Математичка гимназија

МАТУРСКИ РАД

- из рачунарства и информатике -

Прављење игре у Unity-у

Ученик: Лука Менгер IVб

Ментор: Јелена Хаџи-Пурић

Београд, јун 2024.

Садржај

1. Увод 3
2. Историја развоја рачунара 4
2.1 Постојећи радови на сличну тему 4
2.2 Прозор у Unity окружењу 5
3. Unity окружење 6
4. Развој компјутерске игре у Unity-у 7
4.1 Графика 7
4.2 Чаробњак – кретња и понашање 8
4.3 Платформе – кретња и стварање 10
4.4 Браве – кретња, стварање и откључавање 11
4.5 Контрола брзине 12
4.6 Логички менаџер 13
4.7 Крај игре 15
5. Звукови, анимације и стартни екран 16
5.1 Звукови 16
5.2 Анимације 16
5.3 Стартни екран 17
6. Закључак 18
Литература 19

1

Увод

Unity engine (покретач) је популарни пакет алата за креирање игара који је развила компанија Unity Technologies. Првобитно је лансиран као Mac *OS X engine* за игре у јуну 2005. Програмирање самих игара у Unity-у се врши у C# програмском језику што омогућава једноставно прављење игара за скоро сваку платформу.

Одувек сам био заинтересован и волео сам да играм рачунарске игре, а из те заинтересованости је настала и жеља да направим своју игру. Игра коју сам представио у овом раду сам назвао *Чаробњачки изазов* и она је тзв. *Endless runner* тип игре што значи да се игра базира на кретању главног лика (чаробњака) кроз бесконачни ниво све док играч не изгуби.

Поред Unity-а за овај пројекат су коришћени *Adobe Illustrator* и *Adobe Photoshop* за израду 2D модела и *Visual Studio* за куцање кода.

Контроле у игри се састоје од кретања лево (тастер A), кретања десно (тастер D), скока (тастер W или тастер SPACE), убрзаног спуштања карактера (тастер S) и коришћење једне од две чини за повишен скок и откључавање браве (тастери 1 и 2 или леви и десни клик миша редом). Главна тематика игре је кретање чаробњака од платформе до платформе које се померају. Игра се завршава када чаробњак нестане са екрана (оде превише лево, превише десно или превише доле) или падне на шиљке који се налазе на појединим платформама. У игри се такође памти најбољи резултат играча. Изазови у току прављења саме игре, као и начин на који су решени, ће бити касније наведени.

2

Историја развоја рачунарских игара

2.1 Постојећи радови на сличну тему

Рачунарске игре постоје већ деценијама и сматра се да је прву игру направио Американац Стив Расел (*Spacewar!,* 1962. године) али његова игра није доживела велики успех. Прва успешна игра (чак и данас игра је врло популарна и позната) носи назив *Pong* и направио ју је Американац Нолан Бушнел. Након ове игре почео је развој игара у виду жанрова, детаља, анимација и компатибилности са платформама. У даљем тексту се наводе књиге које говоре о развоју самих игара.

