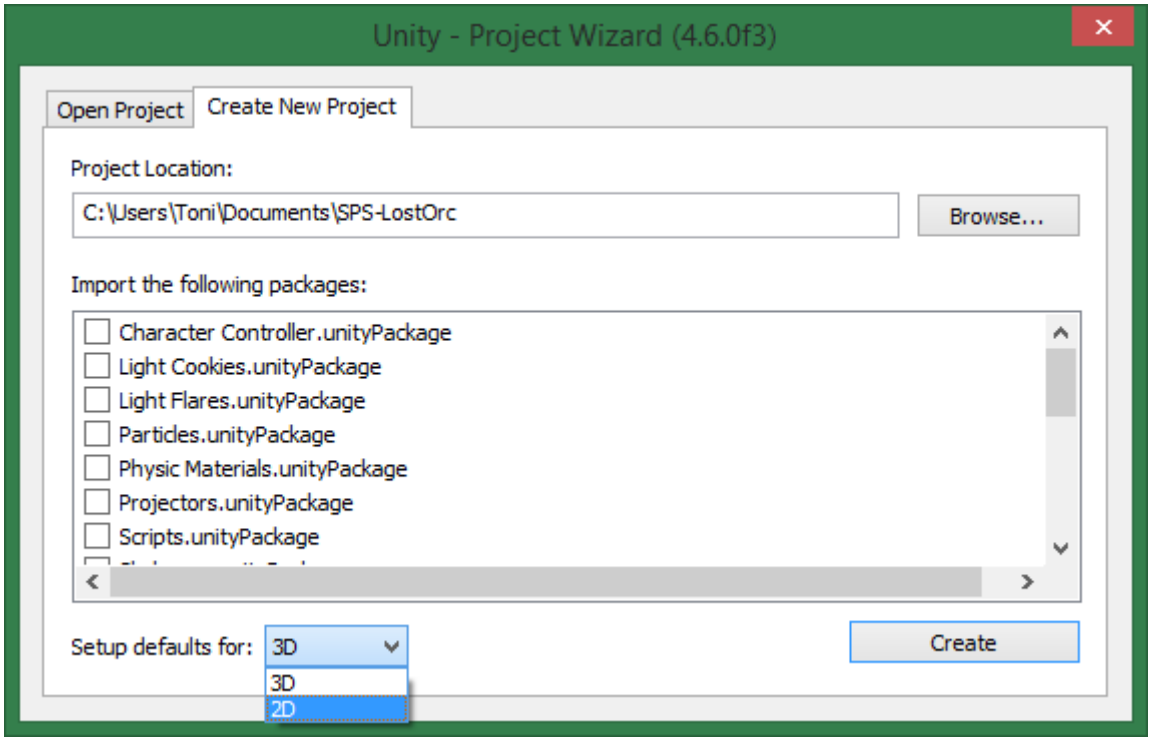
# Uvod u Unity – Vježba 1

Kako bi započeli s kreiranjem igre, pokrenite Unity i kreirajte novi projekt. Bitno je odabrati **Setup defaults 2D**, kako bi sam Unity editor tretirao naš projekt kao 2D projekt, postavio prikaz scene u 2D prikaz, prebacio kameru na ortografski pogled i svaku novu uvezenu sliku označio kao **sprite**[1] (prikazano na slici 1).



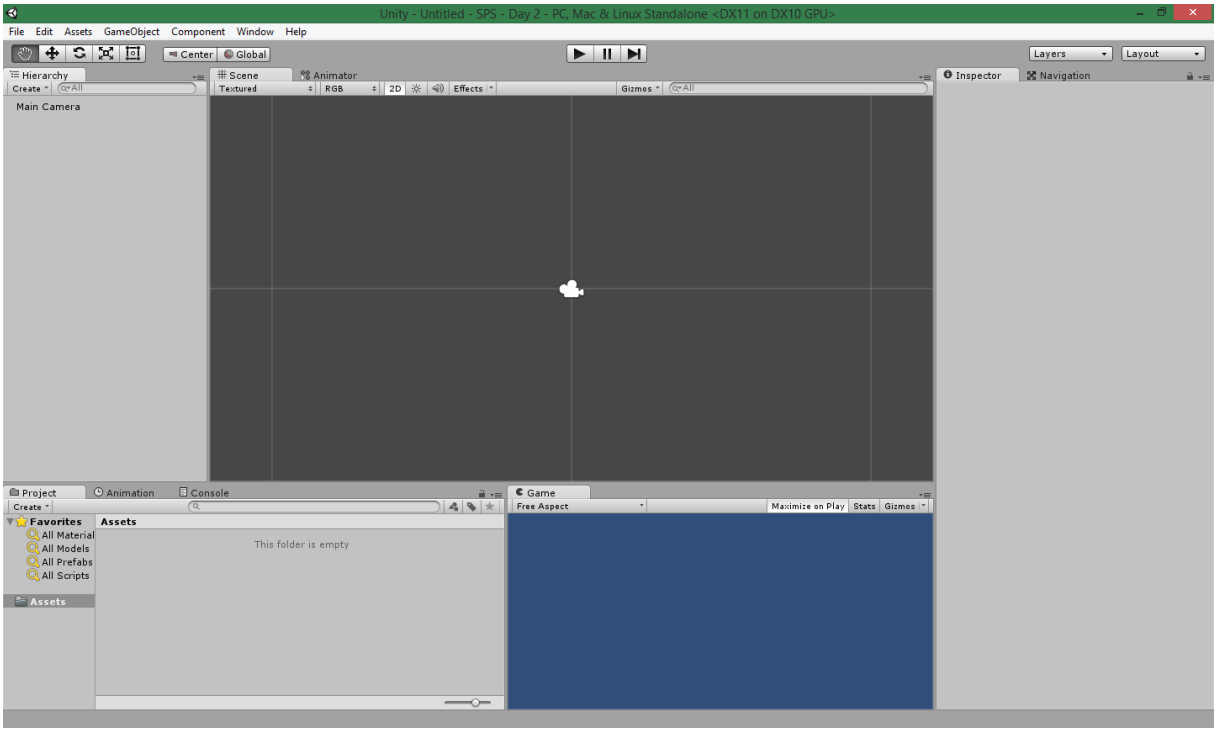

***Slika 1:*** *Kreiranje 2D projekta*

Nakon što ste kreirali projekt otvara se prazan Unity editor u kojem ćemo tijekom današnje vježbe kreirati našeg 2D lika. Za daljnji rad potrebno je preuzeti sve materijale koji se mogu naći na sljedećem linku:

https://github.com/steyskal/Student-poucava-studenta/tree/master/Day%202/Assets

Otvoren editor vidljiv je na slici 2.

