

Универзитет у Београду
Математички факултет



Саша Грујић

Прављење 2D игара коришћењем погодности окружења Unity

Мастер рад

Септембар 2018

Универзитет у Београду - Математички факултет

МАСТЕР РАД

Аутор:	Саша Грујић
Наслов:	Прављење 2D игара коришћењем погодности окружења Unity
Ментор:	доц. др Александар Картељ, Математички факултет
Чланови комисије:	проф. др Миодраг Живковић, Математички факултет
	проф. др Владимир Филиповић, Математички факултет
Датум:	Септембар 2018

Резиме

Данас постоји велики број видео игара најразличитијих садржаја, за чији настанак је потребно много креативности и добрих идеја. У жељи за популаризацијом креирања нових видео игара и *Unity* окружења, настао је и овај рад.
У раду се описују погодности *Unity* окружења кроз пример прављења 2D игре. Обухваћени су сви важнији концепти у развоју једне игре. Објашњене су основе 2D простора, 2D модела и погона игре (енг. game engine), а потом и прецизније описани концепти *Unity* окружења.
Говори се о изради позадине, звуку и звучним ефектима, анимацијама, стварању играча и његовог контролисања, интеракцији у игри, изради специјалних ефеката, графичког интерфејса и тако даље.

Кључне речи:
развој игара, Unity окружење, 2D графика

Тати и Мами

Садржај

1 Увод **3**

1.1 Општи елементи развоја игре 4

1.2 Unity развојно окружење 4

1.2.1 Активе 5

1.2.2 Сцене 5

1.2.3 Објекат игре 6

1.2.4 Компоненте 6

1.2.5 Скрипте 6

1.2.6 Шаблони 7

1.3 Основе 2D игара у Unity развојном окружењу 7

1.3.1 2D простор 7

1.3.2 2D модели 9

1.3.3 Камера 9

2 Опис игре **13**

3 Развој Игре **16**

3.1 Опште о развоју 2D игара у Unity окружењу 16

3.2 Израда позадине 16

3.3 Играч 17

3.3.1 Анимирање играча 17

3.3.2 Кретање играча 20

3.3.3 Праћење играча камером 21

3.3.4 Имплементација пуцања 23

3.3.5 Ракета 26

3.3.6 Живот играча и наношење штете 27

3.3.7 Сакупљање поена 27

3.4 Противници 28

3.5 Декорација терена 29

3.6 Звук 29
3.6.1 Извор звука 30
3.6.2 Пријемник звука 30
3.7 Скрипте и контроле 31
3.8 Интеракције 32
3.8.1 Судар 32
3.8.2 Препознавање судара 33
3.9 Чврста тела и инстанцирање 35
3.10 Систем честица 37
3.11 Графички интерфејс и главни мени 38
3.11.1 Графички интерфејс 38
3.11.2 Главни мени 38
3.12 Тестирање и дистрибуција 39

4 Демонстрација тока игре **41**

5 Закључак **45**

1 Увод

Израда видео игре представља сложен процес који се може поделити у неколико фаза.
Груба подела процеса израде је:

1. Прављење прототипа игре
2. Израда дизајна
3. Развој игре
    - Израда изворног кôда
    - Израда текстура, модела, графичких ефеката, итд.
    - Израда звучних ефеката
4. Тестирање игре
5. Орђжавање

Игра [1] је најчешће развијена од стране тимова, иако понекад то може бити и мања група од неколико људи или чак и појединац. Издавач видео игара је компанија која објављује и промовише игре. Издавач игре не мора нужно бити иста компанија која је игру и развила.
Развојни тим обично чине: дизајнер, уметник (енг. game artist), програмер, дизајнер нивоа (енг. level designer), инжењер звука (енг. sound engeneer), тестер.
Кроз овај рад ће бити демонстриран животни циклус развоја једноставне игре намењене млађој публици. У даљем тексту, посебна пажња ће бити посвећена трећој фази израде, односно самом развоју игре коришћењем погодности окружења *Unity*.

1.1 Општи елементи развоја игре

Погон игре (енг. game engine) је систем за израду и развој видео игара. Погон игре брине о основним задацима као што су исцртавање, детекција и одговор на колизију, креирање анимација, кориснички унос и слично. Са тиме је постигнуто да уметници, дизајнери па и сами програмери могу бринути о самој имплементацији игре, а не о начинима на који ће основни задаци бити обављени. Програмери развијају моћне програмске пакете за погон игре у којима су укључени многи елементи који ће користити тим за развој игара.
Типични елементи и функционалности које пружа сваки погон игре су:

- Рад са звуком;
- Скрипте - подршка за писање нових програмских елемената у одређеном програмском језику;
- Анимације - 2D и 3D;
- Исцртавање (енг. Rendering) - 2D и 3D;
- Физички погон (енг. Phisics engine) - детекција колизије, одговор на колизију, физичке силе;
- Вештачка интелигенција (AI) - технике стварања привидне интелигенције;
- Управљање меморијом.

Погон игре најчешће омогућава апстракцију платформе допуштајући да се уз минималне преправке изворног кôда игре, иста игра покреће на различитим платформама (персонални рачунар, мобилни уређаји, веб, конзоле).