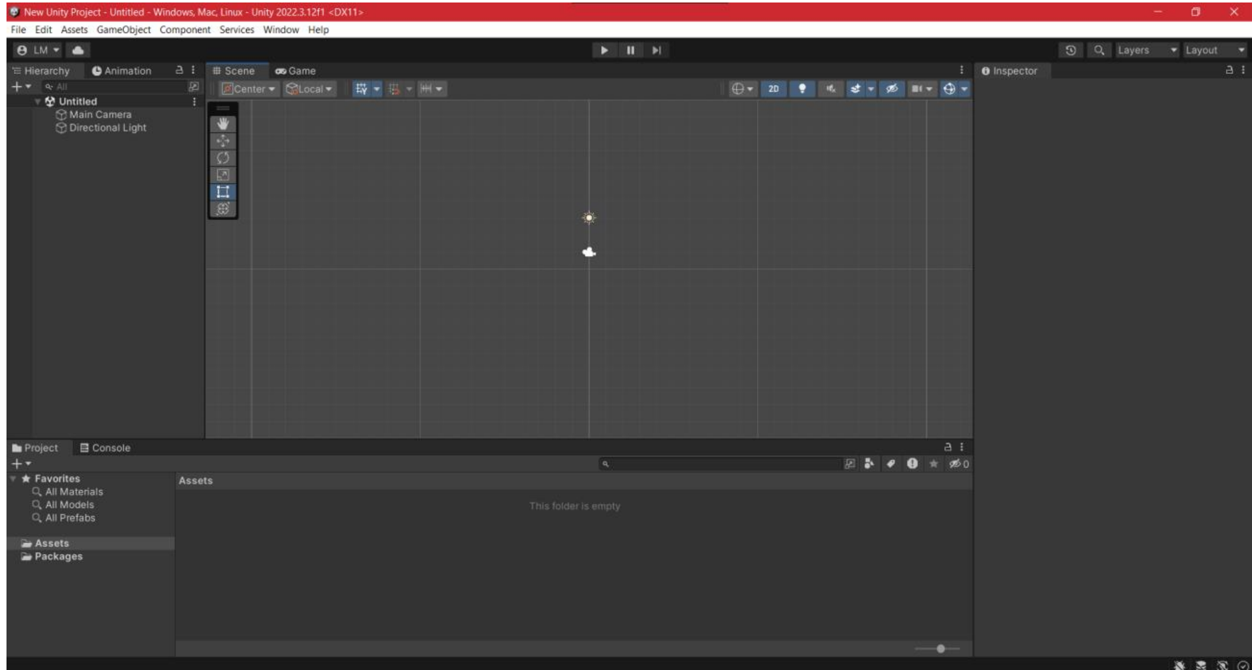
Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture је књига коју је написао Дејвид Кушнер и књига прати причу оснивача компаније *id Software* и стварању игара као што су *Doom* и *Quake* које су поставиле темеље модерних пуцачина из првог лица (*FPS* игре).

Replay: The History of Video Games је књига коју је написао Тристан Донован и она пружа преглед историје видео игара, почевши од почетка аркадних игара па све до модерног доба, са фокусом на кључне догађаје, игре и људе који су обликовали индустрију.

Extra lives: Why Video Games Matter је књига коју је написао Том Бисел и она даје занимљив преглед историје кроз ауторове личне приче и искуства.

2.2 Прозор у Unity окружењу

При отварању новог пројекта у Unity-у отвориће се прозор налик прозору на слици испод:



Можемо приметити наредних неколико значајних делова прозора:

На самој средини се налази *Scene* картица. У овом делу се постављају ликови, околина и остали елементи игре. Могућа је ротација објеката, њихово скалирање и одређивање њиховог положаја али без неке одређене прецизности.

На дну прозора се налази *Project* картица која приказује све фајлове који су повезани са пројектом и представља главни метод проналажења и навигације средстава (*Assets*) и фајлова у пројекту.

Са леве стране се налази *Hierarchy* картица која садржи све објекте у тренутно приказаној сцени (нпр. Камера, главни лик итд.).

Са десне стране се налази *Inspector* картица која омогућава прецизно (за разлику од *Scene* картице) мењање особина објекта као што су: ротација, величина, положај на сцени итд.

3

Unity окружење

Unity функционише и базира своје пројекте на тзв. структурама *Game objects* (објекти игре), којима се додељују различите компоненте које ће учинити да се објекат понаша онако како је то предвиђено. Једне од најважнијих компоненти које се користе су:

Sprite renderer који омогоћава да објекат користи жељену графику (цртеж, фотографија, слика направљена помоћу програма итд.).

Transform који садржи све податке о објекту на сцени и омогућава њихово мењање (величина, позиција, ротација итд.).

Rigid body објекат претвара у физички објекат са различитим физичким величинама (гравитација, врзина итд.).

Collider омогућава објекту да интерагује и не пролази кроз друге објекте са истом овом компонентом.