[1] tip teksture koji se koristi u 2D razvojnom okviru

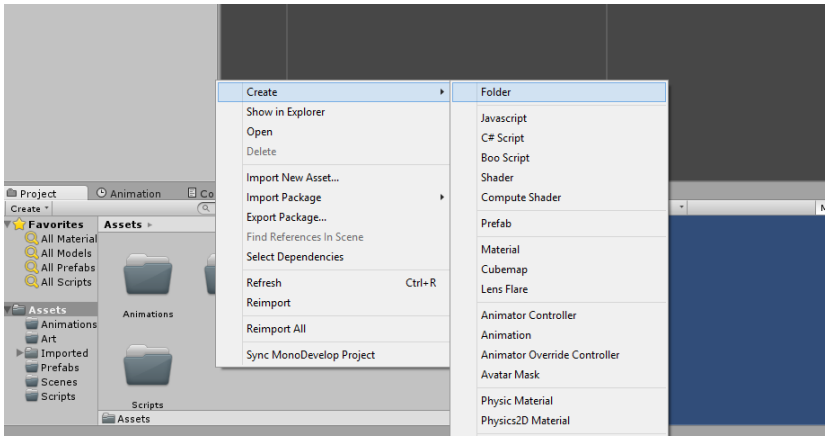



***Slika 2:*** *Unity editor*

Preimenujte Main Camera u **MainCamera** preko **Inspector**-a, jer će se kroz projekt koristiti **CamelCase**[2] notacija da postoji jedinstven način nazivanja objekata u hijerarhiji ukoliko ih je potrebno kasnije referencirat, a i to predstavlja stvar dobre prakse. ☺

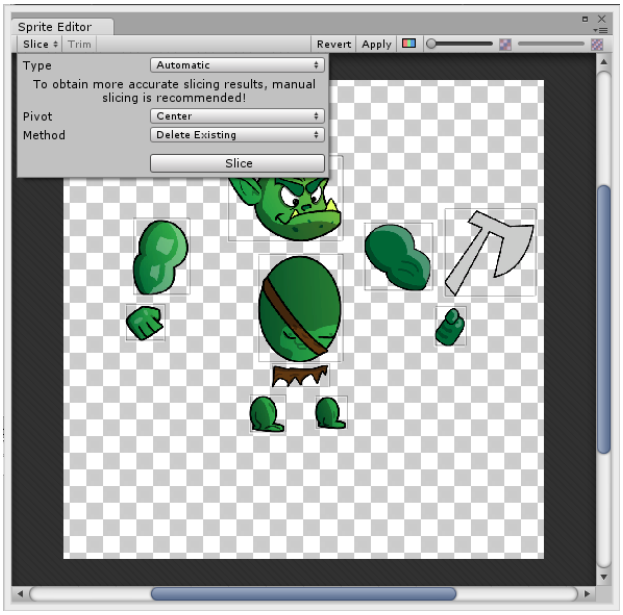
Zbog daljnje preglednosti kreirajte sljedeće datoteke: **Animations, Art, Prefabs, Scenes** i **Scripts**. Kao i što sami nazivi pojedinih datoteka govore, u pojedine će se spremati razne animacije, slike, scene (nivoi), skripte i **prefab**[3]-ovi koji će se koristiti tijekom razvoja igre. Kreiranje nove datoteke je vidljivo na slici 3.

[2] praksa pisanja složenih riječi ili fraza tako da svaka riječ ili kratica počinje velikim slovom
[3] predložak iz kojeg se može stvoriti novi objekt u sceni




***Slika 3:*** *Kreiranje nove datoteke*

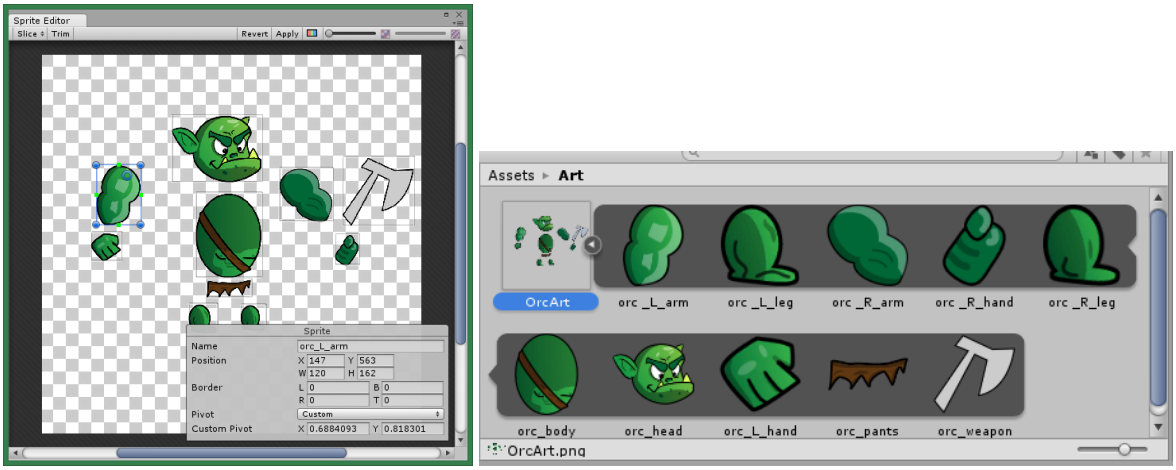
Iz preuzetih materijala kopirajte (ili drag & drop) **OrcArt.png** u kreiranu Art datoteku unutar Unity Editora. Označite uneseni sprite te promijenite njegove postavke **Sprite Mode -> Multiple** i odznačite **Generate Mip Maps**[4]. Kako se zapravo naša slika sastoji od više dijelova samog lika potrebno ju je urediti unutar **Sprite Editor**-a. Rascijepat ćemo ju na manje dijelove koje ćemo koristiti za kreiranje našeg lika, kao što je vidljivo na slici.




***Slika 4:*** *Rascijepanje sprite-a*

[4] Mip Maps predstavlja listu manjih verzija slike koje se koriste za optimiziranje performansi za objekte koji su udaljeni od kamere

Nakon pritiska na **Apply** sprite OrcArt dobit će zasebne dijelove odjeljenje u slici 4. Pojedine dijelove potrebo je preimenovati (unutar Sprite Editor-a) prema slici 5. I dodijeliti im pozicije pivota prema tablici 1.




