volje da prelazi, a ako je previše laka, sva ta volja se izubi jer se ne oseća izazvanim.

Postoje nekoliko stvari koje bi trebalo mi kao programeri da isplaniramo, a to su:

- Animacije
- Akcije igrača
- Neprijateljova odbrana i reakcije
- Posebni potezi
- Interakcije

Svaka od navedenih stavki predstavlja to iskustvo sukoba zanimljiviji igraču i samim tim borba će postati zanimljivija i vrednija igranja. Ako se ne fokusiramo na svaku od navedenih stavki i napravimo da to sve lepo izgleda i funkcioniše, igraču će možda igra postati dosadna.

11.2. Leveling up – progres sa nivoima

Iako nije uvek krucijalno da igra ima određenu priču ili napredak za sledeću oblast ili nivo koji treba da se igra, u većini slučajeva je jako bitno da se obezbedi da igrač stekne neki utisak da nešto postiže kako napreduje u igrici. Ovo može biti koliko je zlata skupio od ubijenih neprijatelja kako bi mogao da kupi neki vatreni mač koji zahteva da budete 20. lvl npr, ili da može da se unapredi statistika ili same sposobnosti igrača što bi im omogućavalo da mogu da napadnu neke neprijatelje većih i jačih sposobnosti nekim posebnim udarcima i slično.

Miks ovih stvari je upravo ono što može istaći vašu igricu. Najviše se vremena provodi na planiranje ovih stvari, na planiranju kakve sposobnosti da mogu da unaprede, ili koja oružja, a uzgred da sve to bude dobro balansirano. I to je jako bitno, često je bitnije i od same borbe, baš to planiranje šta se dešava sa igrom kada se izađe iz borbe.

11.3. Balansiranje

Ubedljivo najteža stvar za implementiranje u svakoj igrici je upravo balanisranje. Ako se odradi kako treba, oko 50% vremena će biti potrošeno na testiranje, prepravljanje, ponovo testiranje i ponovo prepravljanje igrice.

Ne bi trebalo da samo jedna grupa ljudi uvek testira istu stvar, najbolje je dati ljudima iz svih pravaca života. Uvek davati deci, publici, čak ženi, mužu, partneru da igra vašu igricu, jer feedback odnosno njihov utisak je veoma bitan komentar. Što više ljudi igra igricu, bolji balans se može napraviti same igrice.

Naći zlatnu sredinu između teškog, moguće igranog, zabavnog i izazovnog je uvek teško, tako da nikako ga ne treba izostaviti ili zanemariti. I zapamtite, ako vi igrate igricu na neki vaš poseban način, ne znači da će svi igrati tu igricu na vaš zanimljiv način.

I sada, kada smo sve to objasnili kako bi igrica trebalo da izgleda, možemo konačno nastaviti dalje sa razvojem naše borbe.

11.4. Spremanje `BattleManager` skripte

Kako se spremamo da napadnemo našeg neprijatelja, prvo što treba da odradimo jeste da omogućimo našem igraču da klikne na neprijatelja kojeg želi da napadne.

Dakle otvorite `BattleManager` skriptu i dodajte sledeće varijable na vrh klase:

```
private string selectedTargetName;
private EnemyController selectedTarget;
public GameObject selectionCircle;
private bool canSelectEnemy;
bool attacking = false;
public bool CanSelectEnemy
{
    get
        {
            return canSelectEnemy;
        }
}
public int EnemyCount
{
    get
        {
            return enemyCount;
        }
}
```

Dakle, dodali smo propertije da vidimo koga smo selektovali, EnemyController skripta će se tek napraviti, selectionCircle ćemo takođe implementirati kasnije, to služi da bi postojao krug ispod goblina kada selektujemo nekog, i da li može da se selektuje neprijatelj u tom trenutku ili ne (nećemo dozvoliti da se selektuje neprijatelj ako nismo odabrali oružje).

11.5. Pojačavanje neprijatelja

U ovom trenutku, samo postoje goblini kao neprijatelji. Sada ćemo dodati jos neke neprijatelje, ćisto da goblini ne budu jedini.

11.5.1. Profil neprijatelja/kontroler

Prvo ćemo nove profile za neprijatelje, počećemo sa enumeracijom za Enemy klasu. Napravite novu C# skriptu i nazovite je EnemyClass i zamenite njen kod sa sledećim:

```
public enum EnemyClass
{
    Goblin,
    Ork,
    NastyPeiceOfWork
}
```

A onda napravite Enemy C# skriptu i istom folderu:

```
public class Enemy : Entity
{
    public EnemyClass Class;
}
```

A sada, treba napraviti skriptu koja će biti kontroler za našeg neprijatelja. Dakle pravimo C# skriptu EnemyController:

```
using System.Collections;
using UnityEngine;

public class EnemyController : MonoBehaviour {
    private BattleManager battleManager;
    public Enemy EnemyProfile;
    Animator enemyAI;
    public BattleManager BattleManager
    {
        get
        {
            return battleManager;
        }
```

```
        set
        {
            battleManager = value;
        }
    }
}
```

Dakle imamo reference na BattleManager koja nam je potrebna zato što će uticati na neprijatelje u toku borbe, imamo profil neprijatelja i referencu na AI animator kontroler koji smo napravili jos ranije.