Script је компонента у којој се пише код за само понашање објекта (објекат може имати више скрипти). Свака скрипта ће у себи већ имати део кода који је стандардан и он подразумева два заглавља под именима *Start* (почетак) и *Update* (освежавање) које покрећу свој код при почетку игре и у сваком тренутку у игри. У скрипти се такође дефинишу јавне (*Public*) променљиве које се после у Unity-у повезују са компонентама које ће се користити у коду (повезују се најчешће превлачењем). Те променљиве се такође не морају повезивати са компонентама, али зато можемо видети и мењати њихове вредности у Unity-у. На слици испод се може видети како то изгледа у Unity-у.

.

4

Развој компјутерске игре у Unity-у

4.1 Графика

Као што сам већ раније напоменуо графика је урађена у програму *Adobe Illustrator* и *Adobe Photoshop,* програми специјализовани за цртање, дизајнирање и уређивање слика.

Модели који су направљени и појављују се у игри су модели чаробњака, мале и велике платформе и велике платформе са шиљцима. Сваки од модела је направљен помоћу више различитих обојених геометријских тела које програм нуди. У прилогу се налазе модели чаробњака и мале платформе.

4.2 Чаробњак – кретња и понашање

Чаробњаково кретање је, као што сам већ навео, регулисано контролама на тастатури. Чаробњак поседује две *Collider* компоненте, једна за сударање са другим објектима и друга која је помоћна за регулацију скока. У *Update* заглављу скрипте се то проверава следећим кодом:

```
if ((Input.GetKeyDown(KeyCode.Space) || Input.GetKeyDown(KeyCode.W)) && !Air)
{
   WizardRigidBody.velocity = Vector2.up * JumpStrenght;
   Sound.PlaySfx(Sound.Jump);
   Animator.SetTrigger("InAir");
}
```

Овај део кода проверава да ли је унет један од тастера за скок. Променљива *Air* означава тренутно стање чаробњака (да ли је скочио или није) и онемогућава двоструки скок ако је чаробњак већ скочио. У случају да је скок могућ и унет је тастер за скок, чаробњаков вертикална брзина се мења преко променљиве *WizardRigidBody* (која је повезана са компонентом *RigidBody*) и постаје производ јачине скока (*JumpStrenght*) и дводимензионалног вектора који је предефинисан и има координате (0,1) тј. усмерен је ка горе и има јачину 1. Последње две линије кода су везане за анимације у игри и звукове и њих ћу детаљније објаснити касније.

```
private void OnTriggerEnter2D(Collider2D collision)
{
   Air = false;
}
private void OnCollisionExit2D(Collision2D collision)
{
   Air = true;
}
```

Ове две функције мењају вредност променљиве *Air*. Прва функција проверава да ли је компонента *Collider* (друга од раније поменутих), која је подешена да буде тзв. *Trigger* (окидач), дошла у контакт са другим објектима и ако јесте мења вредност променљивој. Друга функција прати тренутак када чаробњак неће бити у колизији ни са једним објектом и тада мења вредност променљиве. Прва функција прати другу компоненту *Collider* зато што би у супротном чаробњак могао да скочи и када додирне платоформу својом главом например (у случају падања).

```
if (Input.GetKey(KeyCode.A))
{
   gameObject.transform.eulerAngles = new Vector3(0, 180, 0);
   gameObject.transform.position += Vector3.left * Speed * Time.deltaTime;
   if (!Air)
      Animator.SetBool("IsRunning", true);
   else
```

```
        Animator.SetBool("IsRunning", false);
}
else
    Animator.SetBool("IsRunning", false);
if (Input.GetKey(KeyCode.D))
{
    gameObject.transform.eulerAngles = new Vector3(0, 0, 0);
    gameObject.transform.position += Vector3.right * Speed * Time.deltaTime;
    if(!Air)
        Animator.SetBool("IsRunning", true);
    else
        Animator.SetBool("IsRunning", false);
}
if (Input.GetKeyDown(KeyCode.S))
{
    WizardRigidBody.velocity = Vector2.down * DownStrenght;
}
```