***Slika 5:*** *Imenovanje dijelova*

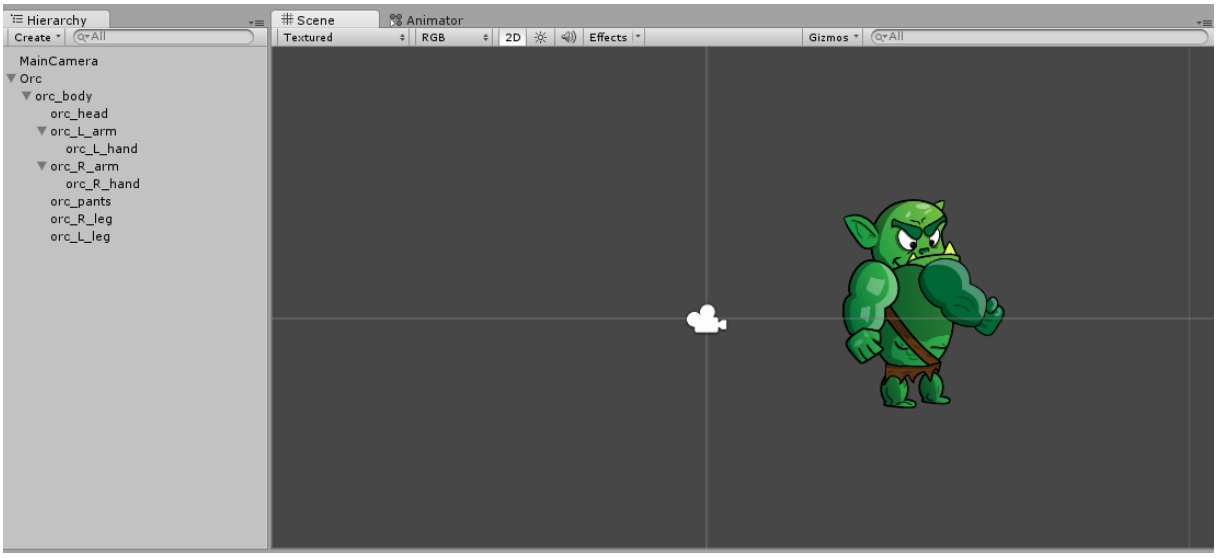
Izuzetno je bitno da se dijelovi imenuju kako su prikazani, jer se predefinirane animacije za taj preuzeti asset vežu na nazive i hijerarhiju objekta. Tako su nazivi redom: **orc_body, orc_head, orc _L_arm, orc_L_hand, orc _R_arm, orc _R_hand, orc_pants, orc _R_leg** i **orc _L_leg** (kako autor lika očito ne poznaje notacije imenovanja i konzistentnost, postoje razmaci i prije povlake _, koje je kasno u noć teško uočiti, zato pazite kako dajete nazive i uštedite korisnicima poput meni noć debuggiranja i traženja greške zbog par razmaka…).

| Orc | Pivot X | Pivot Y |
|---|---|---|
| **Head** | 0.486188 | 0.1940409 |
| **Body** | 0,5 | 0,5 |
| **LeftArm** | 0.6884093 | 0.818301 |
| **LeftHand** | 0.263753 | 0.7673514 |
| **RightArm** | 0.219031 | 0.8085149 |
| **RightHand** | 0,5 | 0,5 |
| **Pants** | 0,5 | 0,5 |
| **LeftLeg** | 0.3634006 | 0.9244167 |
| **RightLeg** | 0.3787468 | 0.921729 |

***Tablica 1.*** *Pivot pozicije*

Kako bi kreirali vlastitog lika prvo moramo stvoriti prazan objekt u sceni koji će sadržavati sve njegove dijelove. Desni klik unutar hijerarhije pa **CreateEmpty**, kreirani objekt preimenujte u **Orc**. Tom objektu dodajte pojedine dijelove (drag & drop) OrcArt-a kao što je prikazano na slici

6, izuzev sjekire, naš Orc nije ratoboran, samo mrzovoljan i izgubljen (ukoliko želite, kasnije možete dodati sjekiru i mogućnost napadanja, te tako proširiti igru).




***Slika 6:*** *Orc hijerarhija*

Naime dolazi do problema nepravilnog preklapanja dijelova koji čine našeg lika. Ova vrsta problema se može riješiti na 2 načina, preko određivanja Z osi zasebno za svaki dio ili slojeva sortiranja. U ovom primjeru će se koristi **Sorting Layer** pristup. Preko inspector-a našeg Orc objekta pod opcijom **Layer** odaberite **Add Layer**. Pod Sorting Layers dodajte Player pod **Layer 1** i ispod u Layer dodajte također Player pod **User Layer 8**. Zatim se vratite na sam objekt te odaberite **Player** kao njegov sloj i dozvolite Unity-u da učini isto njegovoj djeci.

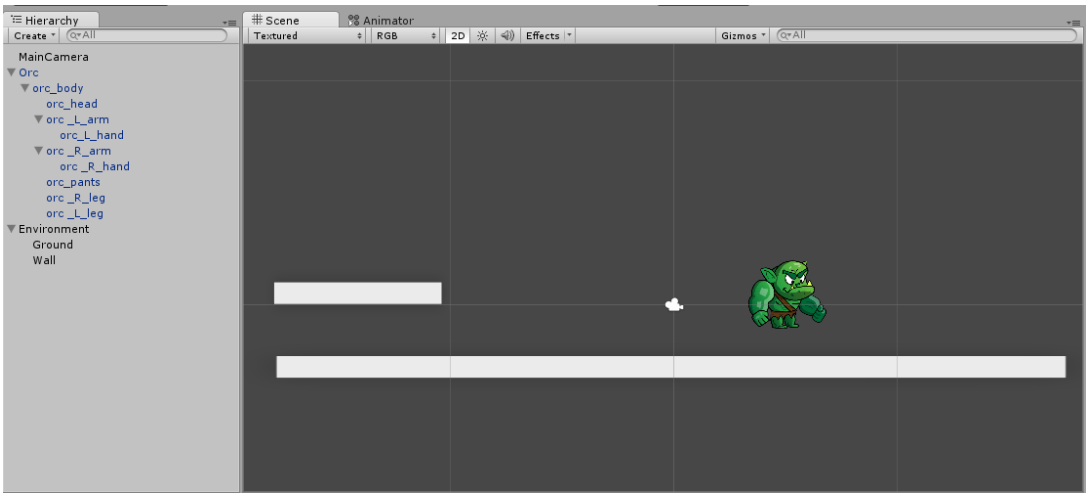
Sad kako smo Orc-a dodali na Player sloj potrebno je njegove dijelove sortirati i pozicionirati kako bi on sam dobio nekakav oblik. Učinite to prema sljedećoj tablici.

| Orc | Layer | Sorting Layer | Order | X position | Y position | Z position |
|---|---|---|---|---|---|---|
| **Head** | Player | Player | 1 | 0,1458159 | -0,1245941 | 0 |
| **Body** | Player | Player | 0 | 0 | 1,620405 | 0 |
| **LeftArm** | Player | Player | 4 | -0,7376138 | 0,183571 | 0 |
| **LeftHand** | Player | Player | 5 | -0,4679554 | -1,230647 | 0 |
| **RightArm** | Player | Player | -2 | 0,5498632 | 0,01226807 | 0 |
| **RightHand** | Player | Player | -1 | 1,059995 | -1,041762 | 0 |
| **Pants** | Player | Player | 3 | -0,01526686 | -1,107126 | 0 |
| **LeftLeg** | Player | Player | 2 | -0,2869641 | -0,9984478 | 0 |
| **RightLeg** | Player | Player | 1 | 0,2737674 | -1,022029 | 0 |

***Tablica 2.*** *Sortiranje i pozicija lika*

Sada kako smo konačno kreirali lika po našoj mjeri, povučemo gotov objekt iz hijerarhije u datoteku **Prefabs**, te tako stvaramo predložak glavnog lika koji kasnije možemo mijenjati ili koristiti i negdje drugdje.

Budući da sada imamo kreiranog Orc-a, potrebno je kreirati određene skripte kako bi mogli njime upravljati. Da bi mogli testirati dali sve što radimo funkcionira kako treba, kreirat ćemo privremenu testu okolinu pomoću **Wall.png** slike. Izgled prve scene može biti proizvoljan, bitno je da postoji dio na kojem naš lik može stajati i na njemu se kretati. Okvirni izgled scene vidljiv je na slici 7. Također promijenite postavu **Size** kamere na 15.