1.2 Unity развојно окружење

Unity је развојно окружење које је развила компанија Unity Technologies [2, 3]. У почетку је коришћен за развијање видео игара и симулација за компјутере,

конзоле и мобилне уређаје. Представљен је 2005. године за OS X оперативни систем и од тада се развио у вишеплатформско развојно окружење за чак 27 платформи од којих су и: Windows, MAC, Android, iPhone, iPad, Xbox 360, Play Station и друге.
Unity подржава 2D и 3D графику и писање скрипти у неком од 3 наредна језика: C#, UnityScript и Boo.

Основни концепти *Unity* развојног окружења су [4]:

- Актива (енг. Asset);
- Сцена;
- Објекат игре (енг. GameObject);
- Компонента;
- Скрипте;
- Шаблон (енг. Prefab).

1.2.1 Активе

Актива (енг. Asset) је основни градивни блок свих *Unity* пројеката. Од датотека креираних ван *Unity* окружења као што су слике, 3D модели и звучне датотеке, до неких типова датотека које се могу креирати у окружењу, као што су *аниматори*, *Unity* третира све датотеке које ће се користити за израду игре као *активе*.

1.2.2 Сцене

Сцена садржи објекте игре. Могу се користити за израду менија, индивидуалних нивоа игре (енг. levels) и било чега другог. Прецизније, свака сцена је посебан ниво игре или мени. Унутар сцене, игра се гради део по део, постављањем позадине, препрека, декорације и других објеката. Креирањем

игре у више сцена постиже се могућност покретања засебних делова игре те и лакшег тестирања тих делова.

1.2.3 Објекат игре

Све што постоји у животном циклусу *Unity* игре мора представљати објекат игре (енг. *GameObject*). Када се актива користи унутар сцене, она постаје нови објекат игре. Објекат не мора да имплементира функционалности, већ се понаша као носач (енг. container) за *компоненте* које надаље имплементирају сопствене функционалности. Да би се концепт објеката игре боље разумео, мора се упознати са компонентама.

1.2.4 Компоненте

Компоненте су функционални делови сваког објекта игре. Понашање објекта игре се контролише преко компоненти које су му додате. Компоненте се јављају у различитим облицима и могу утицати на понашање објекта који их садржи, дефинисати појаву и исцртавање или утицати на неке друге функције објекта унутар игре. Додавањем компоненте на објекат игре, додају се и нови делови погона игре том објекту и тако се модификује сам објекат на жељени начин.
Сваки објекат садржи барем једну компоненту - *Transform* компоненту. Ова компонента погону даје информацију о својој позицији у 3D простору помоћу X, Y, Z координата, његову величину и ротацију. *Rigidbody2D*, *Collider2D*, *ParticleSystem* и *Audio* су различите компоненте које се могу додати објекту игре.

1.2.5 Скрипте

Иако *Unity* има мноштво предефинисаних компоненти за разна својства, јасно је да ће некад бити потребно да се дефинишу сопствене функције играња. *Unity* дозвољава креирање сопствених компоненти коришћењем

скрипти. Скрипте нам дозвољавају активирање догађаја (енг. events) у игри, модификовање својства компоненте у реалном времену, одговорања на кориснички унос на жељени начин. Скрипте су важан и кључан концепт сваког развоја игре.

1.2.6 Шаблони

Шаблон је тип активе које је могуће користити у више сцена као и више пута унутар једне сцене. Шаблон нам омогућава да сачувамо предходно креирани објекат игре са свим компонентама и својствима. Шаблон се понаша као прототип од којег се могу креирати нове инстанце у игри. Свака инстанца повезана је са шаблоном и представља идентичну копију шаблона. Било каква промена шаблона се одражава на све његове инстанце, али, могу се изменити компоненте и подешавања за сваку инстанцу појединачно. Овако сачувани објекат се може касније користити у било којем делу наше игре, неограничен број пута.

1.3 Основе 2D игара у Unity развојном окружењу

1.3.1 2D простор

У физици и математици, дводимензионални простор или бидимензионални простор је геометријски модел планарне пројекције физичког простора. Две димензије се обично називају *дужина* и *ширина*. Када се говори о дводимензионалном простору, обично се мисли на Еуклидски простор. У геометрији, Еуклидски простор обухвата дводимензионалну Еуклидову раван, тродимензионалан простор Еуклидске геометрије и одређене друге просторе. Име је добио по грчком математичару Еуклиду из Александрије.

Иако ће 2D простор у *Unity* окружењу бити, у каснијим поглављима, де-

тачно описан, битно је да се напомене да *Unity* у позадини користи 3D простор јер је превасходно прављен за развој 3D игара. Дакле, *Unity* привидно ствара 2D простор, иако објекте рапоређује у 3D простору по дубини.

Координатни системи и координате

Координате су математички запис којим се описује положај објеката у простору. Њима се може описати положај сваке тачке у простору. Како радни простор може бити бесконачан у координатном систему, узима се и референтна тачка звана центар. Осе координатног система се најчешће означавају ознакама X и Y и Z, где Z представља дубину. *Unity* окружење користи координатни систем коме су осе међусобно нормалне. На овај начин се може описати положај, величина и ротација сваког елемента у простору (у овом раду ће бити описан положај, величина и ротација објекта игре). Тако је положај елемента одређен уређеном тројком која се састоји од три реална броја (x, y, z). Компонента која садржи информације о положају објекта у *Unity* окружењу је *transform* компонента. Координате центра су (0, 0, 0) па ће положај сваког елемента у простору дефинисати као удаљеност од центра координатног система.