Уз помоћ ових упита се редом проверава да ли су унети тастери за кретање лево, десно или убрзано спуштање. Код за спуштање функционише поприлично слично као и за скок па га нећу детаљно објаснити. Код за кретање лево проверава да ли је одговарајући тастер притиснут и ако се то деси, извршава се ротација објекта за 180 степени (зато што почиње да се креће на другу страну). Након тога се тренутна позиција повећава за производ тродимензионалног вектора који је предефинисан за кретање налево (координате (-1,0,0)), тренутне брзине чаробњака (која се временом мења и то ће касније бити покривено) и променљиве *Time.deltaTime* која се стара о томе да се извршавање игрице одвија у истом времену на сваком рачунару (без ње би се на бољим рачунарима игра много брже одвијала и не би било могуће играти је). Кретање надесно је изведено на исти начин као и налево. Поново се јављају делови кода везани за анимације па ће и то бити касније објашњено.

```
if ((Input.GetKey(KeyCode.Alpha1) || Input.GetMouseButton(0)) && Timer > Cooldown && Air)
{
    RechargeCircle.Used = true;
    WizardRigidBody.velocity = Vector2.up * MagicStrenght;
    Timer = 0;
    Sound.PlaySfx(Sound.Magic);
    Sound.PlaySfx(Sound.DoubleJump);
    Animator.SetTrigger("InDoubleJump");
}
else
    Timer += Time.deltaTime;
```

Прва чаролија се реализује помоћу кода изнад и она омогућава додатни скок. Чаролија се не може користити два пута у кратком временском периоду (мањем од 3 секунде) и за то се старају променљиве *Timer* (бројач времена) и *Cooldown* (минимално време између две чаролије). Када се унесе тастер за коришћење прве чаролије такође се проверава да ли је прошло довољно времена од прошлог коришћења исте и да ли је чаробњак скочио пре тога. У случају да није прошло довољно времена *Timer* се повећава за *Time.deltaTime* јер је то време после ког се поново покреће заглавље *Update*. Када прође

довољно времена, чаролија се изводи исто као и класичан скок и *Timer* се поставља на 0. Почетна вредност бројача је стављена на минимално време за извођење чаролије да би одмах по стартовању игре чаролија била спремна. Друга чаролија је изведена у другом објекту и биће објашњена касније. Овде се поред кода за анимације и звукове појављује и део кода везан за круг по коме се прати могућност коришћења чаролије и то ће исто бити касније наведено.

4.3 Платформе - кретња и стварање

Већ сам нагласио да постоје три различите врсте платформи, а кретање сваке од њих је реализовано једном истом скриптом.

```
gameObject.transform.position += Vector3.left * Speed * Time.deltaTime;
if (gameObject.transform.position.x < DeleteZone)
	Destroy(gameObject);
```

Код изнад се налази у *Update* заглављу. Први ред кода врши кретање платформи на исти начин као што је извршено и кретање чаробњака, само без икаквог услова. У следећој линији се врши провера X координате платформе, ако је она мања од променљиве *DeleteZone,* тј. платформа се налази ван екрана, објекат платформе се брише.

Стварање платформи је реализовано помоћу још једног објекта по имену *PlatformSpawner* који користи шеме за сваку од платформи и ствара их изнова. Шема за платформе се једноставно ствара превлачењем објекта из *Hierachy* прозора у *Assets* прозор (настаје тзв. *Prefab*). Након тога се у скрипти за стварање платформи дефинишу променљиве типа *GameObject.*

```
if (Timer >= SpawnRate)
{
	SpawnPlatforms();
	Timer = 0;
}
else
	Timer += Time.deltaTime;
```

Овај код се позива стално у игри и помоћу бројача времена ствара платформе након времена које је одређено *SpawnRate* променљивом. Када бројач пређе ту вредност позива се следећа функција:

```
void SpawnPlatforms()
{
	Type = Random.Range(0, 3);
	Lowest = transform.position.y - Offset;
	Highest = transform.position.y + Offset;
```

```
	if (Type == 0)
		Instantiate(Platform, new Vector3(transform.position.x, Random.Range(Lowest, Highest), 0), transform.rotation);
	else
		if (Type == 1)
			Instantiate(LongPlatform, new Vector3(transform.position.x, Random.Range(Lowest, Highest), 0), transform.rotation);
		else
			if (Type == 2)
				Instantiate(LongPlatformSpikes, new Vector3(transform.position.x, Random.Range(Lowest, Highest), 0), transform.rotation);
}
```