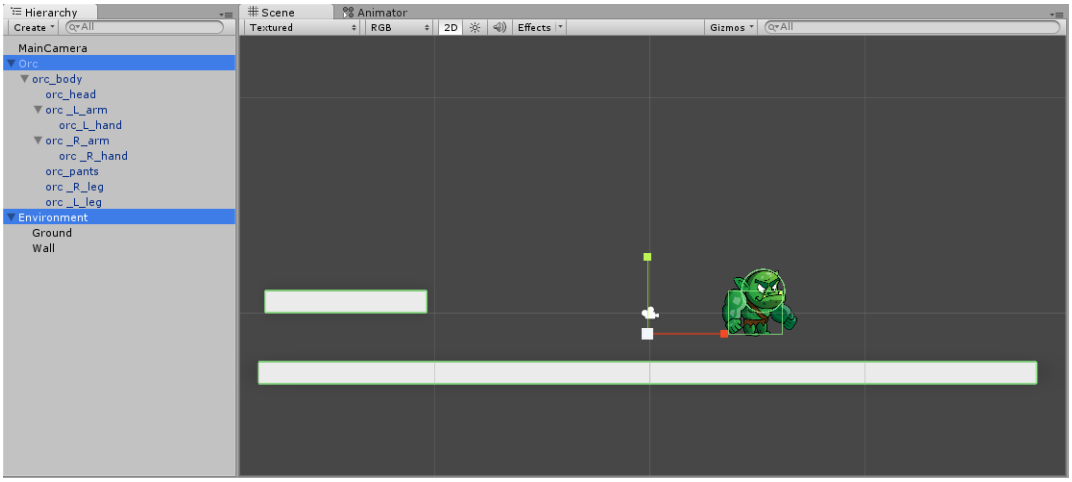



***Slika 7:*** *Dodavanje okoline*

Pokretanjem igre uočavamo da se zapravo ništa ne događa, već sve stoji kako je i predefinirano. Kako bi na našeg lika utjecala gravitacija i kako bi on mogao biti u interakciji s fizikom 2D svijeta, dodajemo mu komponentu **Physics 2D -> Rigidbody2D** [5] . Također označujemo postavku **Fixed Angle** jer u ovoj igri ne želimo da se lik rotira, a ona nam to omogućuje. No i dalje primjećujemo da nije sve kako treba biti, naime naš igrač prolazi kroz pod, da bi to izbjegli dodanim dijelovima okoline potrebno je dodati komponente **Physics 2D -> Box Collider 2D**. Bitno je također podesiti postavke dodanog collider-a da odgovara veličini sprite-a (Size: X = 0.25 i Y = 2.85). Istu stvar je potrebno učiti i kod igrača, kako bi Unity znao koliko gdje su rubne granice igrača, no njemu dodajemo jedan **Circle Collider 2D** (X = 0.35, Y =

[5] klasa koja objektu pruža funkcionalnosti fizike i stavlja objekt pod kontrolu sustava za fiziku

2, Radius = 1) i jedan **Box Collider 2D** (Center: X = 0, Y = 1, Size: X = 2.5, Y = 2). Konačan izgled prikazan je na sljedećoj slici.




***Slika 8:*** *Dodavanje collider-a*

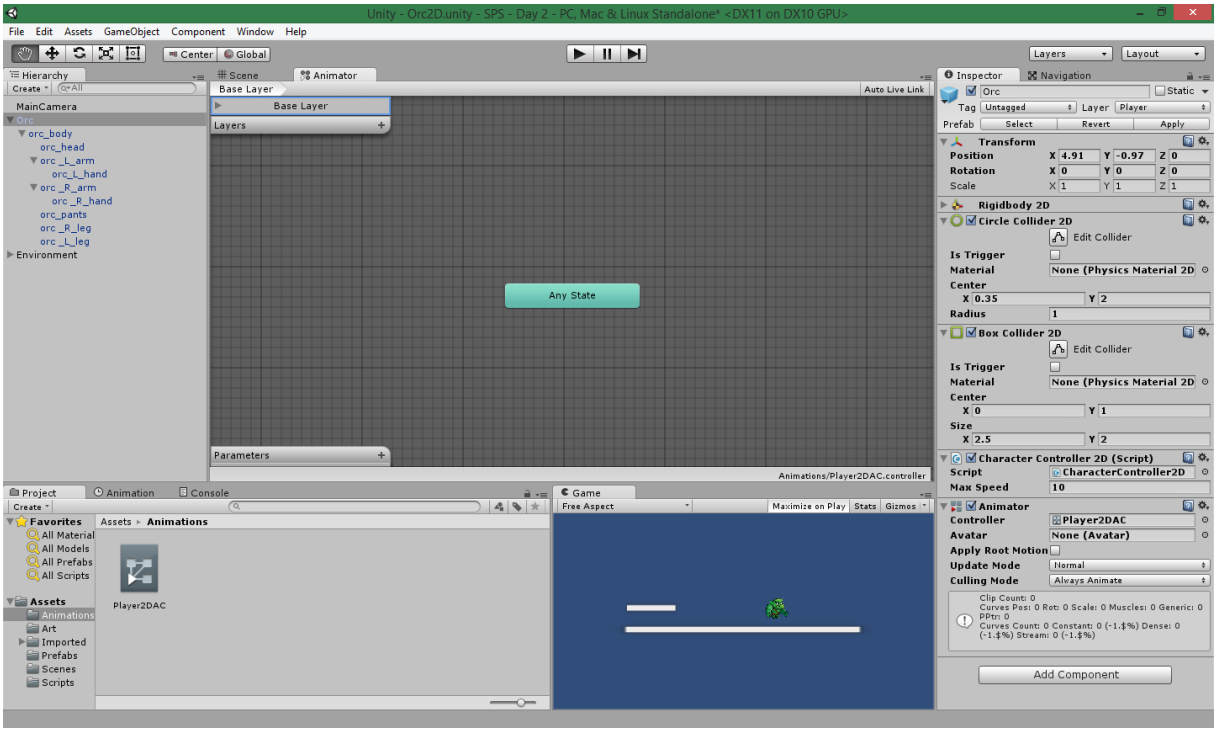
Sad smo definirali okolinu i način interakcije okoline s igračem. Samo kretanje se rješava preko kreiranja skripta (u ovom slučaju koristit ćemo C#) i pridruživanje istih objektu. Unity sadrži ugrađen IDE za kreiranje skripata nazvan **Monodevelop**, no skripte se mogu kreirati u bilo kojem razvojnom okruženju koje podržava jezik u kojem se one kreiraju. Najjednostavniji način je korištenje samog Monodevelop alata kako je on već integriran i prilagođen Unity-u. Kreirajte C# skriptu **CharacterController2D** unutar **Scripts** foldera, dodijelite ju na našeg igrača te ju otvorite. Skripta neće biti detaljno objašnjena već se koriste komentari unutar samog koda kako bi se shvatili pojedini dijelovi. Skripta je vidljiva na slici 9.

```
using UnityEngine;
using System.Collections;

public class CharacterController2D : MonoBehaviour {

    public float maxSpeed = 10.0f;
    bool facingRight = true;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

    // FixedUpdate is called every physics step
    void FixedUpdate (){
        
        // storing the keyboard input in the move variable (-1 or 1)
        float move = Input.GetAxis ("Horizontal");

        // adding movement (velocity) to our rigidbody (Orc) using a vector
        rigidbody2D.velocity = new Vector2 (move * maxSpeed, rigidbody2D.velocity.y);

        // checking in which direction is the rigidbody moving and if it needs to flip
        if (move > 0 && !facingRight)
            Flip ();
        else if (move < 0 && facingRight)
            Flip ();
    }

    // Flip the character
    void Flip(){
        // change our facing variable
        facingRight = !facingRight;
        // store the scale of our rigidbody to theScale variable
        Vector3 theScale = transform.localScale;
        // flip the stored scale
        theScale.x *= -1;
        // flip the rigidbody
        transform.localScale = theScale;
    }
}
```

