Izrada 2D platformske igre korištenjem Unity Enginea

Drulak, Ivan

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni preddiplomski studij računarstva

IZRADA 2D PLATFORMSKE IGRE KORIŠTENJEM UNITY ENGINE-A

Završni rad

Ivan Drulak

Osijek, 2017.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 12.07.2017.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada

Ime i prezime studenta:	Ivan Drulak
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3632, 23.07.2014.
OIB studenta:	24631697502
Mentor:	Doc.dr.sc. Časlav Livada
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Izrada 2D platformske igre korištenjem Unity Enginea
Znanstvena grana rada:	**Informacijski sustavi (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 2 razina
Datum prijedloga ocjene mentora:	12.07.2017.
Datum potvrde ocjene Odbora:	17.07.2017.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 17.07.2017.

Ime i prezime studenta:	Ivan Drulak
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3632, 23.07.2014.
Ephorus podudaranje [%]:	1

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada 2D platformske igre korištenjem Unity Enginea**

izrađen pod vodstvom mentora Doc.dr.sc. Časlav Livada

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Tema ovog završnoga rada je računalna igra koja pripada tipu 2D plaformer. Takav tip računalnih igara je nastao osamdesetih godina dvadesetoga stoljeća među kojima su najpoznatije igre poput Super Maria i Donkey Konga predvođeni Nitendom. Kako bi lakše razumjeli što znači tip računalne igre platformer u nastavku slijedi kratko objašnjenje. Platformer je tip računalne igre u kojoj igrač kontrolira svoga lika koji može trčati, skakati, penjati se itd. po platformama koje predstavljaju podlogu po kojoj se lik može kretati pri čemu cijela okolina može biti strogo definirana (nepomična) ili se može mijenjati ovisno o položaju lika u igri (pomična). Naziv igre čija će izrada i ideja biti detaljnije objašnjena u ovom završnom radu je Return of Darkness koja pripada žanru *medival – fantasy* u kojoj se bore ljudi i bića poput goblina. Cijela igra napravljena u programu Unity koji predstavlja *game engine* koji omogućuje stvaranje računalne igre. Cijeli kod koji je bio potreban kako bi igra radila je pisan u C# programskom jeziku koji je objektno orijentiran. Sve skripte koje sadrže kodove i koje su vezane na objekte u igri omogućavaju upravljanje mehanikom same igre. Sama grafička (vizualna) okolina igre je izrađena u web aplikaciji Piskel, osim pozadine [1] koja je preuzeta. Što se tiče zvukova [2] korišten je program Bfxr te melodija koja je preuzeta sa stranica [3] slobodnog sadržaja.

1.1. Zadatak završnoga rada

Kao zadatak završnoga rada je zadano da se izradi 2D platfromer igra koja ima sve potrebne funkcije i mogućnosti mehanike koje su tipične za takvu vrstu računalne igre. Za uspješno izvršenje ovoga rada potrebno je znati služiti se Unity Engine-om, C# programskim jezikom i Piskel-om.

2. UNITY

Unity predstavlja višeplatformski *game engine* koji je razvijen s primarnim ciljem razvijanja i stvaranja računalnih igara i simulacija za računala, konzole, mobilne uređaje i internetske stranice. Stvorila ga je tvrtka Unity Technologies[4] 2005. godine te su kao prvu platformu odabrali OS X (*Apple Inc*). Ubrzo je omogućeno korištenje na drugim operacijskim sustavima poput Windows-a i Linux-a, a danas je podržano 27 različitih platformi[6]. Unity je pisan u C, C++, C#, UnityScript i Boo programskom jeziku. Svi navedeni programski jezici su objektno orijentirani osim C koji je proceduralni. Unity pruža mogućnost odabira za razvijanje 2D ili 3D računalne igre. Ne sastoji se samo od *engine-a* nego sadrži dodatne alate i sadržaje koji pomažu u ostvarivanju željenoga cilja, a to uključuje: Unity Asset Store, Cloud Build, Analytics, Ads, Everyplay i Certification. Prilikom pokretanja Unity-a može se odabrati između kreiranja novog projekta ili otvaranje postojećeg te mogućnost biranja registracije ili rada bez korisničkoga računa (*offline*). Nakon toga slijedi korak u kojem se otvara sučelje koje se sastoji od scene i raznih podajućih izbornika koji omogućuju sve što je potrebno za razvoj jedne igre. Tu se nalaze razne postavke od mjesta gdje želimo spremiti naš projekt i scenu, komponente i objekte koje želimo koristiti u sceni sve do padajućeg izbornika koji nam nudi pomoć i upute oko korištenja. Mjesto gdje se postavlja i stvara okruženje igre je scena. Moguće je imati jednu scenu u cijeloj igri, ali u praksi se koristi više scena pri čemu jedna scena najčešće predstavlja jedan nivo igre. Sve što se stavlja u scenu može biti dio samoga Unity-a ili ukoliko nešto stavljamo što nije u sklopu Unity-a sprema se u mapu *Assets* koja predstavlja glavnu pohranu svih sadržaja koje koristimo. Unutar same mape *Assets* mogu se kreirati vlastite mape koje možemo imenovati i u koje možemo ubacivati željeni sadržaj. Osnovne cjeline od kojih se gradi scena te izgrađuje igra su objekti. Objekti se mogu pomicati bilo gdje u sceni, mogu se rotirati u svim smjerovima, može se promijeniti njihova veličina itd. Kako bi ti objekti izgledali i radili što smo zamislili potrebno je dodati na njih *sprite-ove*[1], skripte, zvukove, animatore te razne druge komponente. Prilikom rada s bilo kojim objektom s desne strane postoji prozor pod nazivom Inspektor koji sadrži dodatne informacije o objektu i pomoću njega je moguće dodavati razne komponente na objekt. Omogućuje i imenovanje odabranoga objekta, postavljanja i kreiranja nadimaka (*Tag*) te mnoge druge pojedinosti. Osim kartice scene mogu se koristi i dodavati neke druge kartice, a neke od njih su: *Animation*, *Animator*, *Game*, *Profile*. Kartica animacije omogućuje kreiranje animacije vezane za neki objekt,. Kako bi uopće mogli znati za koji objekt radimo animaciju prvo moramo pokazivačem miša kliknuti na objekt i zatim stisnuti na karticu

[1] *Sprite – predstavlja bitmapu koja može sadržavati jednu ili više slika*

animacije. Tek tada možemo kreirati novu animaciju koju spremamo u projekt. Sama animacija se sastoji od dodavanja više slika (*Sprite-ova*) u slijedu pri čemu možemo odabirati brzinu mijenjanja slika i time dobivamo slike u pokretu odnosno animaciju. S druge strane animator omogućuje kontrolu animacija i određuje ovisno o parametrima kada će se koja animacija izvršiti, nešto detaljnije o animatoru biti će kasnije objašnjeno u potpoglavlju Animator. Game kartica zapravo je izgled scene kakvu bi igrač vidio prilikom pokretanja igre i sastoji se od dodatnih izbornika, a kako bi imala prikaz bitno je da u sceni postoji kamera. Prvi je prikaz (*Display*) pomoću kojega možemo odabrati koji prikaz želimo vidjeti ovisno o kameri u sceni, a zatim slijedi slobodni aspekt (*Free Aspect*) koji se odnosi na pojedine piksele i njihov omjer visine i širine (npr. slobodni aspekt 4:3 znači da je visina piksela 3 jedinice dužine, a širina piksela 4 jedinice dužine). Poslije slobodnog aspekta dolazi skala (*Scale*) koji omogućuje povećanje prikaza do pet puta, do skale slijedi kartica povećanje pri pokretanju (*Maximize on Play*) koju je moguće aktivirati te prilikom pokretanja scene Game kartica proširuje se preko cijelog prozora Unity-a. U nastavku se nalazi još kartica zanemari zvuk (*Mute audio*), statistika (*Stats*) koja prikazuje neka zvukovna i grafička svojstva i na kraju *Gizmos* koji je vezan za urednik (*editor*) scene. Može biti koristan kod označavanja nekih objekata koji se ne mogu vidjeti u uredniku ili da se neki objekti vide u *game* kartici kao i u uredniku. Kako bi se svi ili samo neki objekti u sceni mogli kontrolirati i pomicati, osobito glavni lik u igri koji je također objekt mora postojati neka poveznica, a ta poveznica je skripta koja se dodaje na objekt u Inspektoru. Sama skripta dodana na objekt neće imati nikakav utjecaj ako nije ništa u nju napisano u jednom od programskih jezika koji su podržani, a to su: C#, Javascript i Boo koji od verzije 5.0 pa nadalje nije ponuđen (neovisno o tome, ukoliko se piše tim jezikom Unity prevoditelj [*compiler*] će i dalje prevoditi kod pisan tim programskim jezikom). Svaka novostvorena skripta se otvara u uredniku za skripte (*Script Editor*) pri čemu se može koristiti ugrađeni MonoDevelop ili Visual Studio ili bilo koji drugi urednik. Svaka skripta sadrži javnu klasu s nazivom skripte i uključenu osnovnu klasu MonoBehavior koja sadrži brojne funkcije, operatore i ostalo definirano u samome Unity-u te se može pozivati i koristiti i to činilo tijelo skripte. Unutar same skripte se nalaze dvije funkcije: *Start()* i *Update()*. Prva funkcija se poziva samo jednom prilikom pokretanja scene dok druga funkcija se poziva jednom po slici (*frame*). Ponekad funkcija *Update()* se ne izvršava dovoljno često za pravilnu interakciju kontroli lika zbog kašnjenja jedne slike u odnosnu na drugu te se zbog toga koristi funkcija *FixedUpdate()*. U ovom radu je korišten isključivo C# programski jezik, a kao urednik skripti Visual Studio 2013.

3. RAZVOJ IGRE

3.1. Ideja

Zamisao rada je bila izraditi 2D računalnu igru koja će imati više nivoa i koja će u konačnici pratiti priču navedenu u uvodu. Kao glavni izvor ideja samih nivoa i princip rada igre uzeto je iz poznate 2D platformer igre Super Mario te je veliki utjecaj imala i vlastita mašta. Glavni lik kojim igrač upravlja je Maximus koji treba poraziti neprijatelje kroz tri nivoa i time osloboditi planet. Igračevim upravljanjem Maximus se može kretati lijevo-desno po platformama, skakati, bacati metalne zvijezde (*Shuriken)* i napadati mačem. Sve instrukcije koje služe za objašnjenje tipki koje se koriste za lika nalaze se u glavnom izborniku (*Main menu*). Igrač osim što se bori protiv neprijatelja može skupljati zlatnike koji mu služe da u konačnici ima bolji rezultat na kraju igre. U igri postoje četiri vrste neprijatelja, a to su: goblini rase Diranos, ljudi rase Rupex i njihov vođa Meranox te šišmiši. Goblini su bića koja nalikuju tijelom ljudima, a kao oružje koriste sjekire koje mogu bacati i nemaju oklop. U početku se kreću lijevo-desno, točnije patroliraju platformama na kojima se nalaze, a kada im se igrač približi na određenu udaljenost zastanu i počinju bacati sjekire. Svaka sjekira koja pogodi igrača smanjuje mu broj života za jedan dok svaka metalna zvijezda koja pogodi goblina skida jedan život goblinu, a ukoliko igrač napada mačem dovoljan je jedan udarac da ubije goblina. Goblin ima tri života dok igrač može imati od jednoga do osam života ovisno koju težinu igranja igrač izabere (ukoliko igrač izgubi sve živote kraj je igre i može pokušati ponovo). Drugu vrstu neprijatelja predstavljaju ljudi crnih oklopa koji koriste mač kao oružje i oni također patroliraju područjem sve dok im se igrač ne približi dovoljno blizu pri čemu onda zastanu i zamahuju, napadaju mačem. Svaki njihov udarac mačem smanjuje jedan život igraču, ali zato svaki igračev udarac mačem ih ubija. Ukoliko se igrač koristi metalnim zvijezdama potrebno je da 6 njih pogodi ljude crnih oklopa kako bi umrli. Šišmiši kao neprijatelji nisu tako brzi i opasni te su skriveni u spilji i imaju svega jedan život. Oni nemaju nikakav poseban napad osim što lete prema igraču i ukoliko ga uhvate on odmah izgubi sve živote. Kao zadnji neprijatelj i najjači je Meranox koji ima 20 života. On osim što patrolira svojim područjem ima iste sposobnosti napada kao i sam igrač, a to su napad mačem i bacanje metalnih zvijezdi. Za razliku od ljudi svoje rase ne zastaje samo prilikom kada mu se igrač dovoljno približi nego ga i prati i baca metalne zvijezde. Pobjedom i smrti Meranoxa predstavlja kraj same igre. Svi podaci koji su potrebni za prikaz broj skupljenih zlatnika, ubijenih neprijatelja i stanje igračeva života nalazi se u lijevom gornjem kutu scene. Život neprijatelja nije vidljiv u sceni osim Meranoxa čiji se život prikaže kada mu se igrač

približi i prikazan je u desnom gornjem kutu. Igra se sastoji od šest scena u kojoj prve tri scene se odnose na glavni izbornik, priču i odabir težine igranja, a ostale na tri nivoa.