Локални простор - Глобални простор

Као што је већ наведено, положај сваког објекта игре унутар простора дефинисана је као удаљеност од центра простора односно света. Такав простор се назива *глобални* простор а координате објекта које представљају удаљеност од центра су *глобалне* координате.

Уз такав простор, у сврху поједностављења, користи се и *локални* простор познатији као простор објекта. На овај начин се може дефинисати однос између различитих објеката. Овакав простор одређује нови координатни систем јединствен за сваки објекат. Сваки објекат игре има сопсвени координатни систем и центар (најчешће у центру објекта игре). Ово је практично при стварању односа између два или више објекта као што су однос родитељ-дете (енг. parent-child relationship). Тиме се може упоређивати положај,

ротација и удаљеност једног објекта у односу на други како је за објекат *дете* нова референтна тачка центар објекта *родитељ*. Однос родитељ-дете биће детаљније објашњен у каснијим поглављима.

1.3.2 2D модели

2D модел је геометријски модел објекта као дводимензионалне фигуре, обично у Еуклидовој равни. Иако су сви материјални објекти тродимензионални, 2D геометријски модел је често адекватан за одређене равне предмете, као што су папир и слично.
2D геометријски модели су такође погодни за описивање одређених врста слика, као што су технички дијаграми, лого слике, итд. Они су основни алат 2D рачунарске графике и често се користе као компоненте 3D геометријских модела.

2D рачунарска графика [5] или дводимензионална рачунарска графика, омогућава приказ неке слике на рачунару представљен у две димензије (као што је текст или дигитална слика). 2D графика на рачунарима се користи у топографији, картографији, техничком цртању, рекламирању и тако даље.
Спрајт (енг. Sprite) је 2D графички објекат. Користи се у комбинацији са позадином за приказивање стања разних објеката у игри.
Паралакс је привидна промена положаја објекта у односу на позадину услед разлике у положају двају посматрача, промене положаја посматрача или услед кретања посматрача великим брзинама. Паралакс је угао између две линије вида при посматрању једног објекта из два различита положаја. Док се позиција камере помера лево-десно, објекти у даљини изгледају као да се крећу спорије од објеката у близини камере.

1.3.3 Камера

Најбитнији део простора игре су камере [6]. Оне представљају наш поглед унутар сцене. Камера има прилагодљиво видно поље (енг. field of view (FOV)),

пирамидалног облика. Како су за *Unity* сви објекти унутар сцене објекти игре, тако је и камера објекат игре те самим тим садржи и *transform* компоненту која описује њену позицију и ротацију. Напоменуто је да *Unity* користи 3D простор за распоређивање 2D објеката тако да се и камера може лако поставити било где унутар простора, анимирати или придодати неком другом лику или објекту унутар игре. Ефекти као што су светло, замагљивање и слично, додају се самој камери како би се добила реална симулација људског ока. Могуће је додати и ефекте које људско око не може доживети као на пример светлеће траке при погледу у сунце. Многе сцене у модерним играма имплементирају више камера. *Unity* дозвољава креирање више камера унутар сцене и омогућава мењање главне камере током игре (изводи се помоћу скрипти).

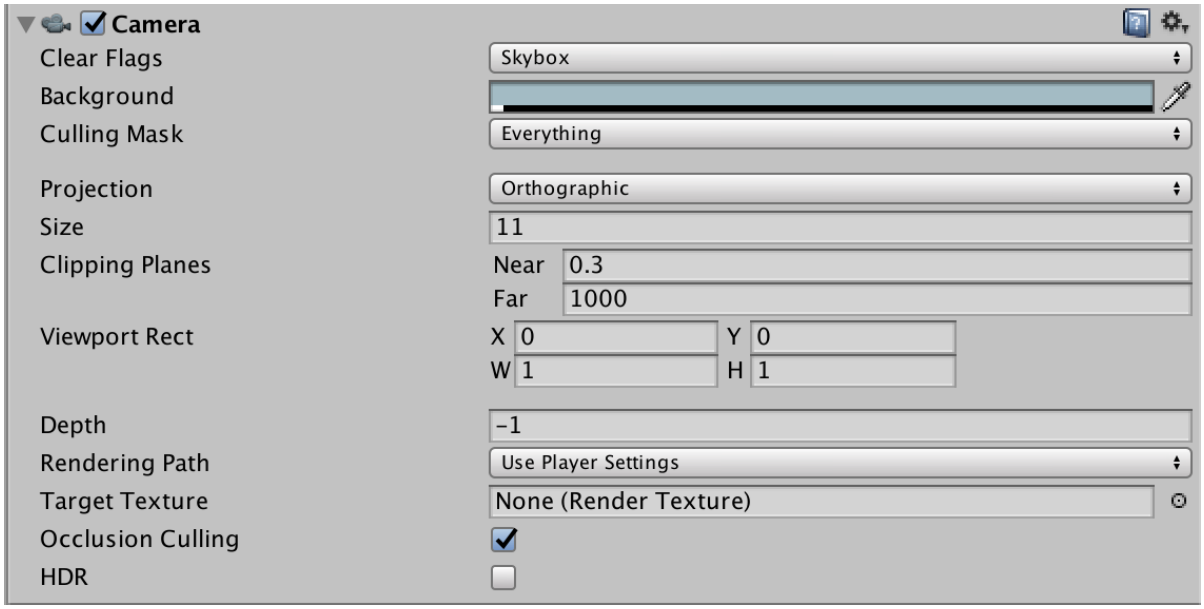
Unity *Camera* компонента

Уз *transform* компоненту коју има сваки објекат игре, објекат камере у сцени има и *Camera* компоненту, која је главна компонента задужена за успостављање гледишта. Да би било јасније како *Camera* компонента креира гледиште, потребно је проучити њене параметре.