Функција у целобројну промељиву *Type* смешта насумичан број између 0 и 3 (укључујући 0, а не 3). Тиме се одређује врста платформе која ће се створити. Променљиве *Lowest* и *Highest* одређују интервал у ком је могуће да се нова платформа створи. Тај интервал се одређује помоћу Y координате шеме и сабирањем, односно одузимањем, са променљивом *Offset.* Након одређивања граница и врсте платформе следи низ упита у којима се сваком броју додељује стварање једне од платформи помоћу *Instantiate* функције која је предефинисана функција за стварање објеката. За параметре функција тражи објекат који се копира, позицију копије и њену ротацију. У *Start* заглављу се такође ствара идентична копија велике платформе, на њој ће се налазити играч на самом почетку, а позива се и функција *SpawnPlatforms* да би постојала платформа на коју играч може да скочи.

4.4 Браве – кретња, стварање и откључавање

Кретање брава је реализовано на потпуно исти начин као и кретање платформи па то нећу навести овде. Њихово стварање је слично изведено као код платформи, само што се браве не стварају периодично, него у неком интервалу.

```
if (Timer >= Spawn)
{
	Instantiate(Lock);
	Spawn = Random.Range(MinSpawnRate, MaxSpawnRate);
	Timer = 0;
}
else
	Timer += Time.deltaTime;
```

У заглављу *Start* променљива *Spawn* добија насумичну вредност у интервалу минималног времена између стварања и максималног времена између стварања. Бројач се повећава док достигне ту вредност и онда се ствара нова брава по шеми (слично као и платформе), а *Spawn* постаје нова насумична вредност.

Откључавање браве, или друга чаролија, је једноставно имплементирана кодом испод:

```
if (Input.GetMouseButton(1) || Input.GetKey(KeyCode.Alpha2))
{
   LockBoxCollider.enabled = false;
   Sound.PlaySfx(Sound.Magic);
   Sound.PlaySfx(Sound.Unlock);
   Animator.SetBool("IsUnlocked", true);
}
```

Када се региструје тастер за извођење чаролије (или десни клик миша) *Collider* компонента браве се искључује и онда је омогућен пролаз кроз њу. Још једном, све линије кода везане за анимације и звукове ће бити обрађене.

4.5 Контрола брзине

Игра је испрограмирана да како време пролази постаје све бржа и самим тим и тежа за играча. За регулацију брзине је искоришћен објекат *SpeedController* који мења брзине кретања платформи, брава и чаробњака али мења и време између стварања платформи. Почећемо од брзине померања браве и платформе:

```
if (PlatformMoving.Speed < 10)
{
        PlatformTimer += Time.deltaTime;
        PlatformMoving.Speed = 3 + SpeedFactor * PlatformTimer * PlatformTimer;
        LockMoving.Speed = 3 + SpeedFactor * PlatformTimer * PlatformTimer;
}
```

Упит на почетку онемогућава да брзина платформе (уједно и браве) буде већа од 10. Да бисмо приступили брзини платформе и браве неопходно нам је да те променљиве буду *static* типа. Након упита, тренутне брзине се мењају и расту (у *Start* заглављу се стављa вредност на 3) и то у квадратној зависноти од времена ($Y=3+k*X^2$).

```
if (WizardStationarySpeed<10)
{
        WizardTimer += Time.deltaTime;
        WizardStationarySpeed = 3 + SpeedFactor * WizardTimer * WizardTimer;
        WizardMoving.Speed = 5 + SpeedFactor * WizardTimer * WizardTimer;
}
```

Чаробњакова брзина је регулисана истом једначином само се уместо 3 додаје 5. Чаробњак такође има и тзв. стационарну брзину која постоји да он не би стајао у месту док се платформе испод њега крећу. Саму вредност брзине чува променљива *WizardStationarySpeed* која ће мењати променљиву која је је повезана са *Transform*