3.2. Priča

Glavni lik igre je Maximus koji je izabran za borbu protiv neprijatelja i oslobodi planet Aris. Priča seže u daleku prošlost kada je cijeli planet bio pod vlasti Dironosa koji su kao rasa bili goblini. Oni su imali sluge Odixa i Rupexa koji su bili pripadnici i predstavnici ljudske rase i morali su raditi kako im je bilo naređeno. Ljudi su se u jednom trenutku odlučili pobuniti i istjerati gobline s planete Aris i u tome su i uspjeli. U čast slugama Odixa i Rupexa koji su bili najviše zaslužni nastale su dvije ljudske rase po njihovim imenima. Prva rasa je bila Odix, a druga Rupex. Oni se nisu razlikovali samo po imenu nego i po oružju i oklopu koje su nosili. Prva rasa je bila prepoznatljiva po sivoj boji dok je druga koristila crnu kao svoj zaštitni znak. Zajedno su te dvije rase živjele skladno i međusobno su si pomagale, ali ne zadugo. Uskoro su se goblini (rasa Diranos) vratili na Aris pod izgovorom da im dopuste kraći boravak na planeti kako bi prikupili dovoljno resursa koji su im potrebni da omoguće boravak na planeti na kojoj trenutno žive otkako su protjerani. Pripadnici rase Odix su shvatili i nisu nasjeli na prekriveni plan goblina koji su htjeli iscrpiti što više resursa kako bi ponovo mogli ojačati i zavladati Arisom. Oni su bili složni u tome, ali je postojao jedan problem, a taj problem su bili rasa Rupex koji su u to vrijeme bili vođeni Meranoxom, vođom kojemu je bilo dosta rase Odixa pošto su bili mnogobrojnija rasa i time više utjecali na donošenje zajedničkih odluka. Meranox je imao u planu da zajedno s goblinima sklopi savez i porazi Prvu rasu. To nikako nije mogao dopustiti Maximus koji je bio vođa rase Odix i koji je znao da bi to bilo presudno za njihov opstanak. On je pokušao urazumiti Meranoxa da ne sklopi savez s goblinima jer će u konačnici to biti pogubno i za njega i njegovu rasu, ali on to odbija i ne prihvaća vođen svojom mržnjom i zavidnošću prema rasi Odix. Na kraju Maximusu ništa drugo ne preostaje nego da krene u borbu i pokuša poraziti gobline i Meranoxa te tako zauvijek osloboditi planet Aris. Porazom Meranoxa predstavlja ujedno i kraj same igre.

3.3. Kamera

Kako bi mogli vidjeti kako bi izgleda naša scena kada bi igrali potrebna nam je kamera. Kamera predstavlja uređaj koji prikazuje svijet igraču i brojne mogućnosti u Inspektoru. U jednoj sceni može biti jedna ili više kamera koje su usmjerene na određeni dio scene. Kameru se može pomicati kao i ostale objekte u sceni, mogu se dodavati razne skripte i komponente. Neka svojstva kamere su da se može postaviti boja pozadine koja može značajno utjecati na preglednost scene i njezinu vidljivost, udaljenost kamere od scene koja se može približiti ili udaljiti, zaslon na koji kamera pokazuje (broj zaslona je od 1 do 8) , slušatelj zvuka (*Audio Listeiner*) koji služi za dohvaćanje svih zvukova koji postoje u sceni i brojne druge mogućnosti. Jedna od bitnijih značajki kamere je vrsta projekcije koja može biti postavljena na perspektivu (namijenjena za prikaz dubine, koristi se za 3D igre) i ortografska projekcija (namijenjena za 2D igre). U ovoj igri se u svakoj sceni koristi samo jedna kamera postavljena na ortografsku projekciju i koja prati igrača. Kako bi kamera znala koji objekt treba pratiti potrebno je dodati skriptu i napisati odgovarajući kod. U nastavku slijedi kod i objašnjenje.

```
Public class Camera : MonoBehaviour {

    public GameObject objectToFollow;
    private Vector3 cameraOffSet;
       // Use this for initialization
       void Start () {
        cameraOffSet = transform.position – objectToFollow.transform.position;
        Cursor.visible = false;
       }
    void FixedUpdate()
    {
        transform.position = objectToFollow.transform.position + cameraOffSet;

        if (transform.position.y <= -5)
            transform.position = new Vector3(transform.position.x, -5f,transform.position.z);

     if (Input.GetKey(KeyCode.C) && objectToFollow.GetComponent<Rigidbody2D>().velocity.x== 0)
           transform.position = new Vector3(transform.position.x, transform.position.y – 4f,
            transform.position.z) ;
}
```

Sl. 3.1. Skripta Camera

U skripti prije samih funkcija su definirane dvije varijable. Prva je javna varijabla tipa objekt koja se zove *objectToFollow* i koristi se da znamo koji objekt treba pratiti u sceni. Druga je privatna varijabla naziva *cameraOffSet* i tipa vektor3 koja ima sadrži tri pozicije (širinu, visinu i dubinu) koja služi za postavljanje udaljenosti od objekta kojega se prati. U funkciji *Start()* koja niša ne vraća se postavlja *cameraOffSet* da odredi udaljenost, odnosno položaj između same kamere i objekta kojeg treba pratiti, zatim slijedi naredba *Cursor* koja je ugrađena funkcija Unity-a i pomoću koje se može maknuti prikaz pokazivača miša što je i učinjeno, točnije postaviti vidljivost kao istina ili laž. U idućoj funkciji koja također ne vraća nikakvu vrijednost je postavljeno da položaj kamere bude kao položaj objekta udaljen za odgovarajuću vrijednost. Nakon toga slijedi *if* naredba grananja koja provjerava koji je položaj kamere po y osi i kao je on manji ili jedan vrijednosti -5 onda se izvršava linija unutar petlje koja zadaje novi položaj kamere s vrijednostima na x i z osi koje je imala, ali s novom vrijednošću na y osi koja je definirana u naredbi grananja. Ispod se zatim nalazi druga naredba koja ima dva argumenta koji trebaju biti istiniti kako bi se izvršila. Prvi argument je ako je pritisnuta tipka C na tipkovnici i (pri čemu se kao veznik koristi operator i [&&]) ako je brzina kretanja objekta po x osi jednaka nuli onda se može izvršiti kod unutar naredbe koji radi isto što i prijašnja naredba samo s drugom vrijednosti na -4 na y osi pri čemu se koristi i slovo f (*float*) kraj broja što označava da je broj decimalne vrijednosti.

Sl. 3.2. Prikaz postavljanja u Inspektoru objekta kojeg kamera treba pratiti u sceni

Kao što možemo vidjeti na slici iznad skripta Camera ima jednu javnu varijablu *ObjectToFollow* koju je potrebno odrediti. To možemo putem miša tako što povučemo objekt u prazan prostor ili stisnuti lijevim klikom na okruglu oznaku na rubu desne strane i odabrati objekt koji želimo staviti. U konačnici smo time postavili koji objekt želimo da kamera prati u sceni.

3.4. Likovi

Kao što je već spomenuto u potpoglavlju ideje (3.1. Ideja) postoje četiri vrste protivnika, likova. Kako bi protivnici i igrač mogli raditi određene kretnje potreban je kod koji će to omogućiti. Svaki lik ima više skripti, ali u nastavku će biti objašnjene samo osnovne skripte. Skripte su također bitne i za kontrolu i pokretanja animacija kada se ostvari određeni događaj. Neke su skripte iste koje se koriste za igrača i protivnike, ali većinom su različite. Ispod možemo vidjeti primjer skripte koja služi za kontrolu objekata (u ovom slučaju oružja) koji se bacaju i skripta koja je gotovo ista za igrača i protivnike.

```
Public class projectileShoot : MonoBehaviour {

public float projectileSpeed;
private Player player;


// Use this for initialization
void Start () {
player = FindObjectOfType<Player>();
if (player.transform.localScale.x < 0)
projectileSpeed = -projectileSpeed;


}

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

GetComponent<Rigidbody2D>().velocity = new
Vector2(projectileSpeed,GetComponent<Rigidbody2D>().velocity.y);

} }
```

Sl. 3.3. Skripta porjectileShoot

Na početku su definirane dvije varijable od koje se jedna odnosi na brzinu objekta koji se ispucava i druga na nasljeđivanje metoda i varijabli iz skripte Player koja je dodana na igrača. U funkciji *Start()* se definira varijabla *player* na način da se u sceni potraži objekt koji sadrži skriptu Player i pridruži toj varijabli. Odmah ispod je *if* naredbe koja provjerava položaj igrača po x osi, točnije smjer u kojem je igrač okrenut. Ukoliko je okrenut u desnu stranu vrijednost x je 1, a ako je okrenut u lijevu stranu iznos x-a je -1 s time se mijenja i smjer projektila. Unutar *Update()* funkcije se nalazi kod koji dohvaća komponentu Rigidbody2D koji omogućuje fiziku (gravitaciju, kretnju, masu, itd.) objekta i dodaje joj novu vrijednost vektor2. Vektor2 sadrži dva

argumenta i u ovom slučaju se predaje argument brzina projektila i kao drugi argument brzina kretanja objekta.

3.4.1. Igrač (Maximus)

Sl. 3.4. Sprite igrača (Maximusa)

Maximus je glavni lik kojim upravlja igrač i pripada rasi Odix (sivi oklopi) kojeg možemo vidjeti na slici prikazanoj iznad (Sl. 3.4.). Igrač se može kretati lijevo-desno kao i skočiti u zrak te bacati metalne zvijezde (*Shuriken-e*) i napadati mačem. Sve kretnje koje su moguće su popraćene odgovarajućom animacijom. Sve slike koje su se koristile za izradu animacije igrača se nalaze u prilozima. Kao glavna skripta koja je dodana na igrača naziva se Player i sadrži gotovo sve potrebne funkcije. U nastavku možemo vidjeti neke od funkcija kao i njihova objašnjenja. Prvi dio koji slijedi odnosi ne na kretanje igrača (Sl.3.5.) pri čemu su se koristile tri varijable tipa *float*. *MaximusSpeed* se odnosi na brzinu kojom želimo da se igrač kreće zatim *jumpStrenght* koja predstavlja jačinu skoka i zadnja *moveVelocity* kao varijabla kojoj će biti dodijeljena brzina i smjer kretanja. U drugom dijelu imamo pet varijabli koje se koriste za određivanja kada je igrač na platformi (koja predstavlja zemlju, čvrstu podlogu).

```
Public class Player : MonoBehaviour {

    public float MaximusSpeed;
    public float jumpStrenght;
    private float moveVelocity;


    public Transform groundCheck;
    public float groundCheckRadius;
    public LayerMask whatIsGround;
    public bool grounded;
    public bool jumpTwice;

void FixedUpdate()
    {

       grounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius,
       whatIsGround);

        if (grounded)
            jumpTwice = false;

        moveVelocity = MaximusSpeed * Input.GetAxisRaw(„Horizontal“);


       GetComponent<Rigidbody2D>().velocity = new Vector2(moveVelocity,
       GetComponent<Rigidbody2D>().velocity.y);

        if (Input.GetKeyDown(KeyCode.Space) && grounded)
        {
            GetComponent<Rigidbody2D>().velocity = new
            Vector2(GetComponent<Rigidbody2D>().velocity.x, jumpStrenght);
        }

        if (Input.GetKeyDown(KeyCode.Space) && !grounded && !jumpTwice)
        {
            GetComponent<Rigidbody2D>().velocity = new
            Vector2(GetComponent<Rigidbody2D>().velocity.x, jumpStrenght+1f);
            jumpTwice = true;
        } } }
```

Sl. 3.5. Prvi dio skripte Player

Varijabla *gorundCheck* omogućuje korištenje objekta koji je dijete (potomak) igrača (Maximusa) koja služi kao provjera kada je igrač na platformi i smještena je u donjoj razini igračevih nogu. *GroundCheckRadius* kao što i sam naziv varijable upućuje da se odnosi na polumjer koji služi za provjeru kada je igrač na čvrstoj podlozi. Nakon toga imamo *whatIsGround* tipa sloja maske (*LayerMask*) koju možemo definirati i dodati u Inspektoru na svaki objekt koji želimo da bude platforma. Nakon toga slijede dvije varijable tipa operatora *bool* koje mogu sadržavati vrijednost 1 ili 0 (istina ili laž), a odnose se je li igrač na čvrstoj podlozi (*grounded*) i je li skočio dva puta (*jumpTwice*). Unutar funkcije *FixedUpdate() grounded* pridjeljujemo vrijednost na temelju

korištenja klase *Phzsics2D* koja sadrži statičku funkciju *OverlapCircle* kojoj su potrebna tri argumenta. Prvi argument je položaj objekta za provjeru, drugi argument je polumjer provjere i treći sloj maske što predstavlja platformu pri čemu rezultat može biti istinit ili lažan. Odmah ispod imamo naredbu *if* koja provjerava je li *grounded* istinita tvrdnja i ako je postavlja se *jumpTwice* na *flase* (laž) kako bi bilo omogućeno da igrač može još jednom skočiti, pošto je zamisao da igrač može dva puta za redom skočiti. Zatim slijedi dodjeljivanje vrijednosti varijabli *moveVelocity* pomoću korištenja varijable *MaximusSpeed* i ulaza koji se definira nizom znakova (*string)* u navodnicima pri čemu „Horizontal" je definirani niz znakova u Unity-u koji koriste tipke tipkovnice A i D (A predstavlja negativnu vrijednost [-1], a D pozitivnu vrijednost [1]) i koje možemo ukoliko želimo promijeniti i ako istovremeno pritisnemo tipke A i D dobivamo vrijednost 0 pri čemu se igrač onda neće moći kretati. Nakon toga slijede naredbe za dohvaćanje komponente 2D fizike (*Rigidbody2D*) koja omogućuje kretanje. Na kraju slijede dvije *if* naredbe koje kontroliraju kada je igrač skočio jednom, a kada dva puta.