Pošto AI-u su potrebne informacije o borbi, moramo da obezbedimo da ga prati u svakom frejmu, tako da dodajemo novu metodu updateAI i pozivamo je u Update metodi:

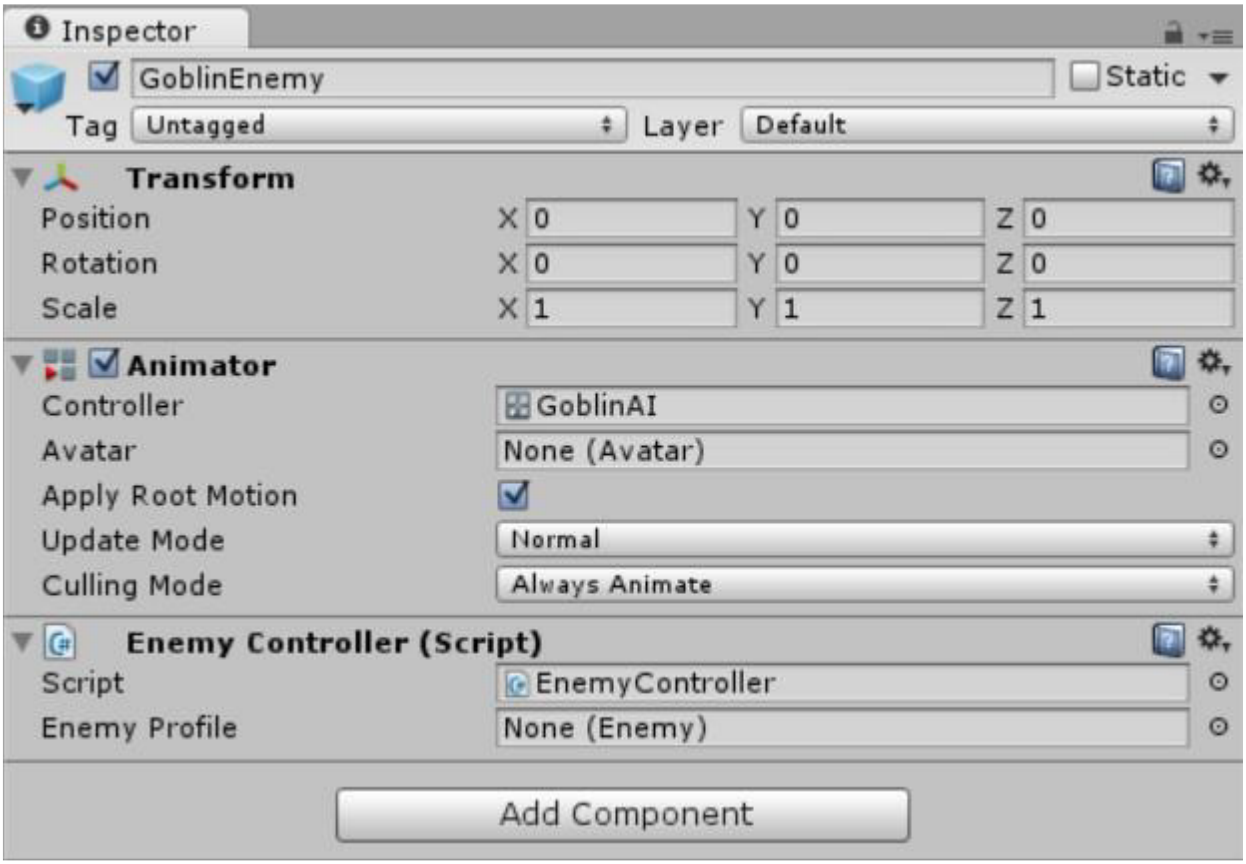
```
void Update()
{
    UpdateAI();
}
public void UpdateAI()
{
    if (enemyAI != null && EnemyProfile != null)
    {
        enemyAI.SetInteger("EnemyHealth", EnemyProfile.Health);
        enemyAI.SetInteger("PlayerHealth", GameState.CurrentPlayer.Health);
        enemyAI.SetInteger("EnemiesInBattle", battleManager.EnemyCount);
    }
}
```

Ovaj kod samo setuje vrednosti iz enemyAI po svakom frejmu, a AI će reagovati ako dođe do promena po njegovim definisanim pravilima. Da bi nam ovo radilo, moramo samo uhvatiti enemyAI kontroller u Awake funkciji:

```
void Awake()
{
    enemyAI = GetComponent<Animator>();
    if (enemyAI == null)
    {
        Debug.LogError("No AI System Found");
    }
}
```

11.5.2. Unapređivanje Goblin prefab-a

Sada treba samo da dodamo našu novo napravljenu skriptu EnemyController na prefab goblina. Dakle selektujte Goblin prefab i prevucite mu EnemyController skriptu.



11.5.3. Postavljanje profila neprijatelja u kodu

Vraćamo se nazad u BattleManager skriptu i sada, u delu gde smo ubacili gobline na scenu, sada umesto da ih samo dodamo, ubacićemo i malo opasnosti sa njihove strane na sledeći način:

```
IEnumerator SpawnEnemies()
{
    // pravi gobline koji se krecu ka svojim spawn pozicijama, dolazeci van
ekrana
    for (int i = 0; i < enemyCount; i++)
    {
        var newEnemy = (GameObject)Instantiate(EnemyPrefabs[0]); // novi
protivnik
        newEnemy.transform.position = new Vector3(10, -3, 0); // setuje se
pocetna pozicija van ekrana
        yield return StartCoroutine(MoveCharacterToPoint(EnemySpawnPoints[i],
newEnemy)); // goblin dolazi na spawn poziciju
        newEnemy.transform.parent = EnemySpawnPoints[i].transform;

        var controller = newEnemy.GetComponent<EnemyController>();

        controller.BattleManager = this;

        var EnemyProfile = ScriptableObject.CreateInstance<Enemy>();
        EnemyProfile.Class = EnemyClass.Goblin;
        EnemyProfile.Level = 1;
        EnemyProfile.Damage = 1;
        EnemyProfile.Health = 2;
        EnemyProfile.Name = EnemyProfile.Class + " " + i.ToString();

        controller.EnemyProfile = EnemyProfile;
    }
    battleStateManager.SetBool("BattleReady", true);
}
```

Dakle, dok prolazimo kroz neprijatelje da ih postavljamo na scenu, uzimamo EnemyController klasu zakačenu na Goblin prefab, pravimo novi EnemyProfile klasu, dajemo mu neke vrednosti i na kraju konačno inicijalizujemo u kontroleru našeg novog EnemyProfile-a.