Clear flags - Најчешће постављено на *Skybox* како би се омогућило исцртавање *skybox* материјала постављеног на сцени. *Skybox* је панорамска текстура која се исцртава иза свих објеката на сцени како би представила небо или било коју другу визуру на великој удаљености. Када се у сцени користи више камера, овај параметар омогућава подешавања камера ради исцртавања различитих делова сцене.

Background - Мењање боје која се исцртава иза свих објеката игре ако није постављен *skybox*. Гледиште које камера исцртава и које је празно, биће обојено изабраном бојом.

Projection - Омогућава камери да симулира перспективу. Могуће је мењати између перспекивног (пирамидални облик, симулација људског погледа) и



Слика 1: Компонента *Camera*

ортографског (правоугаони облик) типа пројекције. Ортографске камере се најчешће користе код стратешких и 2D игара.

Size - Величина приказа камере кад је подешена на ортографску.

Clipping planes - Удаљености од камере од којих се исцртава простор игре. Постоје предња и задња раван одрезивања. Предња је најмања удаљеност од које почиње исцртавање док је задња раван највећа удаљеност, односно удаљеност где исцртавање престаје.

Viewport rect - Четири вредности које индицирају где ће се на екрану камера позиционирати. Најчешће је поглед цео екран. Ово се мења када је потребно приказивати 2 или више погледа камера у исто време, на пример, у играма симулације вожње аутомобила, играч мора видети испред и иза себе (ретровизор).

Depth - Служи за одређивање приоритета камере.

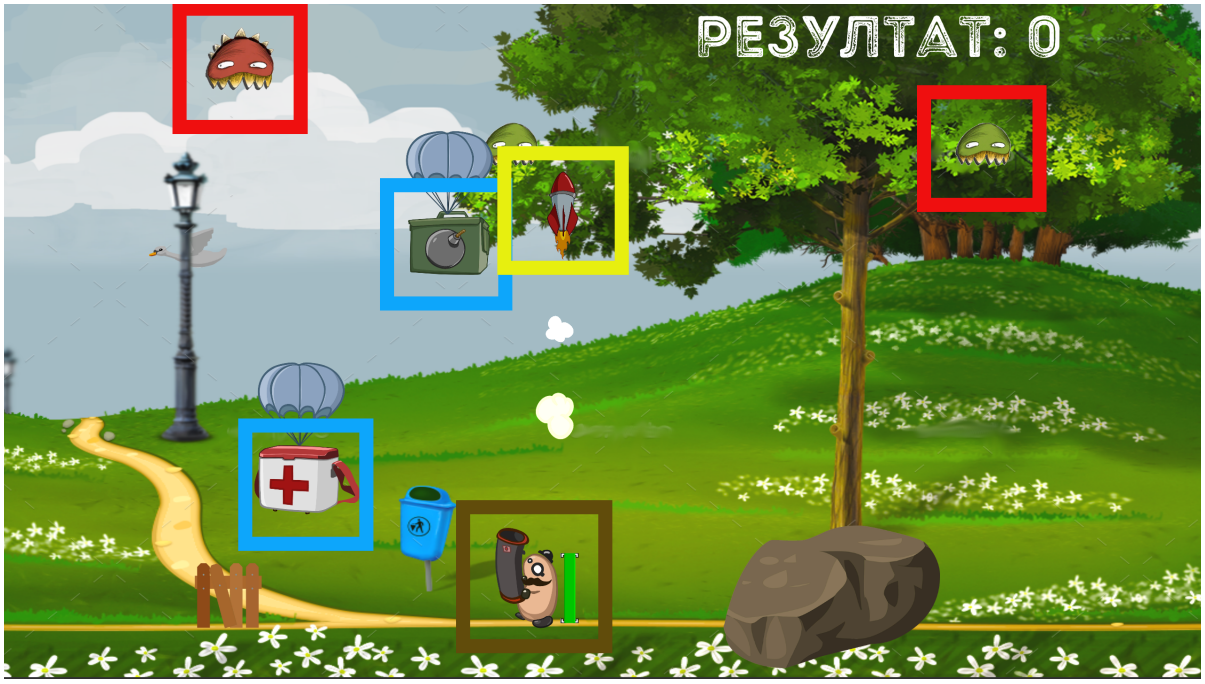
Culling mask - Укључује или изоставља слојеве објеката које ће камера исцртавати. Слојеви у *Unity* окружњу су додатни начин груписања објеката у циљу примене одређених правила на њих. Најчешће се користе за груписање и примену правила исцртавања осветљења и камере.

2 Опис игре

Име игре која ће бити реализована је *Још један дан*, скраћено ЈЈД. Игра се одвија на брдовитом пејзажу усред напада ванземаљаца.

Играч је лик, односно објекат у игри, чијим активностима непосредно управља корисник. Играч ће моћи слободно да се креће дуж граница екрана. Такође, играч ће моћи да испаљује **ракете**, објекте који ће у додиру са противником експлодирати и уништити противника.

Противник је лик, односно објекат у игри, чијим активностима управља сама игра, односно алгоритам. Игра ће садржати више врста противника.