***Slika 9:*** *CharacterController2D*

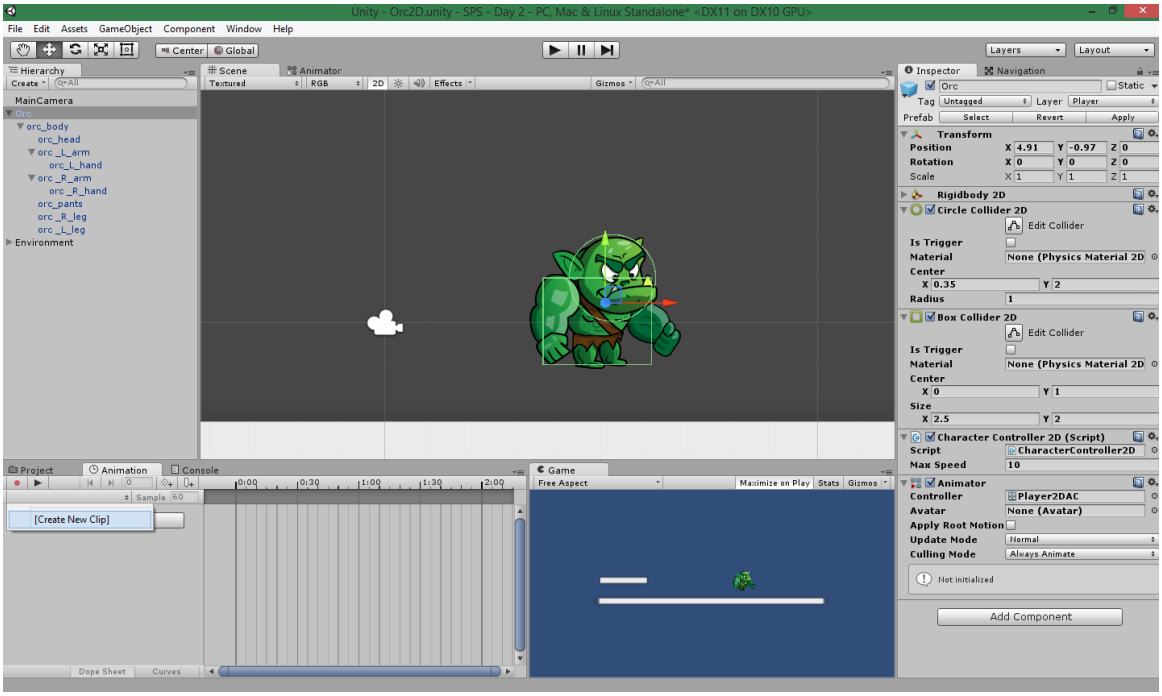
Spremite skriptu te pokrenite igru, sada se Orc može kretati te se okreće prema strani u koju se kreće. No vidimo određenu manu, naime kretanje ne izgleda prirodno, odnosno ne postoje animacije. Animacije ćemo dodati u nastavku. Kao primjer će se kreirati jedna animacija **Idle** (animacija koja će se prikazivat kada lik ne radi ništa drugo), a druge preuzeti gotove iz danih materijala. Unity za rukovanje i kreiranje animacija koristi sustav **Mecanim**, te kako bi započeli kreiranje animacije potrebno je objektu Orc dodati komponentu **Miscellaneous -> Animator**. Potrebno je odznačiti opciju **Apply Root Motion** jer ne želimo da kretnje animacije definiraju

kretnju objekta lika. Kako bi kontrolirali prikazivanje i tijek animacija potrebno je kreirati **Animation Controller** unutar datoteke Animations, nazovite ga **Player2DAC**. Te povucite taj kontroler na Orc objekt pod komponentu Animator i njegovu postavku **Controller**. Otvorite kreirani animator kontroler dvoklikom ili odabirom Animator tab-a kada je selektiran Orc objekt. Početni izgled kreiranog kontrolera vidljiv je na slici 10.



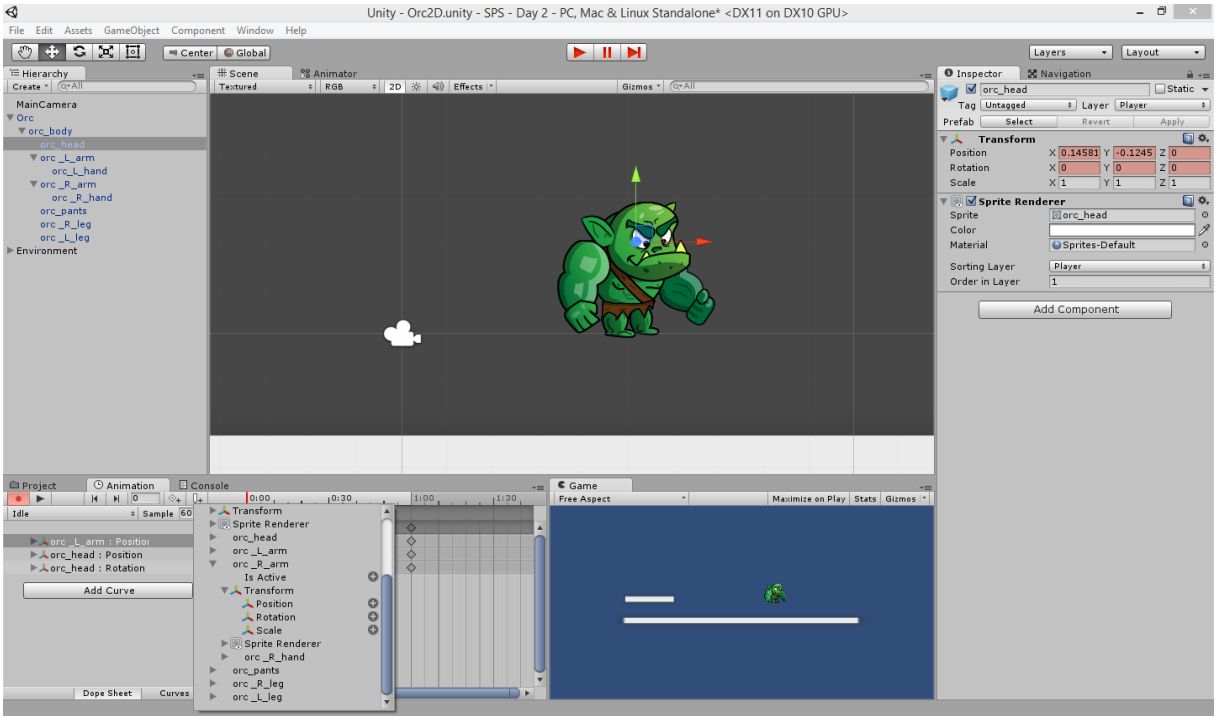

***Slika 10:*** *Animator Controller*

Sada je potrebno kreirati animaciju koju će navedeni kontroler koristiti, otvorite prozor **Window -> Animation**. Zatim označite Orc game objekt kako bi nad njim radili animacije (ukoliko nije selektiran, animacije neće raditi kako bi trebale). Kreirajte novu animaciju **Idle**, te ju spremite u Animations datoteku, prema sljedećoj slici.