компонентом скрипте чаробњака. Да бисмо приступили њој не можемо само да превучемо компоненту чаробњака у променљиву, неопходно је да самом чаробњакуca доделимо *Tag* (ознаку) и да у *Start* заглављу потражимо објекат који има одређену ознаку. Та ознака се додељује у Unity прозору у *Inspector* картици испод имена објекта. Код који ће променљиву у скрипти повезати са компонентом у другој је следећи:

```
WizardStationary = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>();
```

Код изнад повезује променљиву са компонентом чаробњака преко ознаке. Преко *WizardStationary* ћемо сада мењати позицију чаробњака следећим кодом:

```
WizardStationary.transform.position += Vector3.left * WizardStationarySpeed * Time.deltaTime;
```

Стационарна брзина је регулисана на овај начин због начина на који се игра завршава. У самом коду се чаробњакова чаробњакова скрипта искључује и онда би чаробњак проклизао по платформи јер нема стационарну брзину више. На овај начин он ће је имати и након краја игре и нестаће са екрана заједно са платформама.

```
if (PlatformSpawn.SpawnRate>1)
{
   SpawnerTimer += Time.deltaTime;
   PlatformSpawn.SpawnRate = 4 - SpawnFactor * SpawnerTimer * SpawnerTimer;
}
```

Последњи упит у овој скрипти регулише брзину стварања платформе. Због бржег кретања целе игре неопходно је да се платформе чешће стварају. Потребно је да се приближно у исто време заврши мењање брзине стварања и кретања платформи и да њихов однос остане приближно исти. За то су задужене променљиве *SpawnFactor* и горе наведени *SpeedFactor*, то су бројеви одређени да се ове две квадратне функције мењају приближно исто.

4.6 Логички менаџер

Логички менаџер је скрипта која се бави исписивањем резултата у горњем левом углу, памћењем најбољег резултата, приказивањем екрана за крај игре као и регулисање компоненти истог екрана.

Почећемо са бројачем резултата и његовим приказивањем.

```
if (Timer >= ScoreDelay && !GameOverActive)
{
   AddScore();
   Timer = 0;
}
else
   Timer += Time.deltaTime
```

Овај упит се налази у Update заглављу и позива функцију AddScore која додају 1 на резултат када год је бројач већи од ScoreDelay (након 2 секунде) и када је игра још активна тј. није дошло до краја игре.

```
public void AddScore()
{
   PlayerScore++;
   ScoreText.text = PlayerScore.ToString();
}
```

У AddScore функцији се, као што сам рекао, играчу додаје 1 на резултат, али се и текст резултата мења. То се дешава помоћу променљиве ScoreText која је повезана са објектом игре који представља текст.

```
public void GameOver()
{
   Wizard.GetComponent<WizardMoving>().enabled = false;
   PlatformSpawner.GetComponent<PlatformSpawn>().enabled = false;
   LockSpawner.GetComponent<LockSpawn>().enabled = false;
   GameOverActive = true;
   GameOverScreen.SetActive(true);
   if (PlayerScore > PlayerPrefs.GetInt("HighScore"))
      PlayerPrefs.SetInt("HighScore", PlayerScore);
   HighScoreText.text = "High Score: " + PlayerPrefs.GetInt("HighScore").ToString();
}
```

У функцији изнад се приказује екран за крај игре али се и само симулирање игре зауставља. Прве три линије кода онемогућују скрипте везане за платформе, браве и кретање чаробњака. Променљиве *Wizard, PlatformSpawner* и *LockSpawner* су повезане са компонентама објеката на исти начин како је и чаробњакова *Transform* компонента повезана у контролеру брзине. Променљива *GameOverScreen* је објекат који је заправо екран на крају игре, и постаје активан тада када је дошло до завршетка. У последњим редовима се проверава да ли је тренутни резултат бољи од најбољег резултата. Најбољи резултата у Unity-у на појединачном рачунару се памти преко *PlayerPrefs,* то су подаци игре који остају упамћени на рачунару и након њеног гашења. Такође након те провере се на екрану исписује тренутни најбољи резултат.