Sada kada imamo jakog protivnika, hajde da započnemo napad.

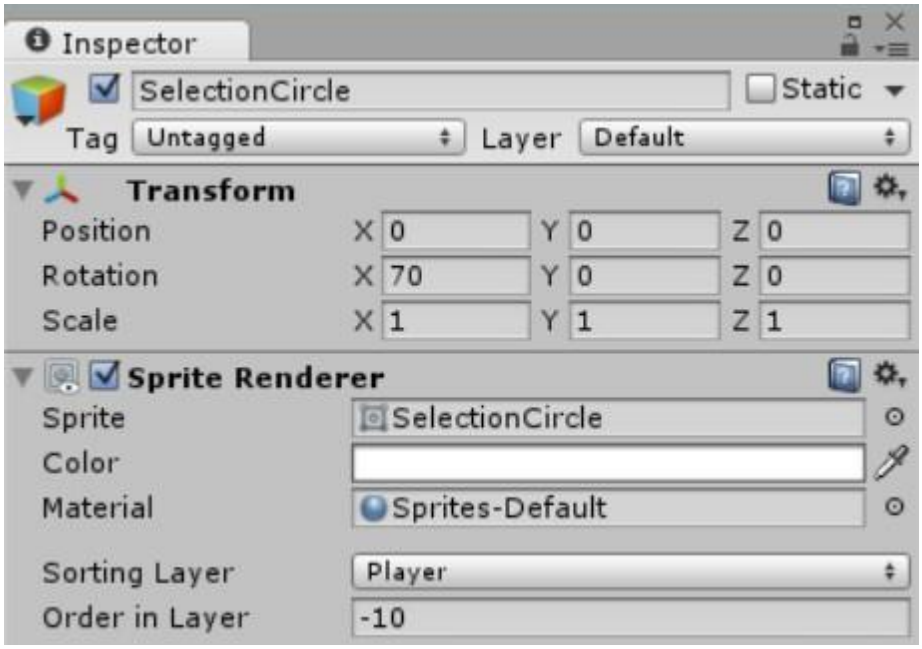
11.6. Selektovanje neprijatelja

Kao sa CommandBar-om, igrač treba da ima neku vizuelnu reprezentaciju da potvrdi da njegove akcije imaju zapravo nekog uticaja i efekta u igri. Da bi to odradili, samo treba da ubacimo nešto malo logike za selektovanje i lep vizuelni efekat u 2Du. Prvo ćemo napraviti prefab za ovo sa malom animacijom, a onda prikačiti ga na varijablu iz BattleManager skripte koju smo dodali ranije.

11.6.1. Prefab za krug za selekciju

Prvo ćemo ubaciti novi sprite `SelectionCircle.png` u folder za sprajtove. Nakon toga, napravićemo prefab od ovog sprajta, a pošto ćemo ga koristiti nekoliko puta u sceni, to samo znači da ćemo koristiti različite kopije jedne instance.

Sada prevucite `SelectionCircle.png` sliku negde na scenu i postavite podešavanja kao na slici.



Kada ovo odradite, samo prevucite objekat sa Scene hijararhije u Prefabs folder da napravite novi prefab i nazovite ga `SelectionCircle`. Na kraju obrišite objekat sa scene jer nam više nije potreban.

A sada, u Battle sceni, u editoru, selektujte **BattleManager** objekat i prevucite **SelectionCircle** prefab na **SelectionCirce** properti iz **BattleManager skripte** da ga nalepi na **BattleManager** objekat.

11.6.2. Dodavanje logike za selektovanje u EnemyController klasi

Kada smo to odradili, sada možemo da se vratimo u `EnemyController` skriptu i da sredimo funkcionalnost da igrač može da klikne na goblina i da ga označi. Prvo nam trebaju varijable da proverimo da li je neprijatelj selektovan ili nije, znači na vrh `EnemyController` klase dodajemo:

```
private bool selected;
GameObject selectionCircle;
```

Sada, da bi oživeli malo proces selektovanja postavićemo da se krug rotira. Da bi to odradili, dodaćemo prostu korutinu da se konstantno update-uje i rotira naš selection circle. Dakle u `EnemyController` skripti dodajemo:

```
IEnumerator SpinObject(GameObject target)
{
    while (true)
    {
        target.transform.Rotate(0, 0, 180 * Time.deltaTime);
        yield return null;
    }
}
```

Sledeće, kada igrač klikne, koristićemo kombinaciju **`BoxCollider2D`** i **`OnMouseDown`** funkcije da selektuje Goblina i pokaže taj krug.Dakle dodajemo BoxCollider2D komponentu na **Goblin** prefab i dodajemo sledeću funkciju u EnemyController skriptu:

```
void OnMouseDown()
{
    if (battleManager.CanSelectEnemy)
    {
        var selection = !selected;
        battleManager.ClearSelectedEnemy();
        selected = selection;
        if (selected)
        {
            selectionCircle =
            (GameObject)GameObject.Instantiate(battleManager.selectionCircle);
            selectionCircle.transform.parent = transform;
            selectionCircle.transform.localPosition = Vector3.zero;
            StartCoroutine("SpinObject", selectionCircle);
            battleManager.SelectEnemy(this, EnemyProfile.Name);
        }
    }
}
```

Dakle, šta se ovde dešava zapravo. Prvo što radimo jeste cuvamo trenutno stanje selektovanog Goblina (ako kliknemo dva puta na isti, skida se selekcija), osiguramo da ni jedan drugi Goblin nije selektovan. Nove funkcije trenutno ne postoje u BattleManager skripti, tako da ćemo ih posle napraviti.