Drugi dio skripte (Sl. 3.6.) sadrži definirane varijable koje su potrebne da bi igrač mogao napadati metalnim zvijezdama i mačem te u kojem periodu to može učiniti, odnosno koliko često. Vrijeme napada određuju varijable *fireRate* (koliko često može se može pucati) i *lastShot* (posljednji pucanj). Tipka koja se koristi za bacanje metalnih zvijezdi je lijevi klik miša (*Mouse0*), a desni klik miša se koristi za napad mačem (*Mouse1*). U prvoj *if* naredbi se provjerava ako je pritisnuta lijeva tipka miša, ako je apsolutna vrijednost kretanja igrača jednaka nuli i ako je proteklo vrijeme veće od vremena napada onda se omogućuje napad i stvara se objekt u sceni pomoću statične funkcije *Instantiate* koja klonira izvorni objekt i koristi kopiju. U toj funkciji je potrebno predati argumente, koji objekt želimo koristiti, na kojem mjestu i njegovu rotaciju. Druga naredba grananja je kratka i provjerava je li pritisnuta lijeva tipka i ukoliko nije postavlja se *spikeAttack* na vrijednost 0, false (laž). Treća naredba *if* se koristi za napad mačem. Provjerava se ako je pritisnuta desna tipka miša i ako je apsolutna vrijednost kretanja igrača jednaka nuli onda će napad mačem biti omogućen i postavljen na *true* (istinu). U zadnjoj petlji se provjerava ako nije pritisnuta desna tipka ili (operator ili vraća istinu ako je barem jedna tvrdnja točna) ako je apsolutna vrijednost kretanja igrača različita od nule. Nadalje, ako je tvrdnja u petlji točna, biti će onemogućeno korištenje mača.

```
Public class Player : MonoBehaviour {

    public Transform shurikenSpawn;
    public GameObject shuriken;

    private bool swordAttack;
    public GameObject sword;

    private bool spikeAttack;


    private float fireRate = 0.1f;
    private float lastShot = 0;


        if (Input.GetKeyDown(KeyCode.Mouse0) &&
Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) == 0 && Time.time > fireRate + lastShot)
        {
            spikeAttack = true;
            Instantiate(shuriken, shurikenSpawn.position, shurikenSpawn.rotation);
            lastShot = Time.time;
        }

        if (!Input.GetKeyDown(KeyCode.Mouse0) )
        {
            spikeAttack = false;
        }

        if (Input.GetKeyDown(KeyCode.Mouse1) &&
       Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) == 0 )
        {
            swordAttack = true;
        }



        if ((!Input.GetKeyDown(KeyCode.Mouse1) ||
       Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) != 0))
        {
            swordAttack = false;
        } }
```

Sl. 3.6. Drugi dio skripte Player

Sada će biti prikazana skripta Particles u kojoj su nasljeđene neke varijable iz skripte Player. Kako bi uopće mogli naslijediti neku klasu potrebno je da su javne kao i njihove varijable ili metode koje se nasljeđuju. U ovom postupku klasa Player predstavlja nadklasu ili baznu klasu, a klasa koja nasljeđuje moguća svojstva nadklase zove se podklasa. Ukoliko su privatne njihovo nasljeđivanje (dohvaćanje) unutar druge skripte nije moguće. Način na koji definiramo novu varijablu koja će naslijediti varijablu nadklase je taj da prvo moramo definirati hoće li ta varijabla biti javna ili privatna, a zatim točan naziv nadklase i zatim ime varijable. Kako bi sve to bilo razumljivije i jasnije možemo vidjeti kako to izgleda u kodu ispod (Sl. 3.7.).

```
public class Particles : MonoBehaviour {

    public GameObject walkingPatricle;
    private bool particleTrigger;

    private Player groundCheckHeritage;
    private Player groundCheckRadiusHeritage;
    private Player whatIsGroundHeritage;
    private Player groundedHeritage;

       void Start () {
        groundCheckHeritage = FindObjectOfType<Player>();
        groundCheckRadiusHeritage = FindObjectOfType<Player>();
        whatIsGroundHeritage = FindObjectOfType<Player>();
        groundedHeritage = FindObjectOfType<Player>();

       }

    void FixedUpdate()
    {
        groundedHeritage.grounded =
       Physics2D.OverlapCircle(groundCheckHeritage.groundCheck.position,
       groundCheckRadiusHeritage.groundCheckRadius, whatIsGroundHeritage.whatIsGround);

        if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.D))
        {
            particleTrigger = true;
        }

        if (!Input.GetKey(KeyCode.A) && !Input.GetKey(KeyCode.D))
        {
            particleTrigger = false;
        }

        if (Input.GetKey(KeyCode.Space) || !groundCheckHeritage.grounded)
            particleTrigger = false;

        if (Input.GetKey(KeyCode.A) && Input.GetKey(KeyCode.D))
        {
            particleTrigger = false;
        }

        if (particleTrigger)
            walkingPatricle.SetActive(true);
        else
            walkingPatricle.SetActive(false);

    }

}
```

Sl. 3.7. Skripta Particles

Nakon definiranih varijabli koje želimo da naslijede varijable iz skripte Player gdje ih u ovom slučaju ima četiri potrebno je u funkciji *Start()* postaviti koje objekte treba pronaći u sceni, točnije s kojom skriptom (vidimo da je to sa skriptom Player). Kako smo time definirali odakle će se naslijediti svojstva možemo ih u funkciji *FixedUpdate()* koristiti i to pomoću naziva naše

varijable podklase, zatim stavimo točku (operator koji omogućuje dohvaćanje) i naziv varijable koju želimo koristiti iz nadklase. U ovoj skripti u funkciji dalje slijedi kod koji provjerava kada je igrač na platformi, a kada je u zraku isto kao i u skripti Player. U *if* naredbama se provjerava kada je stisnuta tipka A ili D da se onda omogući *particleTrigger* (postavlja se na istinu) koji će aktivirati čestice da se stvaraju kada igrač hoda. Također petlje provjeravaju i kada igrač drži obje tipke istovremeno ili je u zraku da se postavi *particleTrigger* na laž i time onemogući stvaranja čestica.

3.4.2. Neprijatelji

Kao što je već prije spomenuto postoje četiri vrste s kojima se Maximus mora suočiti, a to su:

a) Goblin

Sl. 3.8. Sprite goblina

Goblini su neprijatelji koji kao oružje koriste sjekire koje kako je već prije spomenuto i koje mogu bacati. Oni su postavljeni na određene platforme po kojima hodaju lijevo-desno i ukoliko im se igrač dovoljno približi stanu i počinju bacati sjekire. Kada se igrač dovoljno udalji oni nastavljaju kretati se lijevo-desno po platformi. Kako bi oni mogli izvoditi sve navedene aktivnosti potrebna je skripta koja sadrži kod koji to omogućuje. Za razliku od lika Maximusa neprijatelje nije moguće kontrolirati i upravljati od strane igrača.

```
public class EnemyMovement : MonoBehaviour {

    public float enemySpeed;
    public bool moveLeft;

    public Transform wallCheck;
    public float wallCheckRadius;
    public LayerMask whatIsWall;
    private bool hittingWall;
    private bool edge;
    public Transform edgeCheck;

    public Transform axeSpawn;
    public GameObject axe;
    public GameObject axeLeft;

    private float fireRate = 2f;
    private float lastShot = 0;

    private bool axeActivator;
    public GameObject player;
    public Transform axeSpawnRight;


    // Use this for initialization
    void Start()
    {
                moveLeft = true;

    }

    void FixedUpdate()
    {
        hittingWall = Physics2D.OverlapCircle(wallCheck.position, wallCheckRadius,
                      whatIsWall);
        edge = Physics2D.OverlapCircle(edgeCheck.position, wallCheckRadius, whatIsWall);

        if (hittingWall || !edge)
            moveLeft = !moveLeft;

        if (moveLeft)
        {
            transform.Translate(Vector2.left * -enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(-1f, 1f, 1f);

        }
        if (!moveLeft)
        {
            transform.Translate(Vector2.right * enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(1f, 1f, 1f);
        }
```

```
            if (axeActivator && Time.time > fireRate + lastShot)
        {

            if (player.transform.position.x < transform.position.x && axeSpawn)
            {
                Instantiate(axeLeft, axeSpawn.position, axeSpawn.rotation);
            }

            if (player.transform.position.x > transform.position.x && axeSpawn)
            {
                Instantiate(axe, axeSpawnRight.position, axeSpawnRight.rotation);
            }
            lastShot = Time.time;
        }


        if (axeActivator && player.transform.position.x < transform.position.x )
        {

            transform.localScale = new Vector3(-1f, 1f, 1f);

        }


        if (axeActivator && player.transform.position.x > transform.position.x)
        {

            transform.localScale = new Vector3(1f, 1f, 1f);
        }
    }
}
```

Sl. 3.9. Skripta za kretanje goblina

U ovoj skripti se nalazi cijeli kod koji je potreban za kretanje i napadanje. Kako ne bismo objašnjavali cijeli postupak korištenja nekih varijabli koje su već prije objašnjene samo kratko ćemo napomenuti koje su to i čemu služe. Prve dvije se odnose na brzinu kretanja i smjer dok zatim skupina nakon od 6 varijabli se odnosi na provjeru što je zid i gdje je kraj platforme kako bi neprijatelj znao da se treba okrenuti u drugom smjeru te kako ne bi pao s platforme. Koristi se isti princip određivanja vrijednosti varijabli *hittingWall* i *edge* kao što je bilo kod igrača kod varijable *grounded* (pogledati Sl. 3.5.). Ono što je bitno u ovoj skripti je način na koji neprijatelj zna kada treba promijeniti smjer kretanja i ujedino promijeniti *Sprite* u istu stranu kako bi dobili smisleno kretanje. Za to se brine varijabla *moveLeft* čije je vrijednost istinita kada se neprijatelj kreće u lijevu stranu, a neistina kada se kreće u desnu stranu te time se daje do znanja na koju stranu mora *sprite* igrača biti okrenut. Ukoliko je okrenut ulijevo brzina je postavljena na negativnu vrijednost i izgled na -1f što znači također neprijatelj gleda u lijevu stranu, ukoliko je okrenut u desno sve je obrnuto (vrijednost brzine je pozitivna, a izgled postavljen na 1f). Kako bi se znalo gdje je položaj i u koju stranu treba goblin bacati sjekiru provjerava se u petlji odnos

položaja igrača i samoga neprijatelja. Ako je igrač s lijeve strane goblina imat će manju vrijednost položaja po x osi i time će znati da se treba postaviti goblina okrenutog u lijevu stranu kao i smjer bacanja sjekire. S druge strane ako je igrač s desne strane u odnosu na goblina onda vrijednost položaja goblina manja i on se okreće u desnu stranu. Osim svega navedenoga ujedino se provjerava i vrijeme bacanja sjekire koje je definirano na svake dvije sekunde kako bi igrač mogao reagirati i izbjeći napad.