***Slika 11:*** *Kreiranje animacije*

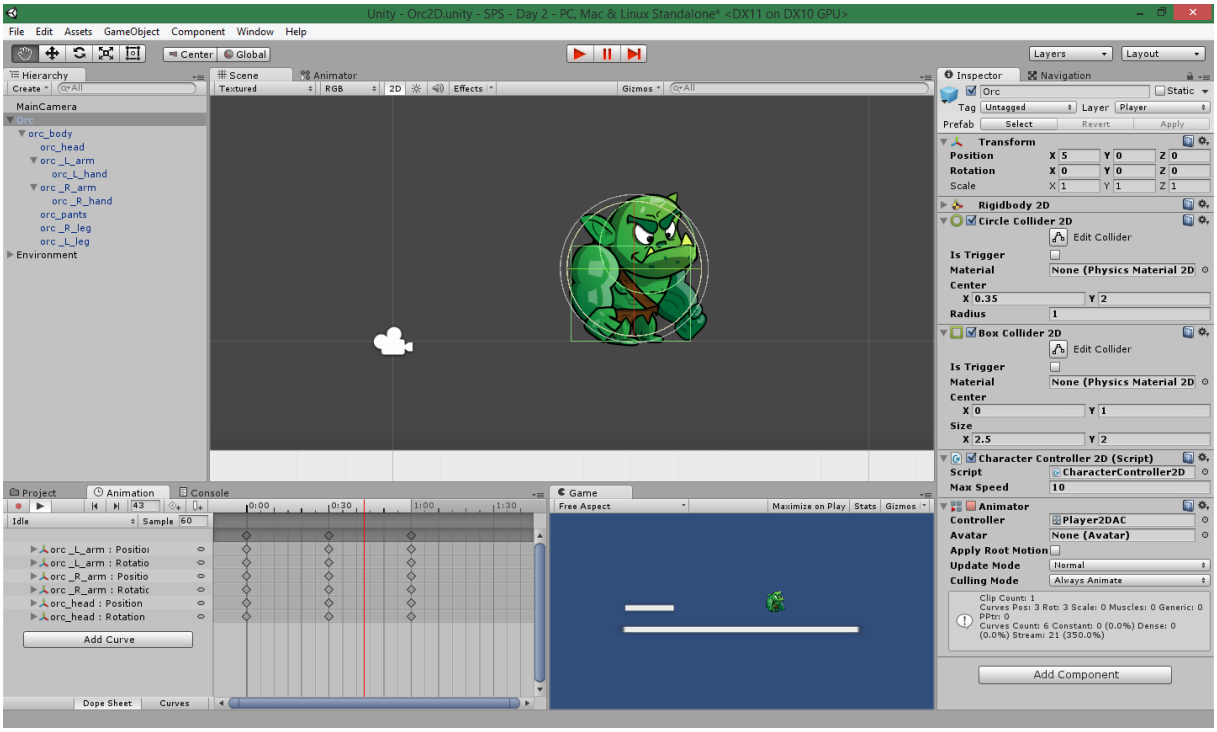
Nakon što se kreirala animacija Idle, u Animator tabu možemo vidjeti novo stanje pod nazivom Idle koje je vezano uz kreiranu animaciju. Unity također automatski postavlja to stanje kao default stanje (označeno narančastom bojom). Da bi kreirali animaciju moramo dodati određene dijelove koji će se mijenjati (opcija **Add Curves**). Kako bi to trebalo izgledati vidljivo je na slici 12.




***Slika 12:*** *Idle animacija u izradi*

Na vremenskom intervalu 30 sekundi dodajte novi ključ (desni klik -> **Add key**) te tu promijenite pozicije i rotacije crveno označenih postavka samog game objekta (preko inspector-a ili pogleda na scenu). Moguće je i mijenjati druge dijelove preko inspector-a, te ih Unity tada sam dodjeljuje kao ključ u animaciji. Kako bi osigurali da nam animacija ima neprimjetan prijelaz prilikom njezinog ponavljanja (kako se radi o animaciji koja se može prikazivati duže vrijeme) obrišemo ključeve na 1:00, odnosno kraju animacije, te kopiramo početne na njihovo mjesto. Pritiskom na Play gumb u bilo kojem trenutku možete vidjeti tijek vaše animacije u izradi. Animacija u tijeku je vidljiva na slici 13. Pokrenite igru i prođite kroz tabove **Scene** i **Animator** da vidite kako to izgleda tijekom pokretanja igre (imajte označeno Orc game objekt).

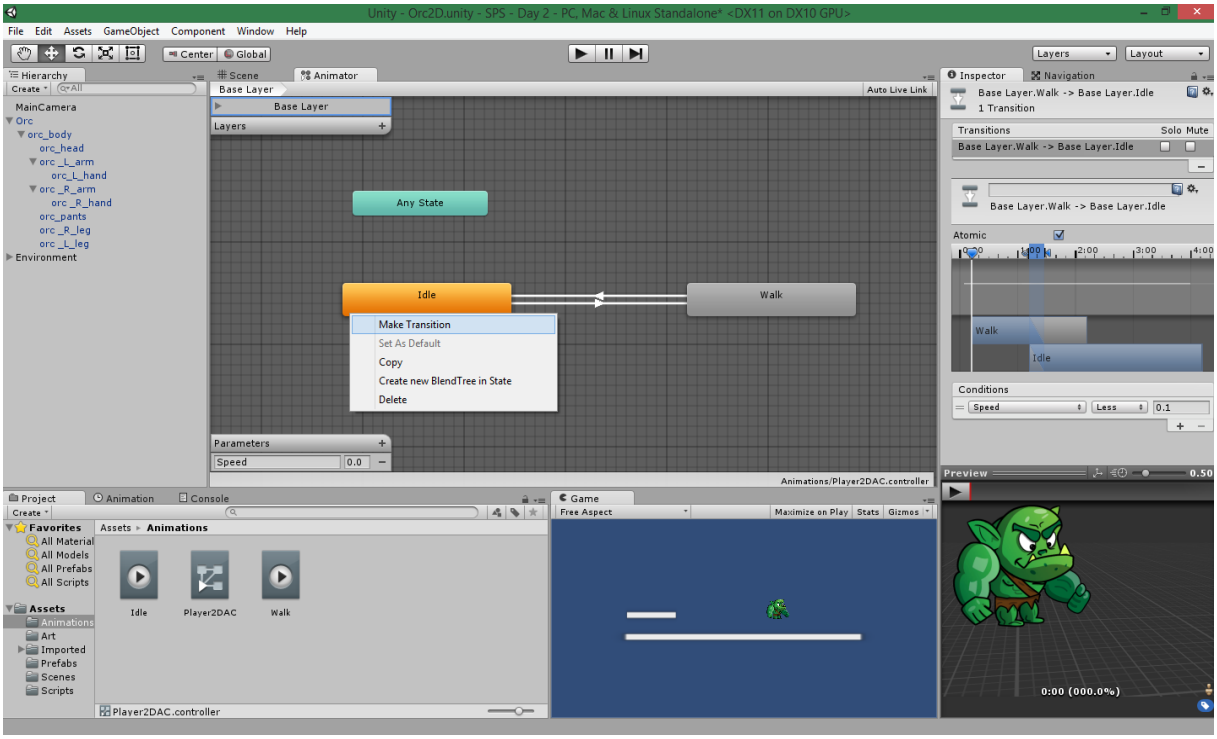
Što se tiče same animacije, ona može biti bilo kakva, te možete nadodati još dijelova kojima ćete upravljati tijekom animacije. Iskoristite ovu izradu animacije kako bi pohvatali osnove i kreirajte nešto po želji. Gotove animacije ćemo kasnije preuzeti iz skinutih materijala.