Kao sa CommandButtons, trebaće nam funkcija da obriše selekcije sa neprijatelja ako je to traženo, pa ćemo dodati sledeću fuinkciju:

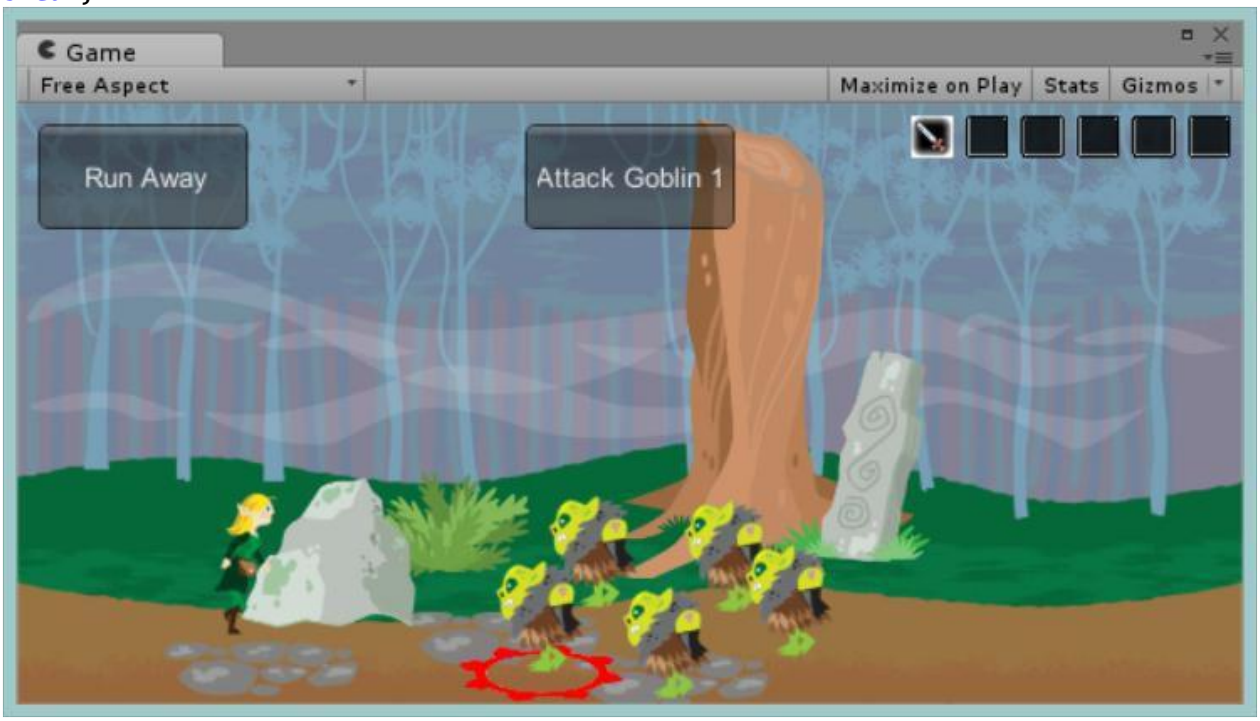
```
public void ClearSelection()
{
    if (selected)
    {
        selected = false;
        if (selectionCircle != null)
            DestroyObject(selectionCircle);

        StopCoroutine("SpinObject");
    }
}
```

Zasad smo završili sa EnemyController-om. Da bi završili sa logikom selekcije, vraćamo se u BattleManager skriptu i dodajemo dve funkcije koje nam fale:

```
public void SelectEnemy(EnemyController enemy, string name)
{
    selectedTarget = enemy;
    selectedTargetName = name;
}
public void ClearSelectedEnemy()
{
    if (selectedTarget != null)
    {
        var enemyController = selectedTarget.GetComponent<EnemyController>();
        enemyController.ClearSelection();
        selectedTarget = null;
        selectedTargetName = string.Empty;
    }
}
```

Obe funkcije su relativno proste, prva samo setuje vrednosti koje smo im poslali, a druga uzima kontroler sa tog goblina kojeg smo kliknuli, poziva funkciju da obrise selekciju, postavlja target na null i targetName na prazan string, u suštini samo obriše sve kao da ništa nije selektovano. Međutim još uvek ne možemo da napadnemo jer nismo to omogućili u BattleManager skripti. Da bi kontrolisali igrača i tok borbe, hoćemo da dozvolimo da igrač napadne samo kada je izabrao oružje, ako nije onda neće moći ni da klikne na goblina. Da bi ovo odradili, promenićemo jedan case u okviru OnGUI metode:

```
case BattleState.Player_Move:
if (GUI.Button(new Rect(10, 10, 100, 50), "Run Away"))
{
    GameState.playerReturningHome = true;
    NavigationMenager.NavigateTo("World");
}
if (selectedWeapon == null)
{
    GUI.Box(new Rect((Screen.width / 2) - 50, 10, 100, 50), "Select Weapon");
}
else if (selectedTarget == null)
{
    GUI.Box(new Rect((Screen.width / 2) - 50, 10, 100, 50), "Select Target");
    canSelectEnemy = true;
}
else
{
    if (GUI.Button(new Rect((Screen.width / 2) - 50, 10, 100, 50),
    "Attack " + selectedTargetName))
    {
        canSelectEnemy = false;
        battleStateManager.SetBool("PlayerReady", true);
        MessagingManager.Instance.BroadcastUIEvent(true);
    }
}
break;
```