Još jedna skripta koja se tiče goblina i samih neprijatelja je EnemyKill koja sadrži kod koji provjerava koliko neprijatelj ima života i koliko mu je još ostalo ovisno o oružju kojim je napadnut.

```
public class EnemyKill : MonoBehaviour {

    public GameObject enemy;

    public AudioSource kill;
    private static int counter;
    public Text EnemyDeath;

    public GameObject enemyDeathParticle;
    private static int enemyLife;
    public GameObject shurikenParticle;

    public AudioSource enemyShootSound;
    private GameObject axe;


       // Use this for initialization
       void Start () {
        counter = PlayerPrefs.GetInt("CurrentEnemyDeath");
        ScoreDisplay();
        enemyLife = 3;

       }


    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("Sword"))
        {
            counter++;
            Instantiate(enemyDeathParticle, this.transform.position, this.transform.rotation);
            enemy.SetActive(false);
            kill.Play();
            ScoreDisplay();
            Destroy(axe);
            PlayerPrefs.SetInt("CurrentEnemyDeath", counter);
        }
```

```
        if(other.gameObject.CompareTag("Shuriken"))
        {
            if (enemyLife == 0)
            {
                enemyShootSound.Stop();
                counter++;
                Instantiate(enemyDeathParticle, this.transform.position,
                this.transform.rotation);
                enemy.SetActive(false);
                kill.Play();
                ScoreDisplay();
                enemyLife = 3;
                Destroy(axe);
                PlayerPrefs.SetInt("CurrentEnemyDeath", counter);
            }
            if(enemyLife > 0)
            {
                enemyShootSound.Play();
                Instantiate(shurikenParticle, other.transform.position,
                other.transform.rotation);
            }
            enemyLife--;

        }
    }

    void ScoreDisplay()
    {
        EnemyDeath.text = "x " + counter.ToString();
    }

    void FixedUpdate()
    {
        counter = PlayerPrefs.GetInt("CurrentEnemyDeath");

    }
}
```

Sl. 3.10. Skripta za dodjeljivanje i provjeru života neprijatelja

Na početku ponovo definirane varijable koje se odnose na zvukove prilikom kada je neprijatelj pogođen, čestica koje će se stvoriti kada je neprijatelj ubijen, životi koji su cjelobrojnog tipa, brojači koji služe za brojanje ubijenih neprijatelja. U funkciji *Start()* je postavljen brojač koji uzima vrijednost trenutno ubijenih neprijatelj koristeći pritom klasu *PlayerPrefs* koja omogućuje pohranu podataka između scena. Nakon brojača imamo funkciju *ScoreDisplay()* koja prikazuje broj ubijenih neprijatelja u sceni i zadnje je definiran broj života u ovom slučaju goblina na 3. U nastavku slijedi ugrađena poruka *OnEnterTrigger2D* koja kao argument koristi klasu *Collider2D* i time se može odrediti ukoliko objekt dođe u dodir (koliziju) s drugim objektom. To nam je bitno jer onda možemo odrediti kada je neprijatelj pogođen ili udaren oružjem od strane igrača. Zatim slijede dvije petlje koje provjeravaju je li došlo do kolizije s mačem ili metalnom zvijezdom. Ukoliko je vrijednost unutar naredbe grananja istinita izvršava se niz naredbi koje

aktiviraju zvuk neprijateljeve smrti, stvara čestice, povećava brojač za jedan, neprijatelja čini nevidljivim u sceni, poziva se funkcija *ScoreDisplay()* kako bi prikazala novi broj ubijenih neprijatelja i na kraju spremanje tog broja u klasu *PlayerPrefs* koja će prenositi trenutni broj kako bi se mogao uzeti i nastaviti koristiti u drugoj sceni. U drugoj petlji se provjerava ako je ostvaren kontakt s metalnom zvijezdom i pri tome se izvode slične naredbe i sadrži još dodatne dvije *if* naredbe grananja koje provjeravaju broj života. Prva unutarnja *if* naredba provjerava ako je broj života nula i pri tome se onda izvršava kod isti kao i kod dodira s mačem dok u drugoj unutarnjoj petlji provjerava je li broj života veći od 0 i ako je pokreni zvuk udarca zvijezde te stvori česticu udarca u neprijatelja. Nakon te dvije naredbe svaki put se izvršava umanjivanje života neprijatelja za jedan.

b) Ljudi crnih oklopa

Sl. 3.11. Sprite neprijatelja rase Rupex

Ljudi crnih oklopa koriste mač kao oružje i također patroliraju platformama kao i goblini. Napadaju igrača samo ako im dođe dovoljno blizu da ga mogu udariti mačem. Imaju gotovo istu skriptu za kretanje kao i goblini te zbog toga neće biti ponovo prikazana i objašnjavana. Također imaju istu skriptu za kao što je EnemyKill jedino u čemu se razlikuje su postavljeni životi, a imaju ih 6. Oni predstavljaju ljude rase Rupex koji su odani svome vođi Meranoxu te zbog toga napadaju Maximusa koji treba doći do vođe i poraziti ga.

c) Meranox

Sl. 3.12. Sprite Meranoxa

Meranox predstavlja zadnjeg neprijatelja u igri Return of Darkness kojeg treba Maximus, točnije igrač poraziti. Nakon njegova poraza otvara se novi prozor koji sadrži konačan rezultat, broj skupljenih zlatnika i broj ubijenih neprijatelja i na kraju gumb za povratak u glavni izbornik. Za ovog lika su također korištene neke iste skripte kao i za prethodna dva neprijatelja, ali ima i neke značajnije promjene koje će biti prikazane u kodu ispod te također objašnjene.

```
public class MeranoxMovement : MonoBehaviour {

//Chasing target AI
    public Transform target;
    public bool chase;

   void Start()
    {
        chase = false;
    }

void FixedUpdate()
    {
        if (moveLeft && !chase)
        {
            transform.Translate(Vector2.left * -enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(-1f, 1f, 1f);
        }
```

```
        if (!moveLeft && !chase)
        {
            transform.Translate(Vector2.right * enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(1f, 1f, 1f);
        }

        //Chase AI
        if (target.position.x < transform.position.x && chase)
        {

            Vector3 targetDirection = target.position - transform.position;
            float angle = Mathf.Atan2(targetDirection.y, targetDirection.x) *
                          Mathf.Rad2Deg - 180f;
            Quaternion q = Quaternion.AngleAxis(angle, Vector3.forward);
            transform.rotation = Quaternion.RotateTowards(transform.rotation, q, 0);

            transform.Translate(Vector2.left * -enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(-1f, 1f, 1f);
        }

        if (target.position.x > transform.position.x && chase)
        {

            Vector3 targetDirection = target.position - transform.position;
            float angle = Mathf.Atan2(targetDirection.y, targetDirection.x) *
                          Mathf.Rad2Deg - 180f;
            Quaternion q = Quaternion.AngleAxis(angle, Vector3.forward);
            transform.rotation = Quaternion.RotateTowards(transform.rotation, q, 0);

            transform.Translate(Vector2.right * enemySpeed * Time.deltaTime);
            transform.localScale = new Vector3(1f, 1f, 1f);
        }
       }

    private void OnTriggerEnter2D(Collider2D other)
    {

        if (other.gameObject.CompareTag("AxeActivator"))
        {
            chase = true;
            Debug.Log("ChaseRange");
        }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("AxeActivator"))
        {
            chase = false;
            Debug.Log("Not ChaseRange");
       }
    }
}
```

Sl. 3.13. Skripta Meranoxa

Skripta iznad (Sl. 3.14.) sadrži kod koji služi za praćenje i napadanje igrača kada dođe na određenu udaljenost u odnosu na Meranoxa. Odmah u početku su definirane tri varijable pri čemu *target* služi za definiranje cilja, *chase* kao *bool* varijabla koja se koristi da se zna kada je igrač u dometu. U funkciji *Start()* je postavljena varijabla *chase* na neistinu kako bi Meranox patrolirao područjem. U drugoj funkciji *FixedUpdate()* imamo četiri *if* naredbe. Prve dvije naredbe provjeravaju u koju stranu se kreće Meranox i je li postavljena varijabla *chase* na laž i ako je nastavlja s patroliranjem do idućeg kraja platforme ili zida te se okreće. Zatim slijede dvije naredbe grananja koje provjeravaju odnos položaja igrača u odnosu na Meranoxa kao i kod skripte koja se koristi za gobline te se izvršava isti dio koda za okretanje neprijatelja prema igraču u točnom smjeru, ali ima i dodatne naredbe koje će biti objašnjene u nastavku. Prvo se definira lokalna varijabla tipa vektor3 naziva *targetDirection* koja određuje smjer onda se definira kut koji ima decimalnu vrijednost pri čemu se koristi ugrađena funkcija Mathf.Atan2 koja vraća vrijednost tangens kuta y/x u radijanima gdje se trebaju predati dva argumenta između koji se računa kut, a to je položaj mete po x i y osi i na kraju se pomoću funkcije Mathf.Rad2Deg pretvara u stupnjeve. Onda slijedi naredba gdje se koristi struktura *Quaternion* koja u najjednostavnijem objašnjenju omogućuje rotaciju pri čemu se Meranox mora pravilno okrenuti prema igraču (izjednačuje se s *Quaternion.AngleAxis* kojem se predaje kut i vetkor3 kako bi došlo do rotacije za predane stupnjeve oko osi). Na samome kraju se definira varijabla *transform* pri čemu se rotacija objekta izjednačuje sa strukturom *Quaternion.RotateTowrads* kojoj se predaje rotacija objekta, kut osi i kao treći parametar vrijednost nula koja omogućuje trenutnu transformaciju Meranoxa. U kodu se još nalaze dvije poruke *OnTriggerEnter2D* i *OnTriggerExit2D* koje se koriste za određivanje kada je igrač došao dovoljno blizu Meranoxu kako bi ga pratio ili se je dovoljno udaljio kako bi prestao pratiti igrača i nastaviti s patroliranjem. Ukoliko igrač dođe dovoljno blizu provjerava se je li došlo do kolizije s objektom koji je potomak objekta igrača i naziva „AxeActivator“ izvršit ću se naredba koja postavlja varijablu *chase* na istinu i zatim će Meranox početi pratiti igrača. S druge strane, ako se izađe iz kolizije vezane za iste objekte izvršit će se ponovo ista naredba, ali s postavljenom vrijednosti na laž pri čemu se Meranox prekinuti pratiti igrača i nastaviti patrolirati područjem.

d) Šišmiš

Sl. 3.14. Sprite šišmiša

Šišmiši predstavljaju jedine neprijatelje koji lete i ne koriste nikakvo oružje. Napad se sastoji od letenja i hvatanja igrača i ukoliko dođe do njega ubija ga. Postoje svega dva šišmiša u igri i to u trećem nivou. Oni se nalaze skriveni u spilji kako je opće poznato da se skrivaju i borave po spiljama. Imaju tri života kao i goblini, ali su sporiji od ostalih kako bi ih igrač mogao lakše ubiti. Na njih su dodane tri skripte, prva je Bat_movement koja će biti objašnjena, druga je EnemyKill koja je već ranije objašnjena (Sl.3.10.) i zadnja DeathOnBat koja provjerava je li došlo do kolizije između igrača i šišmiša i ako je igrač će biti mrtav.

```
public class Bat_movement : MonoBehaviour {

    public float batSpeed;
    public Transform target;

    private bool trigger;
       // Update is called once per frame
       void Update () {

        if (trigger)
        {
            transform.LookAt(target.transform.position);
            transform.Rotate(0, -90, 0);
            transform.Translate(Vector2.right * batSpeed * Time.deltaTime);
        }

       }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.CompareTag("FlowStoneActivator"))
        {
            trigger = true;
        }
    }
}
```

Sl. 3.15. Skripta Bat_movement

Bat_movement skripta služi da šišmiš prati igrača, ali u određenom trenutku, točnije kada se ostvari odgovarajući događaj (okidač). U početku su definirane dvije javne varijable pri čemu se prva odnosi na brzinu kretanja (*batSpeed*), a druga na definiranje objekta kojeg šišmiš treba pratiti. U konačnici treba nam još jedna varijabla tipa *bool* nazvana *trigger* koja će služiti za aktiviranje praćenja mete. Prvo ćemo objasniti već poznatu poruku *OnTriggerEnter2D* koja će provjeravati je li došlo do kolizije između šišmiša i objekta koji ima nadimak „FlowStoneActivator“ i ako je postavlja se okidač na istinu. Nakon toga u funkciji *Update()* u *if* naredbi provjerava ako je okidač (*trigger*) postavljen na istinu i ako je onda se izvršavaju naredbe unutar *if* naredbe. Prvo se izvršava naredba *transform.LookAt* kojoj se kao argument treba predati vektor3 i u ovom slučaju taj nam vektor zapravo predstavlja sama meta i zato dajemo poziciju mete pomoću *target.transform.position.* Zatim kako bi šišmiš gledao prema meti bitno je da bude pravilno rotiran i zato slijedi naredba *transform.Rotate* koja će dohvatiti transformaciju objekta na kojeg je postavljena skripta (a to je na šišmišu) i onda će postaviti na vrijednost koja je postavljena (0,-90,0) iz razloga kako bi se pravilno okrenuo objekt. I posljednja naredba u petlji je *transform.Translate* koja omogućuje kretanje.