***Slika 13:*** *Idle animacija*

Kako većina nas nismo rođeni umjetnici s talentom izrade animacija, obrisat ćemo naš sav dosadašnji trud kreirane animacije te kopirat gotove **Idle** i **Walk** animacije u **Animations** datoteku. Kako je naziv jednak, nije potrebno ništa mijenjat da bi vidjeli definiranu animaciju, no ukoliko bi to bilo potrebno promjena se radi unutar Animatora, Idle stanja, pod opcijom **Motion** gdje odabirete željenu animaciju. Povucite novu animaciju **Walk** u Animator tab, te uočite kako se stvorilo novo stanje Walk (sive boje). Sada moramo definirati Animatoru kada je potrebno prijeći iz jedne animacije u drugu, te se vratiti u početnu. Za to su nam potrebni određeni parametri i tranzicije među animacijama.

Kreirajte **Float** parametar te ga nazovite **Speed**, te kreirajte tranziciju između Idle i Walk (desni klik na Idle i **Make transition**). Na toj tranziciji definirajte uvjet da je **Speed Greater** than **0.1**. Te učinite isto za povratnu tranziciju s Walk na Idle, samo što ovdje stavite uvjet **Speed Less** than **0.1**, kao što je prikazano na slici 14.




***Slika 14:*** *Animator – tranzicije i parametri*

Ukoliko ste već pokrenuli igru i ponadali se da je to to, uočili ste da se prikazuje samo Idle animacija nevezano uz našu kretnju. Razlog tome je što nigdje nismo definirali promjenu parametra brzine, to moramo učiniti u skripti **PlayerController2D**. Dodajte nove linije koda prikazane sljedećim slikama.

```
6   public float maxSpeed = 10.0f;
7   bool facingRight = true;
8   Animator animator;
9
10  // Use this for initialization
11  void Start () {
12      animator = GetComponent<Animator> ();
13  }
```

```
22
23      // storing the keyboard input in the move vari
24      float move = Input.GetAxis ("Horizontal");
25
26      animator.SetFloat ("Speed", Mathf.Abs(move));
27
28      // adding movement (velocity) to our rigidbody
29      rigidbody2D.velocity = new Vector2 (move * max
```

***Slika 15:*** *Promjena parametra Speed*

Sada se igrač kreće i animira kako bi trebao. Kako bi spremili sve promijene na prefab u inspektoru Orc game objekta pritisnemo **Apply**. Ukoliko provjerite opcije prefab-a, one su izjednačene s našim objektom unutar igre.

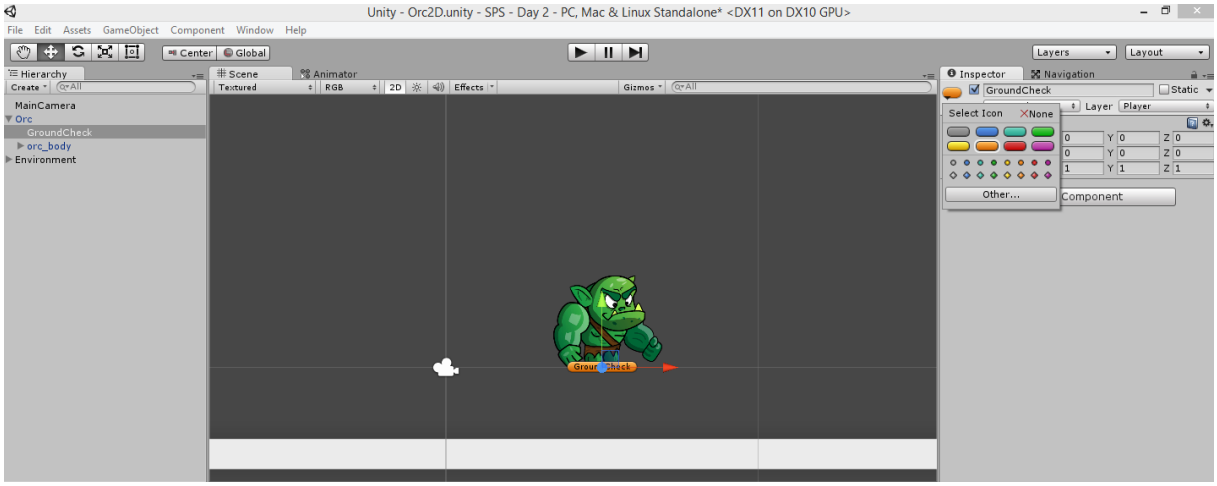
Jedna od bitnih mogućnosti kontrole igrača u 2D platformer igrama je skok, kako bi omogućili našem liku da skače, potrebno je koristiti nove animacije i dodati nekoliko linija koda u našu skriptu. Bitno je napomenuti da se realizacija skoka može izvršiti na mnogo različitih načina, te je ovo samo jedan od njih.

Prvi dio u realizaciji skoka jest provjera da li se naš igrač nalazi na nekom drugom objektu koji predstavlja pod. Dodat ćemo sljedeće linije koda prikazane na slici 16.

```
    Animator animator;

    bool grounded = false;
    // reference to another object which indicates where ground should be
    public Transform groundCheck;
    // size of the sphere in which we check for ground
    float groundRadius = 0.2f;
    // layer that represents ground
    public LayerMask whatIsGround;

    // Use this for initialization
    void Start () {
        animator = GetComponent<Animator> ();
    }

    // Update is called once per frame
    void Update () {

    }

    // FixedUpdate is called every physics step
    void FixedUpdate (){

        // constantly check if player is on ground
        grounded = Physics2D.OverlapCircle (groundCheck.position, groundRadius, whatIsGround);

        animator.SetBool ("Ground", grounded);

        // storing the keyboard input in the move variable (-1 or 1)
        float move = Input.GetAxis ("Horizontal");
```

***Slika 16:*** *Ground check*

Sada kreirajte novi prazni game objekt, preimenujte ga u **GroundCheck**, te ga postavite kao dijete Orc objekta. Resetirajte poziciju tog objekta, te ukoliko je potrebno premjestite ga na samo dno našeg igrača i stavite ga pod layer **Player**. Također, unutar inspektora, postavite objektu ikonu da bude bolje vidljiv unutar pogleda scene. Rezultat dodavanja GroundCheck objekta vidljiv je na slici 17.