11.6. Napad!

Sada kada smo već sve spremili za naš napad, hajde da implementiramo funkciju koja će to da odradi. Dakle dodajemo novu funkciju u `BattleManager` skripti:

```
IEnumerator AttackTarget()
{
    int Attacks = 0;
    attacking = true;
    bool attackComplete = false;
    while (!attackComplete)
    {
        GameState.CurrentPlayer.Attack(selectedTarget.EnemyProfile);
        selectedTarget.UpdateAI();
        Attacks++;
        if (selectedTarget.EnemyProfile.Health < 1 || Attacks >
        GameState.CurrentPlayer.NoOfAttacks)
        {
            attackComplete = true;
        }
        yield return new WaitForSeconds(1);
    }
}
```

Šta radi ovaj kod:

1. Postavlja inicijalno stanje za borbu. Obaveštava nas koliko još poteza nam ostaje da napadnemo, činjenicu da napadamo, i činjenicu da napad još uvek nije završen.
2. Onda, dok ne završimo, nastavljamo da napadamo
    a. Pozivamo Attack funkciju za igrača protiv selektovanog neprijatelja
    b. Update-ujemo AI stanje za selektovanog neprijatelja
3. Ako je neprijatelj mrtav, ili igrač ostane bez udaraca, označiti borbu da je gotova
4. Sačekati kraj frejma da napadne opet ili da se zavrsi petlja

Sve što ostaje je da se pozove ova funkcija kada igrač klikne Attack dugme u `Update` metodi **BattleManager** objekta. Dakle treba da prepravimo `case BattleState.PlayerAttack` na sledeći način:

```
case BattleState.Player_Attack:

if (!attacking)
{
    StartCoroutine(AttackTarget());
}
break;
```

11.7. Reagovanje Goblina sa 3D česticama

Igrač je konačno odradio svoj potez i to je uticalo na Goblina na neki način, ali trebalo bi da se prikaže da bi se stekao utisak o tome da se napad odradio.

U ovom slučaju, igrač ima mač (ili sekiru) koja ima jačinu od 5, dok je jačina igrača 1, znači ukupna jačina igrača sa mačem je 6. Posto Goblini su bili previpe lenji da nose neki štit ili nešto što ga može sačuvati, a njegov život (health) je 1, što znači da dobijamo formulu

Zivot: 1 – Jačina napada 6 = Mrtav

Sve ovo nas dovodi do jednog dela, a to je upoznavanje sa česticama u Unity-u, tj Particles. Da bi se animacija smrti lepo prikazala, mi ćemo dodati efekat čestica kada je Goblin ubijen zajedno sa jos jednom animacijom.

11.7.1. Dodavanje sprite-ova za animaciju smrti

Dodati spritove BloodSplat.png i Tombstone.png u Assets\Sprites.

Poželjno je praviti posebne foldere za sprite-ove, pa kada ubacujete sprite-ove da svaki sprajt ide u folder sa svojim značenjem.

Kliknite na Tombstone sprite i postavite **Pivot** na **Bottom**

11.7.2. Pravljenje materijala za efekat čestica

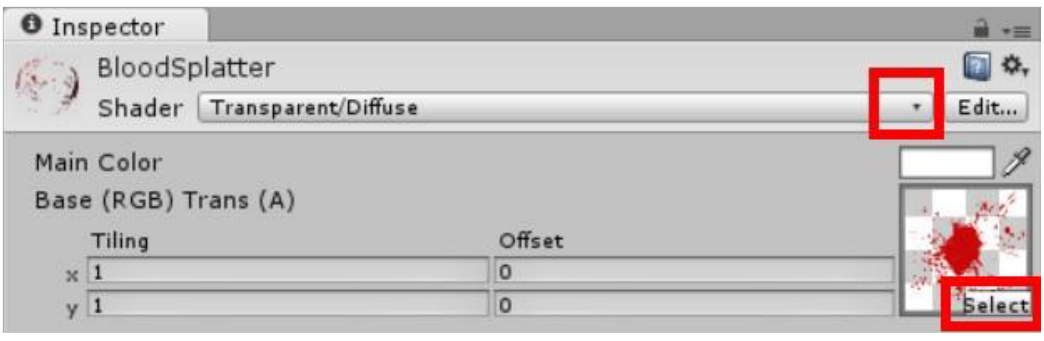
Da bi efekat čestica radio, moramo da definišemo materijal, da ne bi uzimali gotovu teksturu ili sprite. Dakle napravite novi materijal u `Assets\Materials` i nazovite ga `BloodSplatter`.

Sledeće, kliknite na **Select** dugme u inspektoru za vaš BloodSplatter materijal i izaberite sprite koji smo dodali, bloodSplat.png.

A onda, promenite **shader** materijala da koristi **Transparent/Diffuse** shader tako što kliknete na **Shader** properti dropdown i pronađete **Transparent | Diffuse**.

(U novijim verzijama Unity-a treba prvo odabrati Shader, a **Transparent | Diffuse** se može naći u:

Legasy Shaders – Transparent – Diffuse



11.7.3. Rekonstrukcija Goblin prefab-a

Sada, zbog načina na koji rade 2D animacije, animirati dete objekta iz roditelja je ok, animiranje dete deteta objekta ne radi i takođe animiranje roditelja i deteta ne radi.

Ovo nije veliki problem, samo treba unapred da se planira kada kreirate objekte da će imati više sprite-ova ili objekata koji će zajedno da reaguju na istoj animaciji.