3.5. Zamke

Kako bi igru učinili zanimljivijom dodane su zamke na koje igrač mora pripaziti kako ne bi izgubio život. Kao zamke postoje šiljci od kojih su neki statični, a neki pomični i na koje igrač ne smije stati. Druga vrsta zamke je kotrljajuće kamenje koje igrač mora preskočiti kako bi preživio. Postoje i platforme koje su drugačije boje kako bi bile uočljivije i na koje igrač ako stane počinju padati i zadnji tip zamki su sige koje padaju na igrača ukoliko prolazi ispod njih. Nešto ćemo više će biti rečeno o nekim skriptama koje su dodane na zamke te prikazati kod. Na stranici koja slijedi se nalaze slike navedenih zamki. Prvi na slici u šiljci, drugo i četvrto su platforme koje padaju, treće je okrugli kamen i zadnje što se nalazi na slici su sige.

Sl. 3.16. Slika zamki koje se nalaze u igri

Skripta koja je korištena za padajuće platforme ima definirane dvije varijable: *grountTrap* i *groundRigid*. U funkciji Start() smo naveli da u varijabli *groundRigid* bude sačuvana komponenta *Rigidbody2D* koja predstavlja fiziku objekta što je već ranije objašnjeno (poglavlje 3.3. Likovi). Funkcija *Update()* sadrži provjeru u petlji ako je položaj platforme koja pada manji od vrijednosti -20 po y osi da bude uništena kako ne bi cijelo vrijeme padala u sceni. Zatim se koristi poruka *OnCollisionEnter2D* gdje se provjerava je li došlo do kolizije između pojedinih objekata. Ako dođe do kontakta između igrača i platforme izvršava se naredba *groundRigid.isKinematic = false* koja omogućuje da platforma počinje padati. Sve ostale provjere služe kako bi platforma nestala ili ne bi zadržala na ostalim platformama koje se nalaze u sceni. Ispod se nalazi ukratko objašnjeni kod skripte GroundFall.

```
public class GroundFall : MonoBehaviour {

    public GameObject groundTrap;
    private Rigidbody2D groundRigid;
       // Use this for initialization
       void Start () {
        groundRigid = GetComponent<Rigidbody2D>();
       }

       // Update is called once per frame
       void Update () {
        if (transform.position.y < -20)
            Destroy(groundTrap);
       }
    void OnCollisionEnter2D(Collision2D other)
    {
        if (other.gameObject.CompareTag("Player"))
            groundRigid.isKinematic = false;
        if (other.gameObject.CompareTag("GroundFall"))
            GetComponent<Collider2D>().isTrigger = true;
        if (other.gameObject.CompareTag("Ground"))
            Destroy(groundTrap);
    }
```

Sl. 3.17. Skripta GroundFall

Još jedan kod koji će biti naveden i objašnjen vezan uz zamke se nalazi u skripti GroundMove koja je dodana na šiljke kako bi bili pokretni.

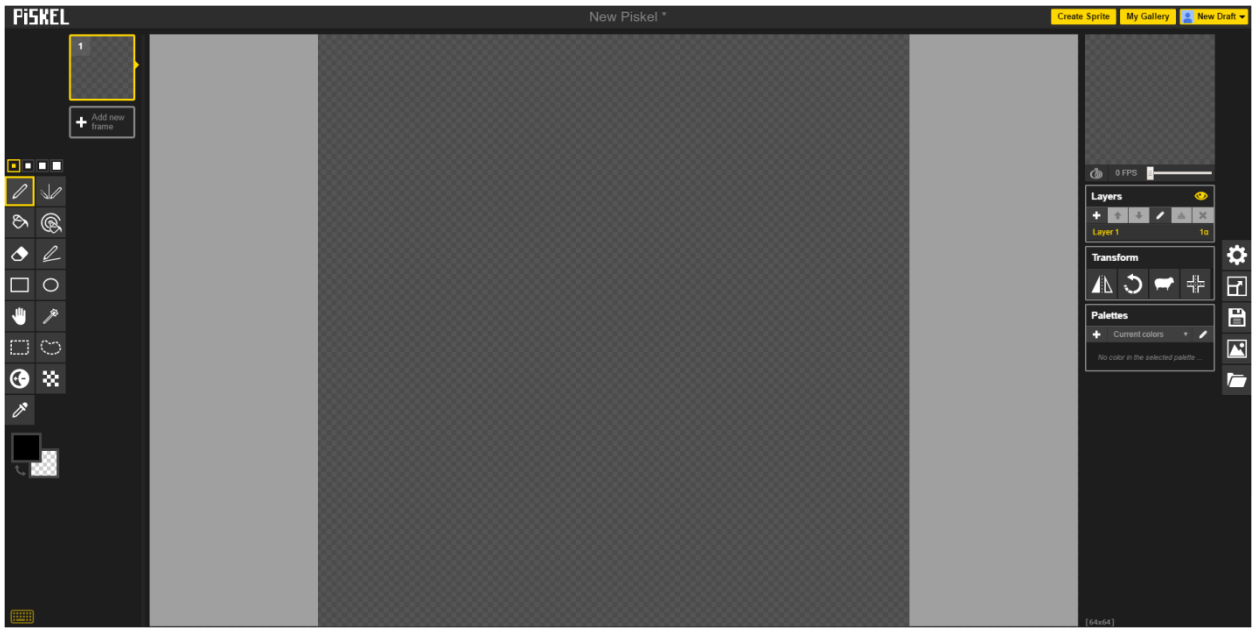
```
public class GroundMove : MonoBehaviour {


    public float groundSpeed;
    public GameObject ground;
    private Transform currentPosition;
    public Transform[] points;

    public int pointSelection;
       // Use this for initialization
       void Start () {
        currentPosition = points[pointSelection];
       }
       // Update is called once per frame
    void Update()
    {
        ground.transform.position = Vector3.MoveTowards(ground.transform.position,
currentPosition.position, Time.deltaTime * groundSpeed);

        if (ground.transform.position == currentPosition.position)
        {
            pointSelection++;

            if (pointSelection == points.Length)
            {
                pointSelection = 0;
            }

            currentPosition = points[pointSelection];
        }  } }
```

Sl. 3.18. Skripta GroundMove

Kako bi se mogli kretati šiljci potrebno je definirati brzinu kretanja za to nam služi varijabla *groundSpeed*, objekt koji će se pomicati, koji je trenutni položaj objekta te polje položaja kojim želimo da se objekt kreće. Samo polje se definira u Inspektoru gdje unosimo vrijednost koliko točaka želimo imati, odnosno putanju po kojoj se treba kretati objekt. Zatim u sceni stvaramo u ovom slučaju dva potomka istog objekta koji predstavljaju krajnje položaje. U *Start()* funkciji je postavljen trenutni položaj na kojem je objekt u definiranom polju položaja. Funkcija *Update()* sadrži naredbe koje omogućuju pomicanje objekta za definiranu brzinu u Inspektoru. U prvoj naredbi grananja *if* se provjerava ako je položaj objekta jedan trenutnoj poziciji u polju onda neka se poveća vrijednost pozicije za jedan (što se odnosi do iduće točke do koje treba doći objekt) i još jedna naredba koja provjerava ako je došlo do pozicije koja je zadnja u polju da se postavi na nulu i time objekt kreće unazad, točnije do početne pozicije i tako u krug.

3.6. Grafika

Što se tiče grafičkog izgleda igre sve osim pozadine u scenama je izrađeno u Piskel-u. Piskel[6] je aplikacija, besplatan uređivač za animiranje *sprite-ova* i slika u kojima se jasno vide pikseli (slike malih rezolucija, npr. 16x16 piksela). Može se koristi preko korisničkoga računa na internetu ili se može skinuti na računalu te se može crtati neovisno o internetskoj vezi. Vezano uz sliku, općenito što je veći broj piksela koji čine neku sliku to je sama slika bolje kvalitete pod time se smatra da ima više detalja i bolje prijelaze boja i slično. U ovoj aplikaciji se na jednostavan način mogu izraditi 2D slike jer je poprilično jednostavna za korištenje. Raspon broj piksela od kojih se slika može sastojati u Piskel-u je od 1 piksela pa do 2048 čak i više, ali tada crtanje postaje usporeno i isprekidano jer nije namijenjeno za slike tako velikih rezolucija. Samo sučelje možemo reći da se sastoji od tri dijela. Prvi dio su alati koji se nalaze s lijeve strane i koje služe za crtanje, drugi dio i to središnji bi bila podloga po kojoj se crta i treći koji se nalazi s desne strane se odnosi na postavke, promjenu veličine slike, spremanje slike, izvoz i uvoz slike. Svaka slika ili njih više se mogu spremiti u galeriju u samoj aplikaciji, ali samo ako imamo korisnički račun ili se mogu spremati na računalu neovisno o korisničkom računu. Svaka slika se može duplicirati ili brisati. U desnom gornjem kutu nalazi se prikaz nacrtanih slika koje se mogu prikazivati određenim brojem slika u sekundi (raspon je od 1 do 24 slike u sekundi). Takav nam prikaz može pomoći da vidimo kako bi izgledala naša animacija. Ispod možemo vidjeti samo sučelje aplikacije.

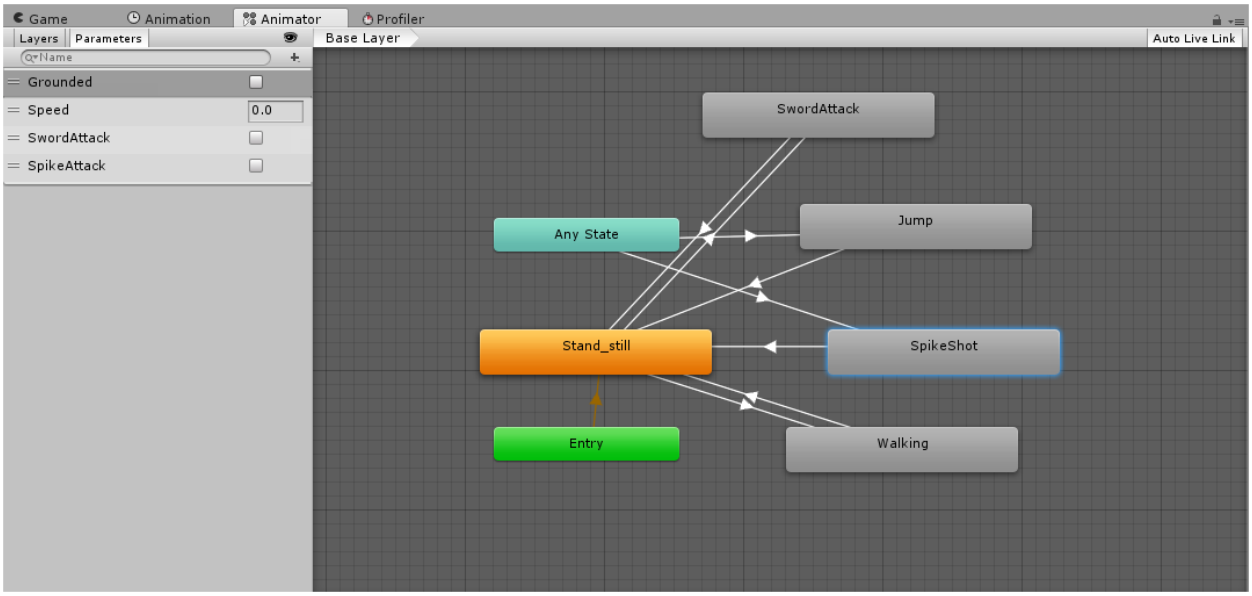


Sl. 3.19. Prikaz sučelja aplikacije Piskel

3.7. Animacije i Animator

Kako bi svaka igra bila privlačnija igračima potrebno je da je popraćena brojnim animacijama koje ostavljaju dojam stvarnih pokreta. Tako je i u ovoj igri napravljeno nekoliko animacija koje su prije svega vezane za likove te neke za okolinu poput lave koja teče. Klasična definicija animacije bi bilo da su animacije slike u pokretu koje se izmjenjuje određenom brzinom kako bi dobili smisleni izgled pokreta ili neke radnje. Sve slike koje su korištene za izradu animacije su izrađene u Piskel-u.