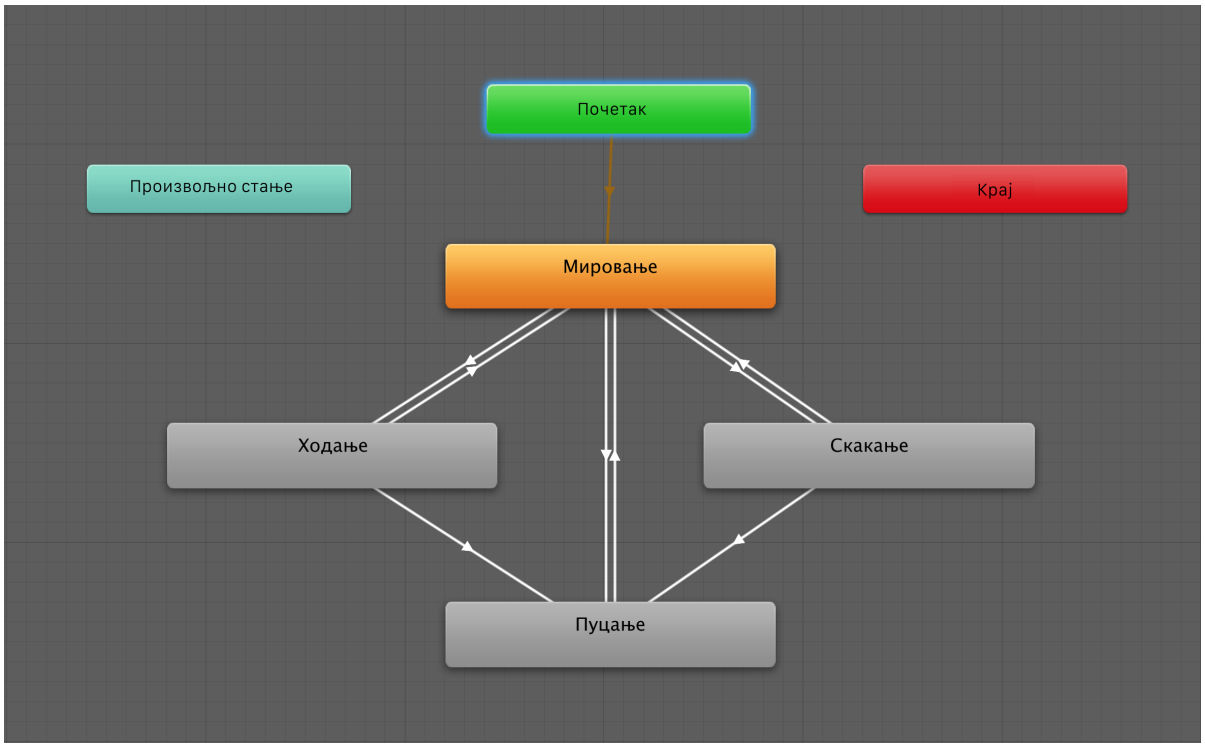


Слика 2: Објекти игре

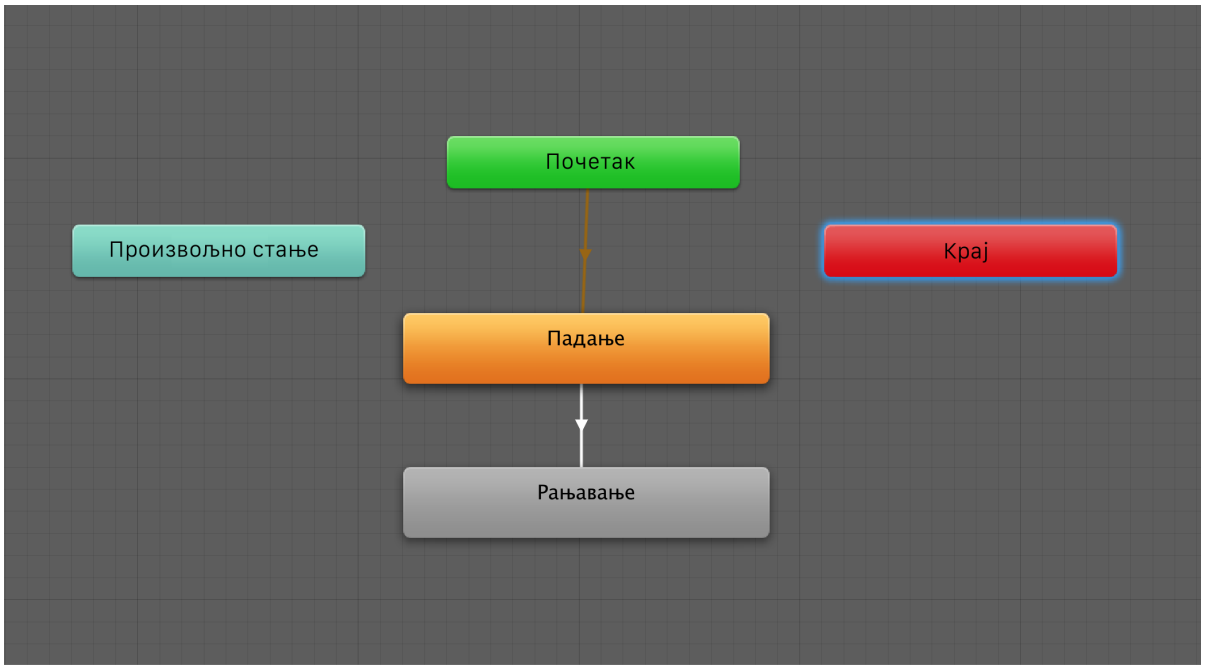
На Слици 2. играч је уоквирен браон бојом, противници су уоквирени црвеном бојом, ракета жутом док су специјални предмети уоквирени плавом бојом.