Da bi počeli, krećemo da odradimo sledeće korake:

- Napraviti prazan objekat na sceni i nazovite ga `GoblinEnemy`
- Prevući postojeći **Goblin** prefab na **GoblinEnemy** objekat kao njegovo dete i resetujte transform na novom detetu
- Obrišite `Animator`, `EnemyController` i `BoxCollider2D` sa starog **Goblin** prefab-a i dodajte na **GoblinEnemy** tako što ćete ih isto podesiti kao na starom prefabu.
- Proveriti da li je `BoxColider2D` na `GoblinEnemy` lepo pozicioniran preko goblina
- Prevući Tombstone sprite na GoblinEnemy objekat. Postaviti transfor pozicije sve na 0, podesiti Scale transform na **X: 0.2** i **Y na 0.2** i postaviti **Sorting Layer** na `player`
- Podesiti Boju na Tombstone SpriteRenderer, gde je A (alpha) na 0.
- Napraviti još jedan prazan objekat i nazovite ga `BloodParticles`, prevući ga na `GoblinEnemy` kao dete, i obavezno postavite sve transform pozicije na 0.
- Obrišite stari prefab, pošto nam vise neće trebati

11.7.4. Dodavanje čestica

Kliknite na BloodParticles objekat i dodajte novi sistem za čestice tako što ćete ići na **Add Component** i odabrati Particle System. Odmah ćemo primetiti problem.
Kada se čestice renderuju, one se uvek renderuju iza 2D pogleda i ne postoje nikakvih dodatnih podešavanja u editoru. Da bi ovo sredili, moraćemo da napravimo skriptu za sistem čestica ili kada pustimo taj sistem čestica u kodu. Mi ćemo napraviti odmah za sistem čestica jednu skriptu i nastaviti da radimo sa tim. Dakle napravite novu skriptu i nazovite je ParticleSortingLayer i zamenite kod sa sledećim:

```
using UnityEngine;
[ExecuteInEditMode]
public class ParticleSortingLayer : MonoBehaviour
{
    void Awake()
    {
        var particleRenderer = GetComponent<Renderer>();
        particleRenderer.sortingLayerName = "GUI";
    }
}
```

Ovde, skripta jednostavno podešava sorting layer za naš sistem čestica tj postavlja ga na “GUI“.

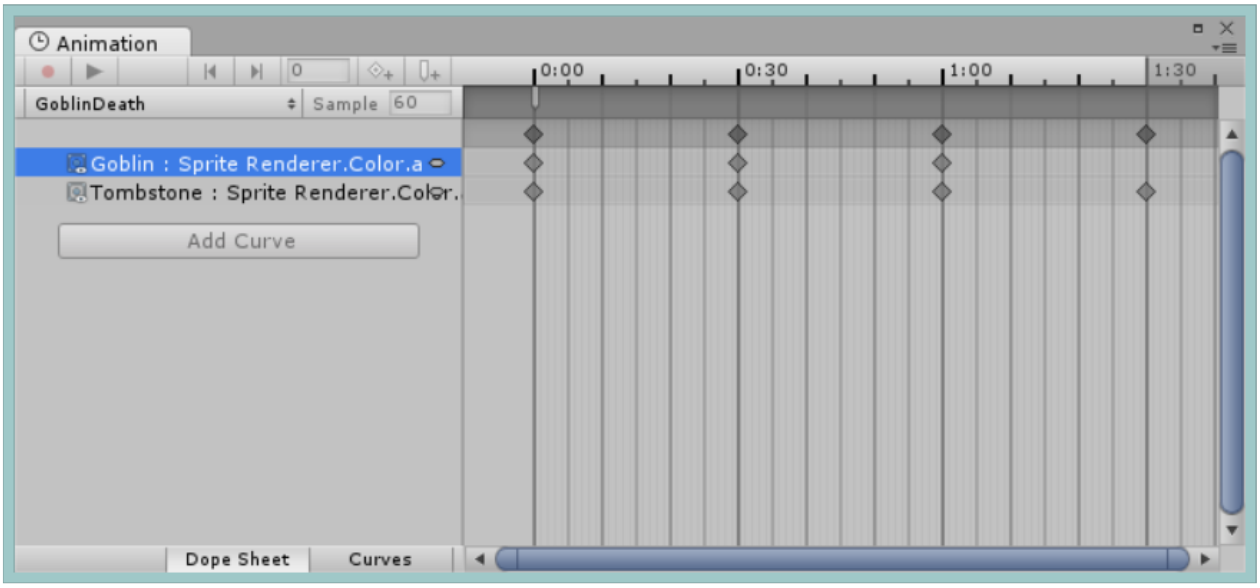
Sada samo prevucite skriptu na nas BloodParticles objekat. Sada treba da ga podesimo za igru, a podšavanja su sledeća:

- **Name**: BloodParticles
- **Rotation**: X: -90
- **Duration**: 1.00
- **Looping**: false (unchecked)
- **Start lifetime**: 0.5
- **Speed**: 2 / 3 (random between two constants)
- **Start Rotation**: 10 / 50 (random between two constants)
- **Gravity Multiplier**: 0.5
- **Inherit Velocity**: 200
- **Play On Awake**: false (unchecked)
- **Shape**: Radius: 0.2
- **Size over Lifetime**: progressive curve
- **Renderer**: Material: BloodSplatter material

11.7.5. Animacija za smrt

A sada, kada smo sve spremili, hajde da napravimo novu animaciju i da je dodamo našem goblinAI-u.

Prvo ćemo kliknuti na naš objekat `GoblinEnemy`, a zatim uključiti Animacioni tab (**Window – Animation** u meni-u). A onda, kliknuti na **Clip** dropdown i selektovati **[Create New Clip].** Kada se otvori, sačuvati novi klip u folderu `Assets\Animation\Clips` i nazvati ga `GoblinDeath`.