На самом екрану на крају игре се налазе три дугмета: *Play Again, Home, Exit.*

Редом имају функције поновног започињања игре, враћања на „кућни екран“ и излажења из игре.

```
public void Restart()
{
   SceneManager.LoadScene("Game");
}
public void Quit()
{
   Application.Quit();
}
public void Home()
```

```
{
   SceneManager.LoadScene("StartScene");
}
```

Горе су наведене функције за свако дугме. *SceneManager* је заслужан за активирање тзв. „сцене“ у игри. Ова игра има 2 сцене: сцену игре и кућну сцену (стартну сцену).

4.7 Крај игре

Крај игре је регулисан у скриптама чаробњака и шиљака.

```
if ((gameObject.transform.position.x<-10 || gameObject.transform.position.x>10 || gameObject.transform.position.y<-6))
            Logic.GameOver();
```

Код чаробњака се посматра његова позиција и у случају да пређе неку од горе наведених вредности, променљива *Logic* , која је повезана са скриптом логичког менаџера, позива функцију за крај игре.

```
   private void OnTriggerEnter2D(Collider2D collision)
   {
      Logic.GameOver();
   }
```

Код шиљака је то регулисано *Collider* компонентом која детектује кад неки објекат дође у контакт са шиљцима. У том тренутку се као и код чаробњака позива функција за крај игре преко променљиве *Logic.*

5

Звукови, анимације и стартни екран

5.1 Звукови

Звукови који се појављују у игри су преузети са интернета, а у за њихово пуштање у Unity-у је задужен објекат ког сам назвао *AudioManager*. Објекат поседује своју скрипту која има променљиве типа *AudioSource* (извор звука), *AudioClip* (сам звук који се пушта) и једну *float* променљиву која узима вредност клизача са стартног екрана и представља јачину звука. У *Start* заглављу се преко *PlayerPrefs* узима вредност клизача и пушта се позадинска музика преко извора звука који је намењен за позадинску музику. То у коду изгледа овако:

```
Volume = PlayerPrefs.GetFloat("Volume");
Background.volume= Volume;
Sfx.volume = Volume;
Background.clip = Music;
Background.Play();
```

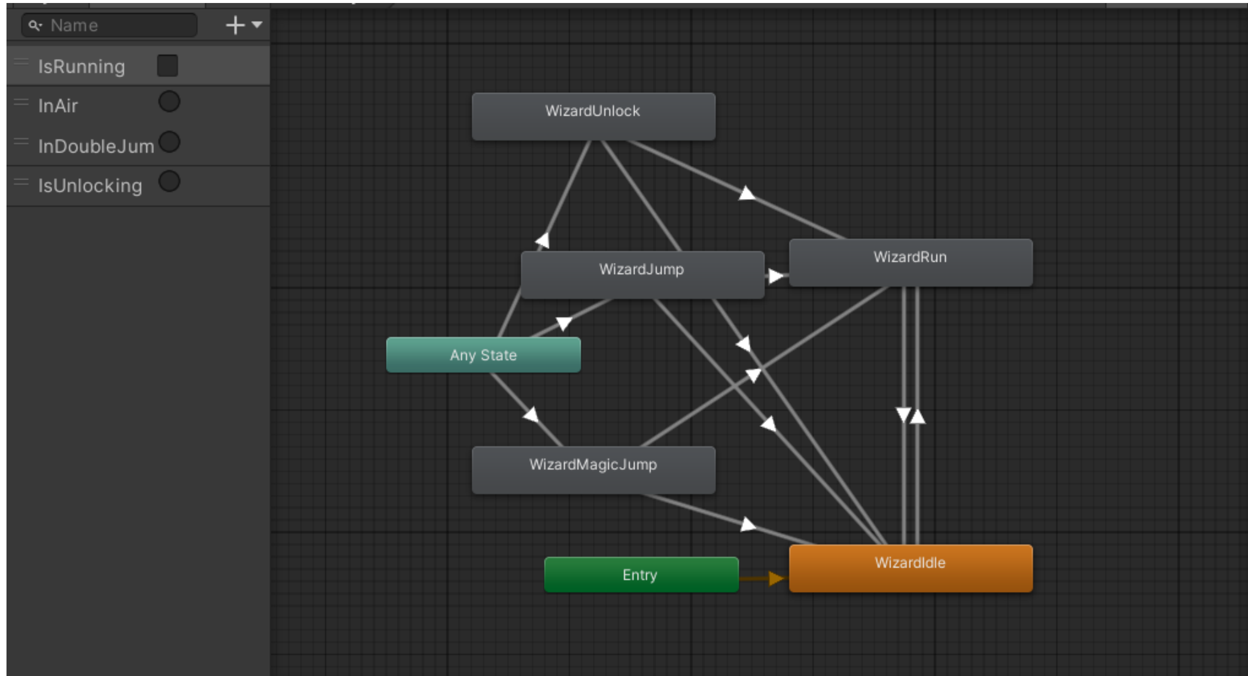
Што се тиче звучних ефеката (нпр. скок), већ смо виђали у коду да се приликом извођења одређене радње позива функција која је задужена за пуштање ефеката. То је следећа функција, при чему је *Sfx* извор звука:

```
public void PlaySfx(AudioClip Audio)
{
   Sfx.PlayOneShot(Audio);
}
```