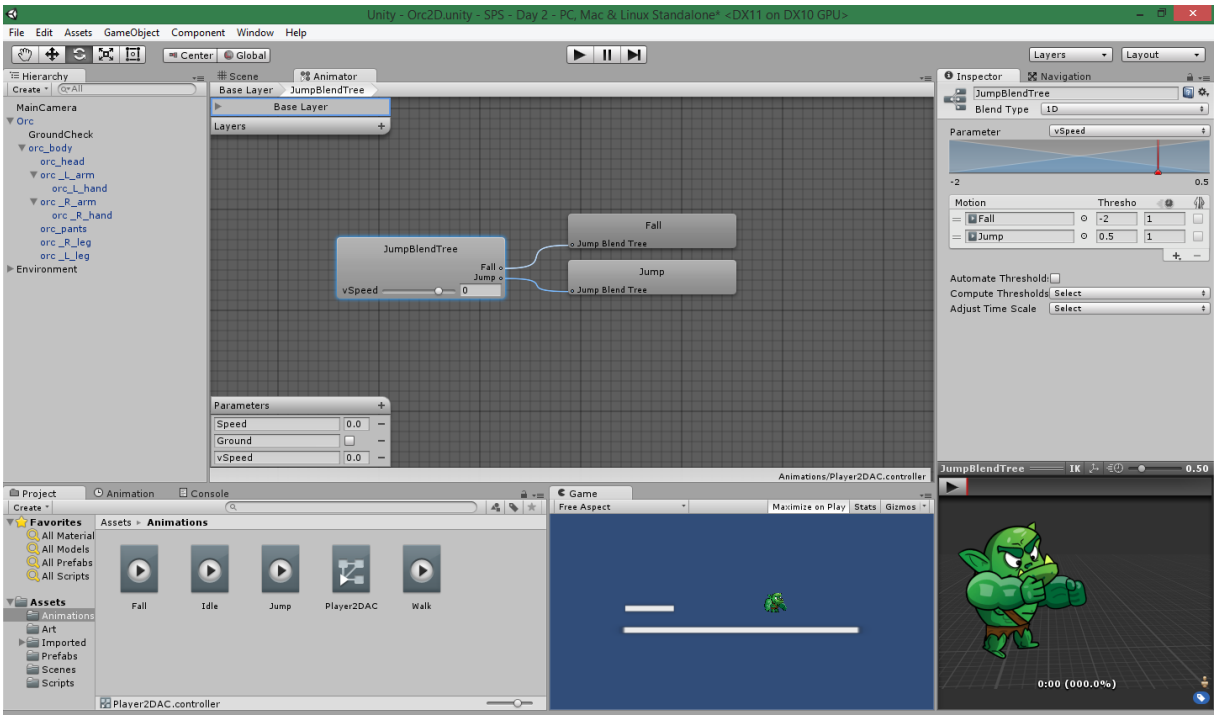

***Slika 17:*** *GroundCheck objekt*

Taj novo kreirani objekt drag & dropp-amo pod Orc komponentu **PlayerController2D** unutar opcije **Ground Check**. Zatim pod opcijom **What Is Ground** odaberemo **Everything** i odznačimo **Player** layer (ne želimo da igrač stvara koliziju sa samim sobom). Unutar animatora dodajte nove parametre **Ground** (bool) i **vSpeed** (float). Ostavite animator tab otvoren, te isprobajte igru (bacite se niz liticu!), da vidite kako se ponaša varijabla Ground. Sada kada znamo u kojem trenutku možemo skočiti potrebno je realizirati skok unutar skripte. Prvo dodajte novu varijablu **public float** *jumpForce = 100.0f* (ispod ***Animator*** *animator*) i unutar skripte (**Update** i **FixedUpdate** funkcije) učinite sljedeće:

```
24      // Update is called once per frame
25      void Update () {
26
27          // check if the player is grounded and jump has been pressed
28          if (grounded && Input.GetAxisRaw ("Jump") == 1) {
29              // player is no longer on the ground
30              animator.SetBool ("Ground", false);
31
32              // make the rigidbody jump
33              rigidbody2D.AddForce (new Vector2 (rigidbody2D.position.x, jumpForce));
34          }
35      }
36
37      // FixedUpdate is called every physics step
38      void FixedUpdate (){
39
40          // constantly check if player is on ground
41          grounded = Physics2D.OverlapCircle (groundCheck.position, groundRadius, whatIsGround);
42
43          animator.SetBool ("Ground", grounded);
44
45          animator.SetFloat ("vSpeed", rigidbody2D.velocity.y);
46
47          // storing the keyboard input in the move variable (-1 or 1)
48          float move = Input.GetAxis ("Horizontal");
```

***Slika 18:*** *Skriptiranje skoka*

I upravo smo omogućili našem igraču da skače, no javlja se isti problem kao i prvi puta kod kretanja, potrebno je dodati animacije za skok i padanje. Dodajte animacije **Fall** i **Jump** u projekt unutar foldera **Animations**, te ih obrišite u **Animator**-u (one se automatski dodaju, no mi to ne želimo). Ove animacije nećemo koristiti zasebno kao Idle i Walk, već ćemo kreirati jedno stanje u animatoru **Create State -> Form New Blend Tree** pomoću kojeg ćemo određivati da li igrač pada ili skače, te koju animaciju je u određenom trenutku potrebno prikazati. Dodajte tranziciju s **Any State** u **JumpBlendTree** gdje uvjet da **Ground** equals **false** i tranziciju s **JumpBlendTree** u **Idle** sa uvjetom **Ground** equals **true**. Dvoklikom otvorite kreirani blend tree. Konfigurirajte blend tree kako je prikazano na slici 19.




***Slika 19:*** *Animiranje skoka*

Sada kako naš igrač sadrži one najosnovnije mogućnosti kretnje, završili smo s kreiranjem igrača. Ukoliko želite poigrajte se još s animacijama, dodajte nove i bolje animacije za skok (jer sam ja radio Jump i Fall ☺ ), ili ubacite nešto svoje u skriptu (mogućnost teleportacije, dvostruki skok, čučanje, napad…). Svakako nemojte zaboravit snimit sve postavke na prefab (preko **Apply**) i snimit scenu.

Kako bi mogli u potpunosti proći nivo koji će biti kreiran u sljedećoj vježbi bit će vam potreban dvostruki skok. Svakako pokušajte to učiniti sami, no ne brinite ukoliko ne uspijete jer ćemo to proći zajedno na sljedećoj vježbi.