A onda, odradite sledeće korake:

1. Kliknite na 0:00 da bi selektovali startnu tačku
2. Proverite da li je record dugme uključeno
3. Selektujte **Goblin** objekat iz hijararhije
4. Otvorite **Color** editor za SpriteRenderer i promenite A (alpha) vrednost, nebitno na šta, samo da bi dodalo properti za animacionu krivu
5. Na poziciji 0:00 postaviti vrednost za **Goblin: SpriteRenderer.Color.a** na 1
6. Na poziciji 0:30 postaviti vrednost za **Goblin: SpriteRenderer.Color.a** na 0.5
7. Na poziciji 1:00 postaviti vrednost za **Goblin: SpriteRenderer.Color.a** na 0
8. Selektujte opet poziciju 0:00
9. Selektujte Tombstone objekat iz hijararhije
10. Otvorite Color editor za SpriteRenderer i promenite vrednost A (alpha) na bilo šta
11. Na poziciji 0:00 postavite vrednost **Tombstone: SpriteRenderer.Color.a** na 0
12. Na poziciji 0:30 postavite vrednost **Tombstone: SpriteRenderer.Color.a** na 0 (postavite na 1 da bi zapamtilo, pa vratite na nulu)
13. Na poziciji 1:00 postavite vrednost **Tombstone: SpriteRenderer.Color.a** na 0.5
14. Na poziciji 1:30 postavite vrednost **Tombstone: SpriteRenderer.Color.a** na 1

Sa ovim odrađenim, napravili smo animaciju da kada ubijemo goblina, kreće animacija nestajanja goblina i iza njega ostaje njegov grob tj Tombstone.

11.7.6. Dodavanje čestica animaciji

Da bi se čestice uključile tačno kada je potrebno, a to je kada ubijemo goblina, potrebno je dodati **Animation** događaj na naš timeline. Kada dođe do tog događaja, pozvaćemo funkciju koju ćemo napraviti.

To ćemo odraditi na sledeći način, prvo ćemo dodati na vrh klase `EnemyController` novu varijablu:

```
private ParticleSystem bloodsplatterParticles;
```

A onda, hvatamo referencu našeg sistema čestica u `Awake()` metodi:

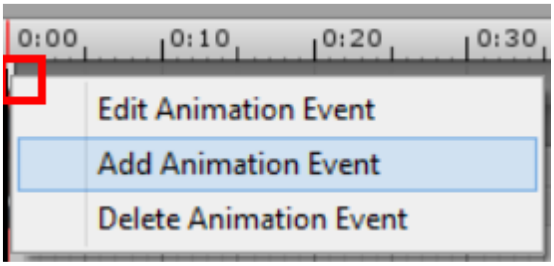
```
void Awake()
{
    bloodsplatterParticles = GetComponentInChildren<ParticleSystem>();
    if (bloodsplatterParticles == null)
    {
        Debug.LogError("No Particle System Found");
    }
    enemyAI = GetComponent<Animator>();
```

```
    if (enemyAI == null)
    {
        Debug.LogError("No AI System Found");
    }
}
```

A onda pravimo funkciju koja će uključiti naš sistem čestica da se prikaže (i obrisati selektovanog neprijatelja sa scene). Dakle udjemo u `BattleManager` skriptu:

```
void ShowBloodSplatter()
{
    bloodsplatterParticles.Play();
    ClearSelection();
    if (battleManager != null)
    {
        battleManager.ClearSelectedEnemy();
    }
    else
    {
        Debug.LogError("No BattleManager");
    }
}
```

Kada smo ovo odradili, vraćamo se na **Animation** pogled, desni klik na 0:00 u tamno sivom delu timeline-a i selektujte **Add Animation Event** kao što je prikazano na screenshot-u:



Kada ovo odradite, pojaviće se sa desne strane prozor **Edit Animation Event** kao što je prikazano na slici, i za funkciju izaberite `ShowBloodSplater()`.

Sada kada krene animacija, osim animacije takođe će se uključiti i naš sistem čestica koji smo namestili i to već izgleda kao prava animacija smrti.

11.7.7. Povezivanje celine

Ako sada selektujete **GoblinEnemy** objekat iz hijararhije i otvorite **Animator** tab, videćete novi klip animacije kao novo stanje u **GoblinAI**-u.