Енергија је особина играча, представљена нумеричком вредношћу. Ако противник додирне играча, играчу ће се одузети енергија сразмерна снази противника која је унапред дефинисана. Када се енергија спусти на нулу, играч бива уништен и игра се завршава.
Поени су представљени нумеричком вредношћу која описује играчев успех на крају игре. Играч добија поене за сваког уништеног противника као и за сваки сакупљени специјални предмет.
Специјални предмети су помоћни објекти у игри као што је медицинска помоћ која враћа енергију играча на почетну вредност.

Слике 3. и 4. приказују аутомат стања за анимације из *Unity* окружења што уједно представља и животне циклусе ових објеката. *Unity* окружење аутоматски повезује све објекте у животни циклус игре.



Слика 3: Животни циклус играча



Слика 4: Животни циклус противника

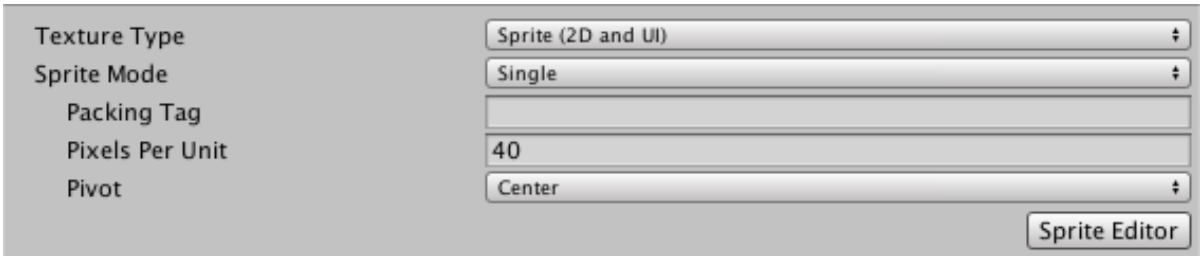
3 Развој Игре

3.1 Опште о развоју 2D игара у Unity окружењу

Могућност креирања 2D игара у *Unity* окружењу почиње од верзије 4.3. Игра ЈЈД је направљена коришћењем ових алата. Игра се комплетно ослања на *Sprite* објекте у *Unity* окружењу и уграђену подршку окружења за физичке симулације. Кроз развој игре ЈЈД, биће објашњено како се праве 2D игре у *Unity* окружењу. У овом поглављу ће бити изложени механизми креирања позадине, карактера, ефеката, камера, анимација и скрипти.

3.2 Израда позадине

Израда позадине [7] почиње скицирањем нивоа а затим реконструкцијом слојева у неким од графичких едитора као што је *Photoshop*. Извезене слике из алата *Photoshop* се могу унети у *Unity* коришћењем новог *Sprite* објекта (Слика 5).



Слика 5: *Sprite* објекат

Да би паралакс (енг. parallax) био касније креиран неке од слика се морају сачувати засебно и постављају се на различите слојеве дубине. Поред сортирања у различите слојеве, *Unity* подржава и сортирање унутар сваког слоја. Паралакс се добија тако што се жељене слике додају родитељу, који је празан објекат игре и који од компоненти има само скрипту која ће контролисати паралакс. У скрипти се израчунава у ком смеру се камера креће и колики је

померај а затим се та вредност користи за померање сваке слике позадине у супротном смеру од померања камере. Што слика позадине има већу "дубину" то је мањи померај (Слика 6).
Затим долази креирање тзв. елемената првог плана (енг. foreground elements) које ће карактери у игри заобилазити. Главни карактер игре ЈЈД ће моћи да се креће кроз окружење док ће противници падати са неба у границама нивоа.

3.3 Играч

Играч [7] је као и позадина прво дизајниран а онда реконструисан у алату *Photoshop*. У игри ЈЈД, играч се сатоји од неколико независних сличица (Слика 7а). Друга опција би била да је играч дизајниран као једна сличица, али би онда било неопходно направити анимацију за сваки кадар (енг. frame), што ће се користити и објаснити касније у раду.
Играч се састоји из више делова које треба уклопити тако да играч у одређеној ситуацији буде виђен на одређен начин уз одређене покрете. Након завршетка дизанирања, сваки део је сачуван на засебном месту да би *Unity* могао да их изолује као засебне *Sprite* сличице у *Importer* прозору. Ово значи да се сваки од ових *Sprite* објеката може гледати као засебни елемент при анимацији. Такође, сваком елементу ће бити подешена различита дубина, којом ће се исцртавати на екрану. На крају, сви *Sprite* објекти ће бити додати празном објекту игре, који ће садржати више компоненти као што су скрипте, граничници, компоненте за рад са физичким симулацијама и још неке.