Animator je dio paketa koji se nalazi u sklopu Unity-a koji omogućuje kontrolu animacije. U animatoru možemo dodavati više slojeva i više parametara koji su potrebni kako bi definirali pokretanje određene animacije. Prije korištenja animatora moramo napraviti neke animacije, a to se može u podprozoru *Animation* gdje se mogu kreirati nove animacije. Kako smo već prije spomenuli da je animacija slika u pokretu tako se i ovdje u navedenom podprozoru trebaju dodati slike koje se mogu poredati željenim redoslijedom i postaviti brzina izmjenjivanja slika. Nakon što smo postavili slike u slijedu kojem smo željeli i njihovu brzinu izmjene još trebamo spremiti tu animaciju pod nekim imenom. U nastavku možemo otvoriti podprozor animatora pri čemu ćemo vidjeti dijagram stanja koji se koristi za vizualiziranje slijeda animacija koje smo u prijašnjem koraku napravili i koje želimo povezati. Neovisno o animacijama koje smo izradili uvijek će postojati animacija tipa ulaz (*Entry)* i bilo koje stanje (*Any State).* Za tip bilo koje stanje je specifično da se niti iz jedne animacije ne može doći u to stanje, ali se zato iz tog stanja može doći, povezati putem tranzicije u bilo koje drugo stanje. Tranzicije predstavljaju putanju i smjer prijelaza animacije iz jedne u drugu. Kako bi Animator znao kada treba preći iz jedne animacije u drugu potrebno je da se definiraju parametri koji predstavljaju okidač kada se koja animacija treba izvršavati. Parametri koji se mogu definirati su tipa: *float, int, bool* i *trigger.* Tako recimo za animaciju hodanja kao parametar možemo postaviti tipa *float* pod nazivom „hodanje“ koja će se aktivirati svaki puta igrač se kreće jer će svaka kretnja igrača imati neku decimalnu vrijednost koja je različita od nule i tada će biti pokrenuta animacija hodanja. S druge strane ako je vrijednost parametra „hodanje“ nula tada možemo postaviti da se koristi animacija kada igrač stoji. U nastavku možemo vidjeti primjer animatora koji je definiran za glavnog lika Maximusa (Sl.3.20.).



Sl. 3.20. Animator glavnog lika Maximusa

Vidimo kako je već na stranici prije objašnjeno da se animator sastoji od dijagrama stanja koja su povezana tranzicijama (bijelim linijama na kojima su strjelice) možemo vidjeti da su neka stanja naglašena bojom. Plava i zelena boja naglašava stanja koja su predefinirana i koja se ne mogu brisati niti mijenjati dok narančasta boja predstavlja stanje koje smo postavili kao glavno, početno. Na lijevoj strani slike možemo vidjeti definirane parametre od kojih je jedno tipa *float* (*Speed*) i tri tipa *bool* (*Grounded*, *SwordAttack* i *SpikeAttack*). Prvi parametar može imati vrijednost 0 ili 1 i koristi se za animaciju stajanja Maximusa na mjestu. Kao drugi parametar se koristi za animaciju Walking koja predstavlja hodanje i sadrži decimalne vrijednost. Treći i četvrti parametar mogu imati vrijednosti kao i prvi i koriste se za pokretanje animacije napada mačem i metalnom zvijezdom. Jasno možemo vidjeti da su stanja međusobno povezana tranzicijama pri čemu smo kod svake tranzicije postavili uvjet (parametar) koji će aktivirati prelazak iz jedne animacije u drugu. Tako je za prijelaz iz animacije mirovanja u animaciju hodanja na tranziciju postavljen uvjet ako je vrijednost parametra *Speed* veća od 0.1 da se pokrene animacija hodanja, a ako je manja od 0.1 da se vrati ili ostane u stanju mirovanja. Tako su redom na svakoj tranziciji definirani odgovarajući parametri koji predstavljaju okidače pojedine animacije. Kako bi se te animacije povezale s objektom potrebno ih je osim definiranja njihovih parametara u Animatoru povezati i sa skriptama koje će dati povratnu vrijednost kada se pojedina animacija uistinu može izvršiti.

```
public class Player : MonoBehaviour {

    private Animator anim;

    private bool swordAttack;

    private bool spikeAttack;


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

    void FixedUpdate()
    {

        anim.SetBool("Grounded", grounded);


        if (Input.GetKeyDown(KeyCode.Mouse0) &&
              Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) == 0 && !paused && Time.time >
              fireRate + lastShot)

        {
            spikeAttack = true;
            lastShot = Time.time;
        }


        if (!Input.GetKeyDown(KeyCode.Mouse0) )
        {
            spikeAttack = false;
        }


        anim.SetBool("SpikeAttack", spikeAttack);

        anim.SetFloat("Speed", Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x));


        if (Input.GetKeyDown(KeyCode.Mouse1) &&
              Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) == 0 )
        {
            swordAttack = true;
        }


        if ((!Input.GetKeyDown(KeyCode.Mouse1) ||
              Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) != 0))
        {
            swordAttack = false;

        }

        anim.SetBool("SwordAttack", swordAttack);

       }
   }
```

Sl. 3.22. Dio skripte Player u kojoj se koriste animacije

Za korištenje i pokretanje animacija iz skripte potrebno je definirati pojedine varijable. Za kontrolu animacije koristi se klasa Animator koja sadrži ugrađene javne funkcije. Tako je i ovdje definirana spomenuta klasa pod varijablom anim kojom će se dohvaćati potrebne funkcije. Također su definirane dvije varijable tipa *bool,* jedna za napad mačem (*swordAttack*) dok je druga za napad metalnim zvijezdama (*spikeAttack*). Unutar funkcije Start() dohvaćamo komponentu Animator koja je dodana na Maximusa i time možemo koristi prije napravljene animacije i definirane parametre. Sve parametre iz Animatora se navode u dvostrukim navodnicima pri čemu naziv parametara mora biti točan. U funkciji *FixedUpdate()* se nalaze naredbe koje provjeravaju događaje i aktiviraju animacije. Prva animacija se izvršava ukoliko je igrač na platformi pri čemu se koristi naredba *anim.SetBool* kojoj je potrebno predati dva argumenta (prvi je parametar „*Grounded*“ definiran u Animatoru, a drugi varijabla *grounded*). Kako ne bi ponovo bilo objašnjavano korištenje petlji koje slijede možemo se vratiti i pogledati prijašnji kod i objašnjenje (se nalazi na slici 3.6.). Ako je vrijednost varijable *spikeAttack* postavljena na istinu onda će se izvršiti animacija napada metalnim zvijezdama pomoću naredbe *anim.SetBool*. Zadnja animacija koja se nalazi u skripti je *anim.SetFloat* koja se izvršava ako je vrijednost kretanja igrača pozitivna i različita od nule. Kako bi izbjegli da se animacija ne izvršava kada se igrač kreće u lijevu stranu pri čemu je vrijednost brzine kretanja negativna koristi se naredba *Math.Abs* koja vraća apsolutnu vrijednost brzine kretanja, točnije uvijek je pozitivnog iznosa. Ovom skriptom smo obuhvatili sve animacije koje se koriste za igrača. Isti princip rada se koristi i za animacije neprijatelja prilikom hodanja i napadanja.

3.8. Grafičko korisničko sučelje (GUI)

GUI je kratica na engleskom za grafičko korisničko sučelje koje predstavlja način interakcije čovjeka računalom putem vizualnih prozora, ikona i izbornika kojima se može manipulirati pomoći miša i dijelom putem tipaka na tipkovnici. U konačnici takav način sučelja olakšava korisniku da ne mora upisivati naredbe koje bi inače bile potrebne kako bi se izvršio neki događaj koji želimo poput otvaranja nekog programa. Svaki klik mišem predstavlja izvođenje neke naredbe u pozadini samoga sučelja. Tako i Unity ima svoje korisničko sučelje koje pruža brojne mogućnosti i opcije.

3.8.1. Glavni izbornik

Početnu scenu koja je vidljiva prilikom pokretanja igre je glavni izbornik. Dijelovi od koji ga čine su: pozadinska slika, naziv igre i gumbovi. Kako bi lakše predočili kako to izgleda možemo pogledati sliku ispod.

Sl. 3.23. Prikaz glavnog izbornika

Vidimo da glavni izbornik se sastoji od svih dijelova koji su bili nedavno navedeni, a sada ćemo objasniti ulogu gumbova. Prilikom pritiska na svaki gumb (*Button*) izvršava se određeni događaj tako prilikom što pritisnemo gumb New Game otvara se nova scena Story koja sadrži priču same igre. Ispod se nalazi Select Level koji služi da možemo odabrati nivo koji želimo igrati. Ukoliko stisnemo gumb Instructions otvara se novi prozor, platno (*canvas*) u kojem se nalazi tekst s instrukcijama kako bi znali koje tipke se koriste za kontroliranje igrača. Predzadnji gumb je

Audio Settings koji otvara novo platno u kojem se nalazi mogućnost podešavanja glasnoće zvuka igre. Na samome kraju glavnog izbornika se nalazi gumb Exit koji služi kako bi izašli iz igre. Na stranici ispod možemo vidjeti skriptu Menu koja je dodana na platno glavnog izbornika.

```
public class menu : MonoBehaviour {

    public string startLevel;

    public GameObject instructionCanvas;

    public GameObject levelSelectCanvas;
    public string level_1;
    public string level_2;

    public string mainMenu;
    public Text title;

    public GameObject audioCanvas;

    public int MaximusLives;

    private AudioSource audio;
    public Slider slider;

    void Start()
    {
        instructionCanvas.SetActive(false);
        levelSelectCanvas.SetActive(false);
        audioCanvas.SetActive(false);
        audio = GetComponent<AudioSource>();
    }

    public void NewGame()
    {
        PlayerPrefs.SetInt("PlayerCurrentLives", MaximusLives);
        PlayerPrefs.SetInt("CurrentCoins", 0);
        PlayerPrefs.SetInt("CurrentEnemyDeath", 0);
        SceneManager.LoadScene(startLevel);


    }

    public void Instructions()
    {

        instructionCanvas.SetActive(true);
        title.GetComponent<Text>().enabled = false;

    }

    public void AudioSettings()
    {
        audioCanvas.SetActive(true);
        title.GetComponent<Text>().enabled = false;
    }

    public void ExitGame()
    {
        Application.Quit();
    }
```

```
    public void SelectLevel()
    {
        PlayerPrefs.SetInt("PlayerCurrentLives", MaximusLives);
        PlayerPrefs.SetInt("CurrentCoins", 0);
        PlayerPrefs.SetInt("CurrentEnemyDeath", 0);
        levelSelectCanvas.SetActive(true);
        title.GetComponent<Text>().enabled = false;
    }



    public void Level1()
    {
        SceneManager.LoadScene(level_1);

    }

    public void Level2()
    {
        SceneManager.LoadScene(level_2);

    }
    public void MainMenu()
    {
        instructionCanvas.SetActive(false);
        levelSelectCanvas.SetActive(false);
        audioCanvas.SetActive(false);
        title.GetComponent<Text>().enabled = true;

    }

    public void Volume()
    {
        audio.volume = slider.value;
    }

}
```

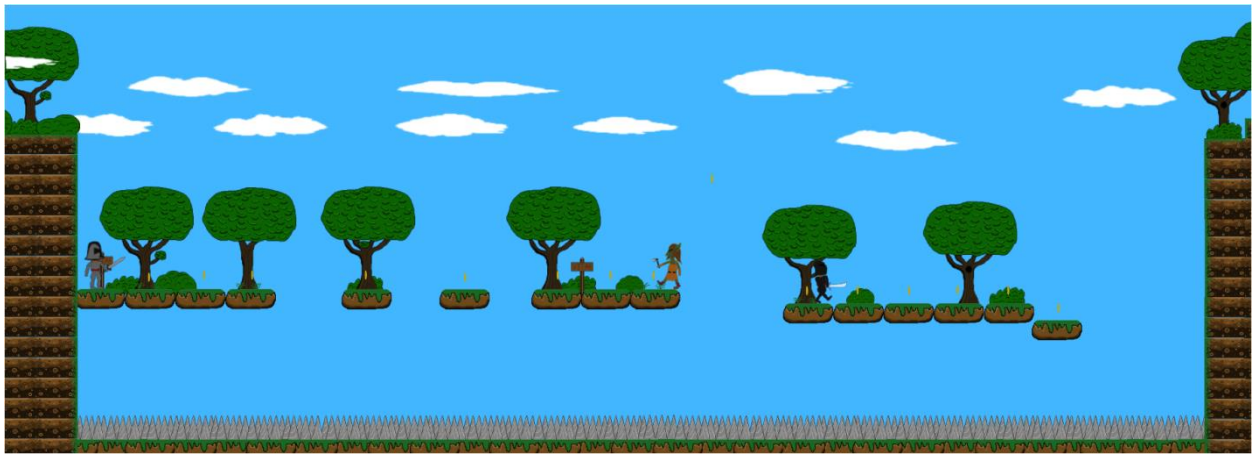
Sl. 3.24. Skripta menu

U skripti su definirane varijable koje smo već ranije mogli vidjeti, ali se pojavljuju i varijable tipa *string* (niz znakova) koje se koriste kako bi odredili koju scenu želimo pokrenuti i potrebno je upisati naziv scene u Inspektoru. Također su određeni i objekti koji će biti vidljivi ili nevidljivi ovisno o gumbu koji pritisnemo. U *Start()* funkciji smo postavili da se *canvas* za instrukcije, odabir nivoa i zvuka ne vidi jer ne želimo da to sve vidimo odjednom, nego tek prilikom odgovarajućeg gumba. Zadnjom naredbom unutar funkcije je definirano dohvaćanje komponente zvuka koja će se kasnije koristiti u drugoj funkciji. Prva javna funkcija je *NewGame()* u kojoj je postavljeno broj života igrača, zlatnika i ubijenih neprijatelja na vrijednost nula i naredba koja zahtjeva argument od niza znakova pri čemu je upisana varijabla koja predstavlja drugu scenu. Druga funkcija *Instructions()* je kratka i sadrži naredbu koja prikazuju instrukcije, točnije objekt koje je definiran kao *instructionCanvas* postaje vidljiv u sceni. Sljedeća funkcija je *AudioSettings()* u kojoj se uključuje *canvas* za zvuk. Zatim slijedi *ExitGame()* koja sadrži

naredbu za izlazak iz igre, aplikacije. Nakon toga slijede funkcije za odabiranje nivoa i vraćanje u glavni izbornik te funkcija *Volume()* koja mijenja vrijednost zvuka u skladu s našim pomicanjem klizača.