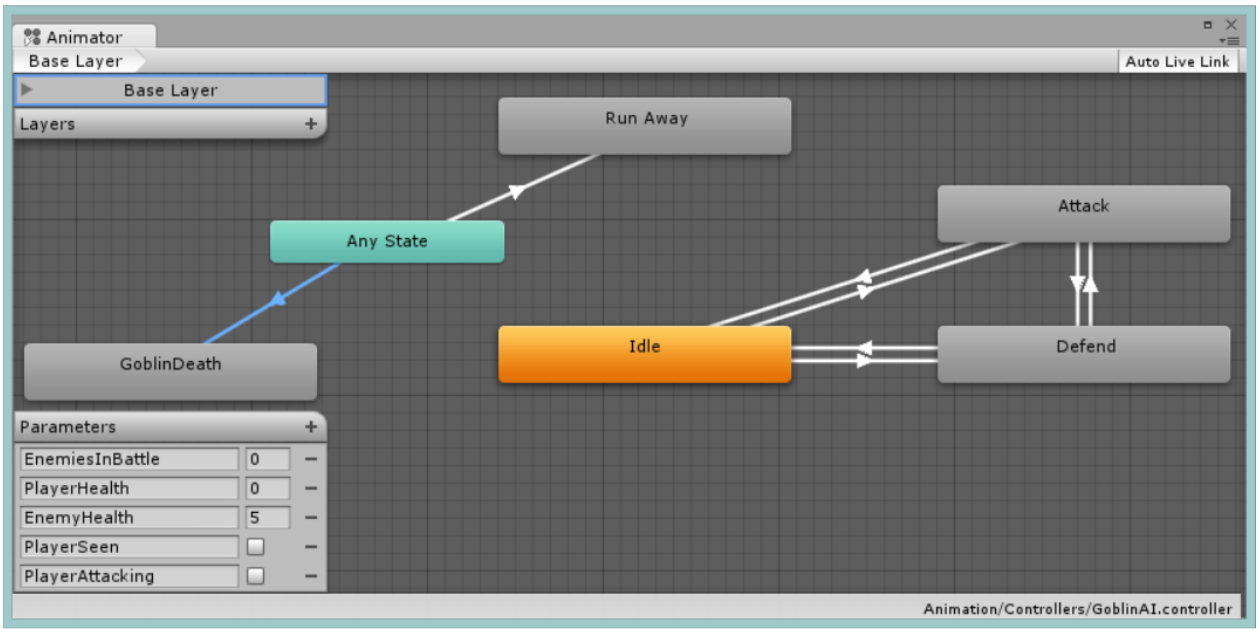
Međutim, nije povezano ni sa čim za sad, a pošto mi želimo da se pokrene ta animacija kada ubijemo goblina, sve što treba da uradimo jeste da povežemo sa **Any** stanjem kada Goblinov život padne na 0 ili ispod.

Dakle, desni klik na **Any** stanje i pravimo novu tranziciju i povežemo je sa **GoblinDeath** stanjem. Onda, setujemo uslov za tranziciju da se desi samo ako je **EnemyHealth** manje od 1.

Da bi sredili malo stvari, treba odraditi sledeće:

1. Obrisati tranziciju izmedju Idle i Any stanja
2. Dodati novu tranziciju **Attack -> Idle** sa uslovom **PlayerSeen = false**
3. Dodati novu tranziciju **Defend -> Idle** sa uslovom **PlayerSeen = false**

Konačni GoblinAI animator bi trebalo da izgleda ovako:



11.7.8. Pravljenje novog GoblinEnemy objekta u prefab i dodavanje u borbu

Sada kada smo napravili novu bazu za našeg Goblina, prevući ćemo GoblinEnemy objekat u Assets\Prefabs (obrišite stari prefab ako niste do sad). Kada napravite prefab, obrisite objekat iz hijararhije pošto ćemo koristiti samo prefab od sad.

Proverite još jednom da li su sve pozicije na 0,0,0.

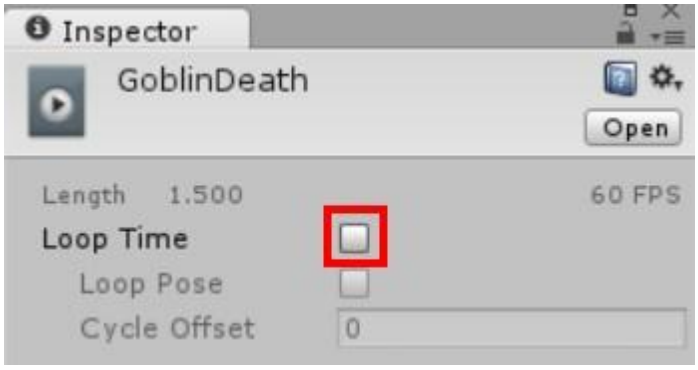
Sada kliknemo na BattleManager objekat i dodajemo novi prefab u EnemyPrefabs properti iz BattleManager skripte, tj samo menjamo onaj koji smo obrisali.

11.7.9. Animacije, stanja i petlje

Sada kada igrica dođe do dela kada treba da se pusti animacija postoji problem, animacija se ne završava. Da bi znali kako da resimo taj problem, treba nam malo više znanja o novom sistemu za animiranje, a to je da po defaultu, sve animacije se zauvek animiraju.

Da bi se zaustavila animacija da se vrti beskonačno, sve što treba da se odradi jeste samo da pređe na neko drugo stanje animiranja u Mecanim animatoru kada se završi prethodno. Međutim, u našem slučaju se animacija nalazi u završnoj fazi i pošto nema na šta sledeće da nastavi, onda će samo vrteti jednu istu. Da bi se to sredilo, sve što treba da odradimo jeste da isključimo da se animacija smrti vrti zauvek, a to radimo na sledeći način.

Dakle uđemo u `Assets\Animation\Clips` i selektujte **GoblinDeath** animacioni klip. Onda u **Inspector** prozoru samo skinite da **Loop Time** bude štiklirano kao na sledećoj slici. I to je sve.



11.8. Šta je sve odrađeno

Sada kada pokrenemo naš projekat u Battle sceni nećemo mnogo šta odraditi. Dakle moramo prvo da se vratimo na glavnu scenu i da pokrenemo igricu ispočetka. Šta je sve omogućeno za sada:

1. Krećemo od početka
2. Idemo do shop-a da kupimo oružje
3. Izlazimo iz shop-a i idemo u Veliki los svet
4. Šetamo po svetu dok ne naiđemo na neprijatelje
5. Biramo oružje s kojim ćemo napasti
6. Biramo Goblina kojeg ćemo napasti
7. I klik na Attack dugme da napadnemo

Kada se sve ovo odradi, onda možemo videti našu celu animaciju kako se odvija.

Zadaci za vežbanje:

1. Probajte da implementirate da Goblini mogu da napadnu igrača
2. Probajte da ubacite neko zlato ili experience u igru kada ubijete neprijatelja
3. Vraćanje nazad na svet kada se borba završi