3.3.1 Анимирање играча

Једном када је игач издвојен у посебне сличице, *Animation* прозор (Слика 8) ће бити коришћен за креирање анимација ходања, скакања, стајања у месту, и пуцања, тако што ће се користити свака од сличица засебно. Са

```
public class BackgroundParallax : MonoBehaviour
{
    // Niz svih pozadina ukljucenih u paralaks.
    public Transform[] backgrounds;
    // Procenat pomeraja kamere za pomeranje pozadine
    public float parallaxScale;
    // Razlika izmedju slojeva paralaksa.
    public float parallaxReductionFactor;
    // Glatkoca paralaks efekta.
    public float smoothing;

    // Referenca ka glavnoj kameri.
    private Transform cam;
    // Polozaj kamere u predhodnom kadru.
    private Vector3 previousCamPos;

    void Awake ()
    {
        // Setovanje kamere.
        cam = Camera.main.transform;
    }

    void Start ()
    {
        // previousCamPos uzima vrednost trenutne pozicije kamere.
        previousCamPos = cam.position;
    }

    void Update ()
    {
        // Paralaks se krece suprotno od kamere.
        float parallax = (previousCamPos.x - cam.position.x) * parallaxScale;

        // Za svaku pozadinu...
        for(int i = 0; i < backgrounds.Length; i++)
        {
            // ... setujemo x poziciju koja je jednaka njegovoj trenutnoj poziciji
            // plus paralaks pomnozen sa razlikom izmedju slojeva.
            float backgroundTargetPosX =
                backgrounds[i].position.x +
                parallax *
                (i * parallaxReductionFactor + 1);

            // Kreiranje Vector3 pozicije
            Vector3 backgroundTargetPos =
                new Vector3(backgroundTargetPosX, backgrounds[i].position.y, backgrounds[i].position.z);

            // Pomeranje pozadine od predhodne pozicije do nove.
            backgrounds[i].position =
                Vector3.Lerp(backgrounds[i].position, backgroundTargetPos, smoothing * Time.deltaTime);
        }

        // Setovanje predhodnog kadra na novi kadar.
        previousCamPos = cam.position;
    }
}
```

Слика 6: Скрипта која контролише паралакс

Animation прозором окружења *Unity* то је једноставно. Потребно је да се анимација дода родитељском објекту а затим креирају тзв. кључни кадрови



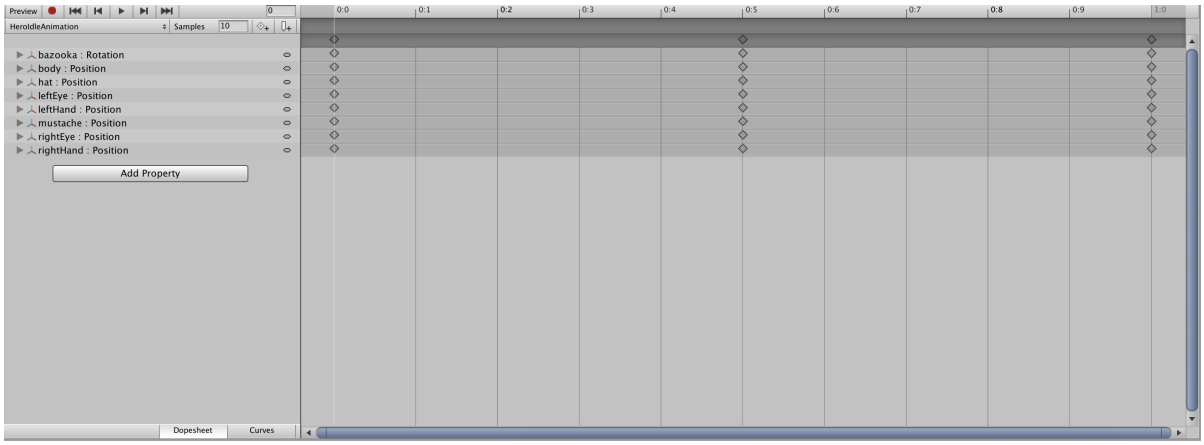
(a) Играч састављен из делова на различитим дубинама (3D приказ)



(b) Играч како га камера види

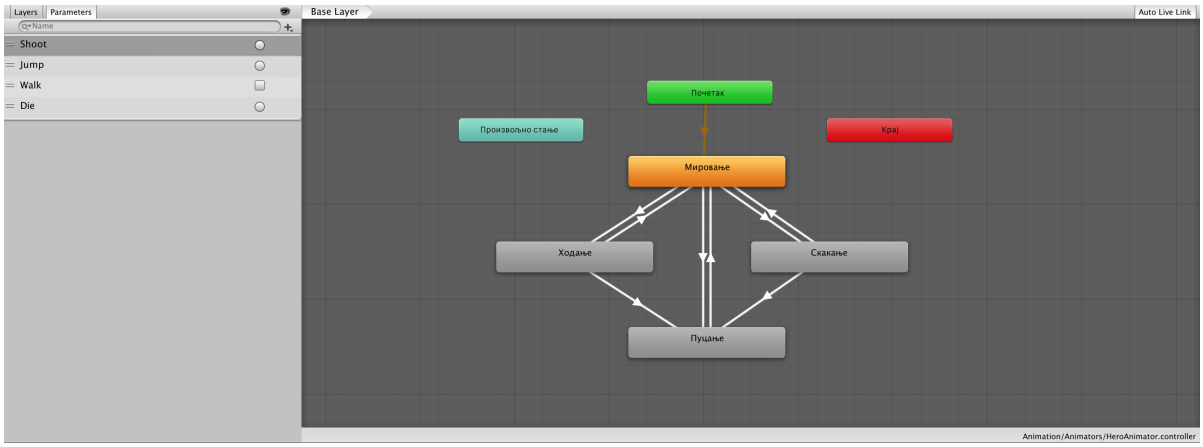
Слика 7: Играч

(енг. Keyframes) за сваки од дете-објеката. То се ради тако што се у различитим кадровима анимације дефинишу различита стања делова играча а *Unity* сам врши транзицију између тих кадрова.