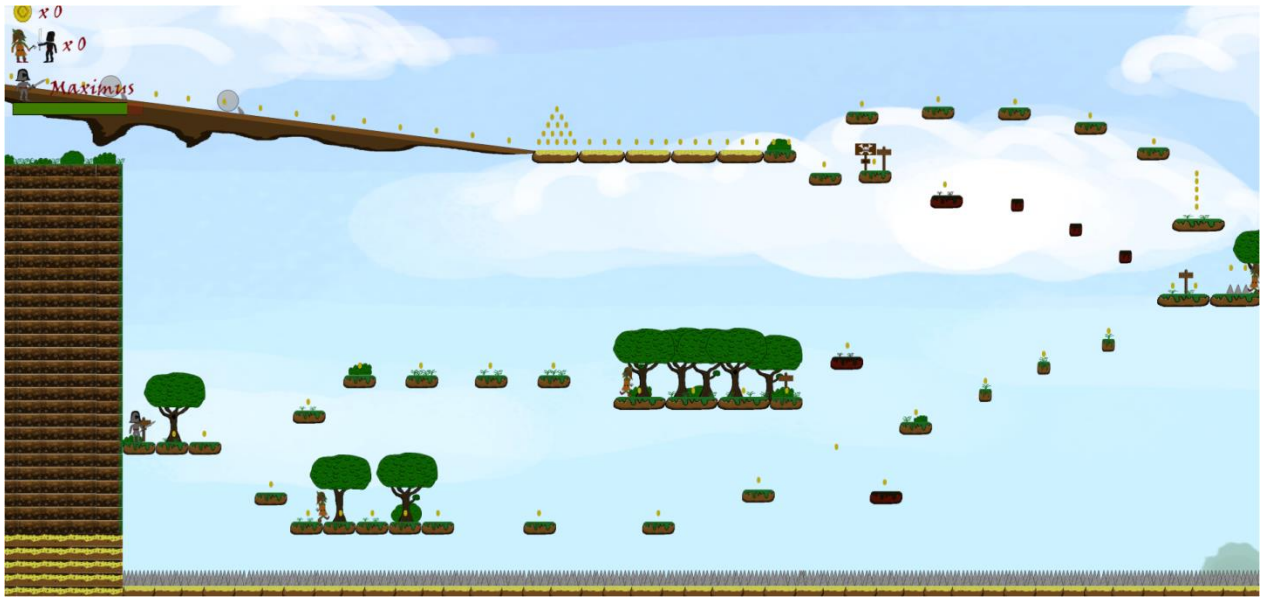
3.8.2. Nivoi

Svaki nivo predstavlja drugu scenu koja je izgledom drugačija i sadrži drugačije prepreke i neprijatelje. Ukupno ima tri nivoa koje igrač treba proći kako bi uspješno završio igru. Svaki nivo se sastoji od platformi po kojima se igrač kreće, neprijatelja, zamki, temeljnih platformi na koje ako igrač skoči umire, pozadinske slike i raznih *sprite-va* koji služe kako bi scena bila zanimljivija. Kako bi se igrač mogao kretati po platformama potrebno je da im se doda komponenta *collider*[2] koja stvara koliziju između igrača i same platforme i time tvori čvrsti oslonac. Također je ista komponenta dodana na razne zamke od šiljaka do kotrljajućeg kamenja. Kraj prvog i drugog nivoa predstavlja zadnja platforma koja ima na sebi dodanu skriptu za učitavanje iduće scene, slično kao što sadrže gumbovi u glavnom izborniku. Na slikama možemo vidjeti kako izgleda pojedini nivo.

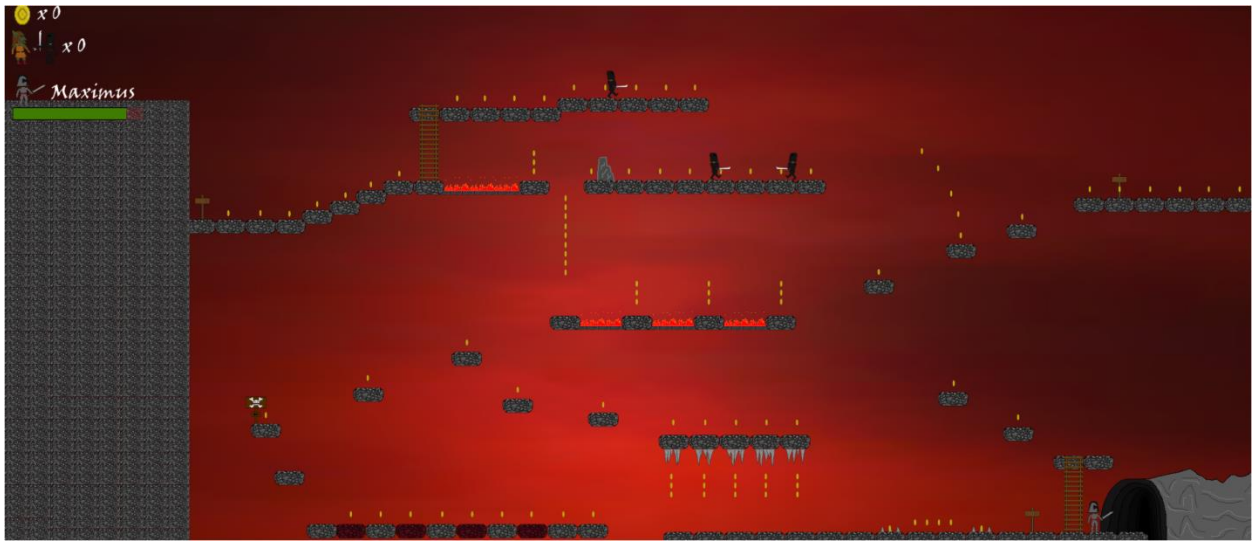


Sl. 3.25. Prikaz prvog nivoa

[2] *Collider* – predstavlja komponentu koja definira oblik objekta u svrhu ostvarivanja kolizije s drugim objektima



Sl. 3.26. Prikaz prvog djela drugog nivoa



Sl. 3.27. Prikaz prvog djela trećega nivoa

Možemo uočiti da svaki nivo ima drugačiji izgled, ali da je rađen na istom principu korištenjem platformi. Vidimo da su unaprijed postavljeni položaji i mjesta na kojima se nalaze neprijatelji.

3.8.3. Sučelje unutar nivoa

Svaka scena koja predstavlja nivo ima podatke koji prikazuju gledajući od gore prema dolje sliku i broj skupljenih zlatnika, sliku neprijatelja i njihov broj i na kraju slika Maximusa kraj koje je ime te ispod vrijednost njegova života. Na slici ispod možemo vidjeti navedene podatke koji se prenose iz nivoa u nivo.

Sl. 3.28. Prikaz podataka vidljivih u nivoima

Svaki nivo sadrži *canvas* koji nije odmah vidljiv i sadrži tekst da je kraj igre (Game Over), konačni rezultat kojeg je igrač ostvario, broj skupljenih zlatnika, poraženih neprijatelja i gumb pa povratak u glavni izbornik. Pozadina mu je crne boje kako bi se tekst bolje vidio i sami podaci koji se prikazuju. Pojavljuje se kada igrač izgubi sve živote. Na idućoj stranici možemo vidjeti skriptu koja je dodana kao komponenta da bi potrebni podaci bili prikazani.

Sl. 3.29. Prikaz kraja igre

```
public class GameOverStats : MonoBehaviour {

    public Text Score;
    public static int coinCounter;
    private static int counter;

    public Text EnemyKilled;
    public Text CoinsCollected;
    private static int coinsNumber;

       void Start () {
        coinCounter = PlayerPrefs.GetInt("CurrentCoins");
        counter = PlayerPrefs.GetInt("CurrentEnemyDeath");
        coinsNumber = PlayerPrefs.GetInt("CurrentCoins");
        ScoreDisplay();
        Cursor.visible = true;
       }

    void ScoreDisplay()
    {
        coinCounter = (coinCounter + counter)*8;
        Score.text = "Your Score: " + coinCounter.ToString();
        EnemyKilled.text = "Enemy Killed: " + counter.ToString() + "/16";
        CoinsCollected.text = "Coins Collected: " + coinsNumber.ToString() + "/300";
    }

    void FixedUpdate()
    {
        counter = PlayerPrefs.GetInt("CurrentEnemyDeath");
    }
}
```

Sl. 3.30. Skripta GameOverStats

Ova skripta GameOverStats je poprilično jednostavna jer zadrži kod koji omogućuje ispisivanje teksta. Tri glavne varijable koje su definirane, a tipa su tekst su: *Score*, *EnemyKilled* i *CoinsCollected* i još postoje varijable cjelobrojnog tipa koje služe kao njihov brojač. U funkciji *Start()* je vrijednost *coinCounter*, *counter* i *coinsNumber* postavljena na trenutnu vrijednost koja je u igri do sada skupljena, a odnosi se na broj zlatnika i poraženih neprijatelja pri čemu se koristi klasa *PlayerPrefs* i statička funkcija *GetInt* koja omogućuje dohvaćanje vrijednosti spremljenih u odgovarajući niz znakova (npr. „*CurrentCoins*“). U nastavku se poziva još funkcija *ScoreDisplay()* koja prikazuje vrijednosti varijabla i naredba *Cursor.visible = true* koja čini pokazivač miša vidljivim. Unutar funkcije *ScoreDisplay()* postavljaju se vrijednosti koje će biti ispisane prilikom kraja igre. Bitno je uočiti kako se za ispis teksta u koje su uključene i varijable cjelobrojnoga tipa koristi njihov naziv i javna funkcija *.ToString()* koja pretvara vrijednost u tekst. Posljednja je funkcija u skripti je *FixedUpdate()* unutar koje se cijelo vrijeme izvršava naredba za dohvaćanje vrijednosti ubijenih neprijatelja i spremanja u varijablu *counter*.

Unutar svakoga nivoa se nalazi još jedan *canvas* koji predstavlja pauzu u igri. Kako bi se uopće pokrenuo potrebno je pritisnuti tipku „P“ kao kratica za pauzu (eng. *Pause)*. Nakon toga vidimo da je igra zaustavljena i prikazan je „izbornik“ koji se sastoji od teksta i gumbova. Također je dodana skripta kao komponenta na taj *canvas* koja će biti prikazana i ukratko objašnjena.