5.2 Анимације

Сваки објекат који желимо да анимирамо мора да има компоненту *Animator* која се повезује са скриптом, и саму анимацију коју креирамо у Unity-у. Да би *Animator* одредио како да пређе из једне анимације у другу неопходно је да посматра променљиве које му корисник додели и да на основу њихове промене искључи једну и укључи другу анимацију. Такође треба и направити везу између две анимације и поставити променљиву по којој се мења анимација. У прозору за анимације се осим анимација налазе и

правоугаоници који говоре када ће се нека анимација одиграти. На пример *Entry* правоугаоник ће одмах при почетку игре покренути одређену анимацију, а *AnyState* ће анимацију покренути из било ког стања објекта ако се испуне услови променљивих. Испод је слика прозора за анимације.



Мењање вредности променљивих *IsRunning, InAir,* итд. смо могли да видимо у на пример чаробњаковом коду.

```
if (!Air)
   Animator.SetBool("IsRunning", true);
else
   Animator.SetBool("IsRunning", false);
```

На овај начин су реализоване све анимације у игри, па нећу сваку анимацију покрити појединачно.

5.3 Стартни екран

Сам стартни екран је врло једноставан, сачињен је од два дугмета, једног клизача за звук, наслова игре и позадине. Он такође има своју позадинску музику која се пушта као и у самој игри. Стартни екран је направљен у новој тзв. сцени и док је он активан сама игра није и обрнуто. Објекти који чине стартни екран и њихов начин функционисања су већ објашњени раније па их нећу објашњавати овде.

6

Закључак

За сам крај бих желео да кажем да је било врло интересантно и забавно радити на овој игри и да сам задовољан како сама игра изгледа. Циљ овог рада је био да прикажем процес рада и да можда заинтересујем још неког да покуша да направи своју игру, пошто технологија све брже напредује и прављење не само игара, него и разних других апликација, постаје све лакше.

Игра се може унапредити веома лако додавањем нове чаролије на пример или додавањем драгуља који повећавају резултат када се покупе. Границе не постоје, важно је само имати добру идеју и вољу за радом. Препоручујем свакоме ко је заинтересован у видео игре да покуша и направи своју прву игру.

Литература

1. Unity engine: here is how games are built: https://www.logiscool.com/rs/blog/coding-tech/unity-engine-here-is-how-games-are-built датум последњег приступа: 16.2.2024.
2. Unity (game engine): https://en.wikipedia.org/wiki/Unity_(game_engine) датум последњег приступа: 16.2.2024.
3. Историја видео-игара: https://sr.wikipedia.org/sr/Историја_видео-игара датум последњег приступа: 17.2.2024.
4. *Replay: The History of Video Games* – Tristan Donovan, Yellow Ant; Illustrated edition, 2010