Sl. 3.31. Prikaz izbornika za pauzu

Lako možemo uočiti da se sastoji od četiri gumba: Resume, New Game, Audio i Main menu. Prvi gumb omogućuje nastavak igre, drugi ponovo pokretanje igre, predzadnji postavke glasnoće zvuka i zadnji povratak u glavni izbornik. Sama skripta Pause koja je dodana upravlja svim navedenim gumbima u smislu njihovih funkcija. Ukupno ima sedam funkcija od kojih se pet odnosi na gumbove. *Start()* sadrži naredbu koja *canvas* koji se odnosi za namještanje zvuka čini nevidljivim dok *Update()* funkcija sadrži provjeru je li pritisnuta tipka P ili nije i ukoliko je aktivira se prozor za pauzu sa svim svojim opcijama. Prva funkcija koja se odnosi na gumb je *Resume()* i postavlja vrijednost varijable *paused* i pokazivač miša na laž kako ne bi bio vidljiv u igri i time se nastavlja igra nivoa. *NewGame()* je druga funkcija koja postavlja živote Maximusa na vrijednost tri te broj zlatnika i ubijenih neprijatelja na nulu i učitava ponovo prvi nivo igre. Treća funkcija *MainMenu()* ima jednu naredbu i to koja učitava scenu glavnog izbornika. Funkcija *AudioSettings()* otvara novi *canvas* te zaustavlja vrijeme i aktivira vidljivost pokazivača miša. Zadnja funkcija *Back()* gasi opcije zvuka i vidljivost pokazivača miša.

```
public class Pause : MonoBehaviour {

    public string newLevel;
    public string mainMenu;
    public bool paused;
    public GameObject pauseMenuCanvas;
    public GameObject audioCanvas;
    private int MaximusLives = 3;

       void Start () {
        audioCanvas.SetActive(false);
       }
       void Update () {
        if (paused)
        {
            pauseMenuCanvas.SetActive(true);
            Time.timeScale = 0f;
            Cursor.visible = true;
        }
        else
        {
            pauseMenuCanvas.SetActive(false);
            Time.timeScale = 1f;
        }

        if (Input.GetKeyDown(KeyCode.P))
            paused = !paused;
       }
    public void Resume()
    {
        paused = false;
        Cursor.visible = false;
    }
    public void NewGame()
    {
        PlayerPrefs.SetInt("PlayerCurrentLives", MaximusLives);
        PlayerPrefs.SetInt("CurrentCoins",0);
        PlayerPrefs.SetInt("CurrentEnemyDeath",0);
        SceneManager.LoadScene(newLevel);
    }
    public void MainMenu()
    {
        SceneManager.LoadScene(mainMenu);
    }

    public void AudioSettings()
    {
        paused = false;
        audioCanvas.SetActive(true);
        Time.timeScale = 0f;
        Cursor.visible = true;
    }
    public void Back()
    {
        audioCanvas.SetActive(false);
        Cursor.visible = false;
    }
}
```

Sl. 3.32. Skripta Pauze

4. ZAKLJUČAK

U ovom završnom radu napravljena je računalna igra tipa 2D platformer pri čemu je korišten Unity Engine u sklopu programskog jezika C#. Ukratko je objašnjeno što je Unity i od čega se sastoji. Obuhvaćen je cjelokupni proces od priče same igre preko svih osnovnih dijelova koji su potrebni za izradu jedne 2D igre pa sve do skripti i kodova koji čine ovu igru. Pri čemu se za izradu likova i animacija koristio program Piskel, a za realizaciju samih animacija animator koji je sastavni dio samog Unity-a. Kako bi se mogla izraditi jedna takva igra potrebno je posjedovati osnovno znanje za programiranje i korištenje programskog jezika i razumijevanja koncepta kako se objekti mogu kontrolirati pomoću skripti. Potrebno je i poznavati osnove fizike kretanja i odnosa među objektima koja je nužna kao element i daje smislenost igri kao takva. Moguće su još brojne dorade na igri poput korištenja pokretnih platformi, dodavanja nivoa te što se tiče samih izbornika opcija za odabir različitih jezika i razne druge mogućnosti.

LITERATURA

[1] Bart Kelsey, OpenGameArt , 28.03.2009. – 01.06.2017.
https://opengameart.org/

[2] Bfxr Standalone v1.4.1 – 29.05.2017.
http://www.bfxr.net/

[3] Mp3linew – 05.06.2017.
http://mp3linew.com/fr/ex.php?q=Medieval+Music

[4] Unity Technologies, 08.06.2005. – 03.06.2017.
https://en.wikipedia.org/wiki/Unity_(game_engine)

[5] Unity, Multiplatform, 01.10.2005. – 31.05.2017.
https://unity3d.com/unity/multiplatform

[6] Piskel, 23.12.2013. – 05.06.2017.
http://www.piskelapp.com/

SAŽETAK

U ovom radu je obrađena 2D platformer računalna igra izrađena u Unity Engine-u. Cilj igre je pobijediti glavnog protivnika Meranoxa pri čemu mu pomažu njegovi podanici (ljudi crnih oklopa) i goblini te time osloboditi svoju rasu. Glavni lik kojim igrač upravlja je Maximus koji se može kretati gore-dolje, lijevo-desno i skakati, a kao oružje bacati metalne zvijezde te udarati mačem. S druge strane protivnici se također kreću, ali ne mogu skakati. Goblini koriste sjekire koje mogu bacati, a ljudi crnih oklopa mačeve s kojima mogu udarati. Kako je igra izrađena u Unity-u koriste se skripte za upravljanje objektima koje su pisane C# programskim jezikom. Što se tiče grafičkog dijela igre, sva vizualizacija je izrađena u Piskel web aplikaciji osim pozadina koje su preuzete sa stranica slobodnih sadržaja. Za izradu animacije korišten je animator koji je sastavni dio Unity-a. Računalna igra ukupno ima šest scena. Prve tri se odnose na glavni izbornik, priču i odabira težine igranja dok se ostale tri scene odnose na nivoe koje igrač treba proći kako bi završio igru.

Ključne riječi: 2D platformer računalna igra, Unity Engine, Meranox, Maximus, skripta, C# programski jezik, Piskel, animator, scena, nivo

DEVELOPMENT OF 2D PLATFORMER VIDEO GAME

ABSTRACT

In this final paper was created 2D platformer video game which was made in Unity Engine. The goal of this game is to defeat main enemy Meranox and his vassals (people with black armor) and goblins in order to free his race. The main character, who is controlled by player, is Maximus who can move up-down, left-right and jump, and can throw shuriken as weapon and can also attack with his sword. On the other hand, enemies can also move but can not jump. Goblins use axes which they can throw while people with black armor use their swords to attack. As the game was made in Unity, there are scripts which are used to control objects in programming language C#. Regarding the graphic part of the game, all visualization was made in Piskel web application except backgrounds which are downloaded from free-content sites. For creating animation, animator is used which is component of Unity. Video game has in total six scenes. The first three applies on the main menu, story and selection of difficulty while the other three scenes apply on levels which player needs to go through in order to finish the game.

Key words: 2D platformer video game, Unity Engine, Meranox, Maximus, script, C# program lanuguage, Piskel, animator, scene, level

ŽIVOTOPIS

Ivan Drulak rođen je 26.09.1995. godine u Virovitici. Od 2002. do 2006. pohađao je OŠ Ivana Gorana Kovačića, a od 2006. do 2010. Vladimira Nazora u Virovitici. Nakon završene osnovne škole upisao je srednju školu, gimnaziju Petra Preradovića u Virovitici koju je završio 2014. Godine. Također je te iste godine upisao redovni preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku koji još uvijek pohađa. Još kao dječak je bio ljubitelj računalnih igara te ljubitelj sporta i fizičkih aktivnosti poput vožnje biciklom.

PRILOZI

Slika 1. Maximus stand still

Slika 2. Maximus walking

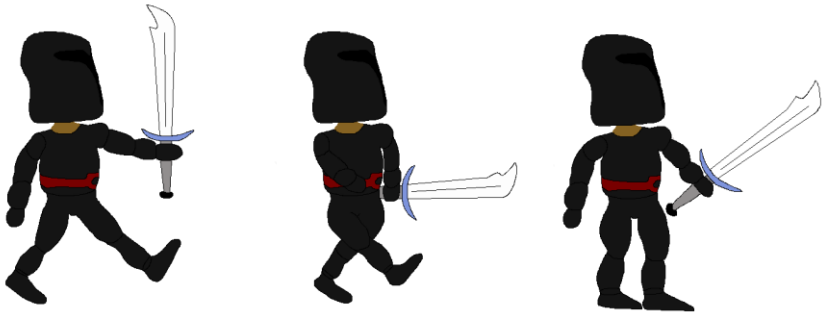
Slika 3. Maximus jump

Slika 4. Maximus sword attack

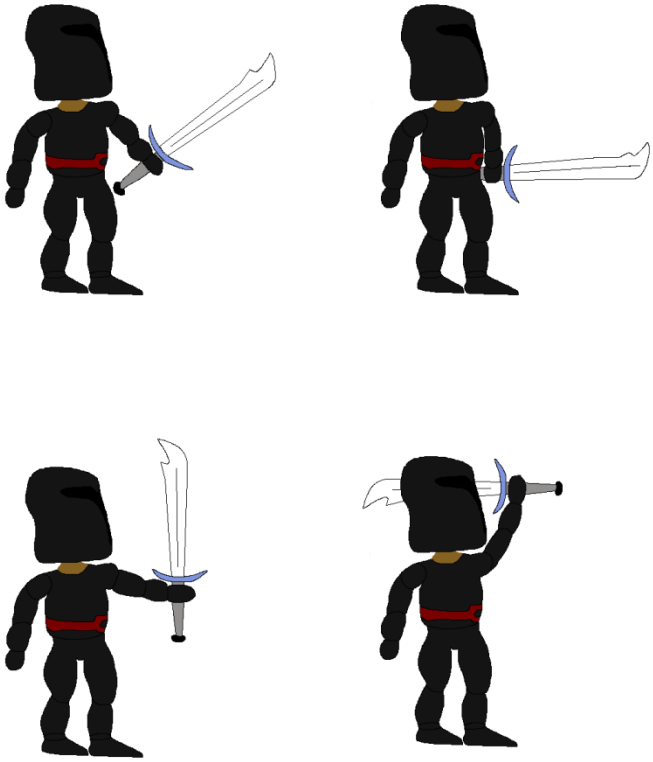
Slika 5. Maximus shuriken attack

Slika 6. Goblin walking

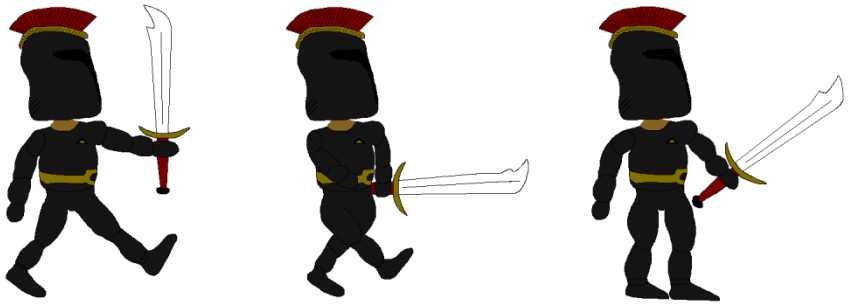
Slika 7. Golbin axe attack



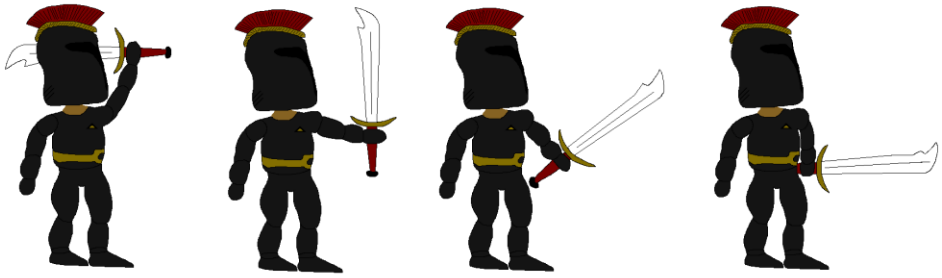
Slika 8. Black enemy walking



Slika 9. Black enemy sword attack

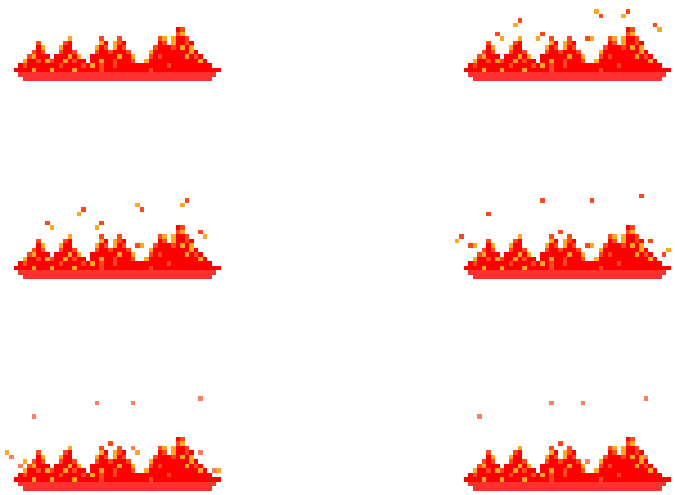


Slika 10. Meranox walking

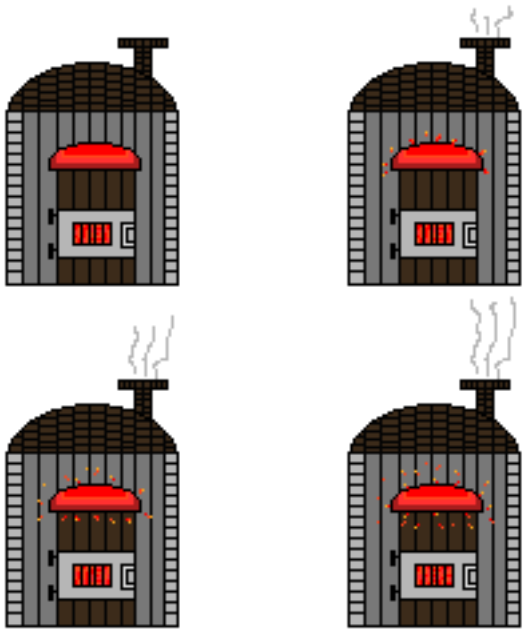


Slika 11. Meranox sword attack

Slika 12. Bat fly



Slika 13. Lava



Slika 14. Furnace