Слика 8: *Animation* прозор

Када се заврши са анимацијама, може се креирати коначни аутомат за



Слика 9: *Animator* прозор

играча у *Animator* прозору (Слика 9), да би се, када се позову у кôду, различите анимације смењивале и репродуковале. Корисна особина *Animator* прозора је та што се у њему може подешавати и брзина анимације па се тако, једном направљена анимација, може убрзати или успорити да би се уклопила са физичким кретањима у игри ЈЈД без потребе за прављењем нових анимација.

3.3.2 Кретање играча

За контролисање играча и анимација, написана је скрипта у којој се помера играч додавањем физичких сила. То значи да се на играча и противнике може утицати у току игре у циљу добијања што динамичније игре. У скрипти која контролише играча се проверава унос који корисник зада, а затим се прослеђује аниматору који ће одредити која од анимација треба бити активирана али и обезбедити гладак прелаз између анимација креираних као стања. Такође, играчу се додају физичке силе (Слика 10), у зависности од улаза, да би био покренут у жељеном правцу.

У скрипти је решено у ком правцу ће се окренути играч, на основу хоризонталног улаза корисника, тако што се проверава да ли је улаз већи или мањи од 0. У *Unity* окружењу, функција *Input.GetAxis* представља вредности између -1 и 1 у складу са хоризонталним или вертикалним улазним

```
// Hvatamo horizontalni ulaz korisnika
float h = Input.GetAxis("Horizontal");

// Ako igrac menja smer (h parametar ima suprotan znak od velocity.x)
// ili nije dostigao maksimalnu brzinu.
if(h * GetComponent<Rigidbody2D>().velocity.x < maxSpeed)
    // ... dodajemo silu igracu.
    GetComponent<Rigidbody2D>().AddForce(Vector2.right * h * moveForce);

// Ako je igracevo horizontalno ubrzanje vece od maksimalne brzine...
if(Mathf.Abs(GetComponent<Rigidbody2D>().velocity.x) > maxSpeed)
    // ... setujemo igracevo ubrzanje na maksimalnu brzinu po x osi.
    GetComponent<Rigidbody2D>().velocity =
        new Vector2(Mathf.Sign(
                        GetComponent<Rigidbody2D>().velocity.x) * maxSpeed,
                        GetComponent<Rigidbody2D>().velocity.y
            );
```

Слика 10: Померање играча додавањем физичких сила

тастерима, а подразумевани су:

- A/D или стрелица лево / стрелица десно за хоризонтални унос
- W/S или стрелица горе / стрелица доле за вертикални унос

Када се не претисне ни један тастер, вредност функције ће бити 0. Стога, када корисник притисне стрелицу лево, на пример, вредност $Input.GetAxis("Horizontal")$ функције ће бити -1, када притисне стрелицу десно, биће једнако 1 а када пусти тастере, вредност ће се вратити на 0. Затим, у скрипти се позива функција која преусмерава X осу играча и тако ствара утисак да се играч окренуо на другу страну.

3.3.3 Праћење играча камером

Померање камере у играма је једна од битнијих ствари. Игра ЈЈД се ослања на механизам померања камере једне од најпознатијих игара свих времена, Super Mario [8]. У игри Super Mario, камера прати играча хоризонтално, али користи невидљиву маргину (енг. dead zone) у центру видног поља у ком

играч може да се помера без праћења камере. Када се играч помери преко ове маргине, камера почиње своје померање ка играчу.
У скрипти додатој камери ће се контролисати померање камере. Укратко, проверава се да ли је играч изашао из замишљене маргине и ако јесте камера се помера ка новој позицији помоћу $Mathf.Lerp$ функције која транслира камеру ка новој позицији играча за неки корак. Корак се одређује параметрима $xSmooth$ и $ySmooth$ који служе за подешавање глаткоће помераја камере. Параметри се множе са функцијом $Time.deltaTime$ која враћа време које је било потребно да се изврши предходни кадар и тако померај камере не зависи од броја кадрова у секунди (Слика 11).

```
void TrackPlayer ()
{
    // Na pocetku ciljne x i y koordinate kamere su njene trenutne koordinate.
    float targetX = transform.position.x;
    float targetY = transform.position.y;

    // Ako se igrac pomerio preko x margine...
    if(CheckXMargin())
        // ... ciljna x koordinata se treba translirati
        // izmedju trenutne pozicije kamere i trenutne pozicije igraca kroz vreme.
        targetX =
            Mathf.Lerp(transform.position.x, player.position.x, xSmooth * Time.deltaTime);

    // Ako se igrac pomerio preko y margine...
    if(CheckYMargin())
        // ... ciljna y koordinata se treba translirati
        // izmedju trenutne pozicije kamere i trenutne pozicije igraca kroz vreme.
        targetY =
            Mathf.Lerp(transform.position.y, player.position.y, ySmooth * Time.deltaTime);

    // Ciljne x i y koordinate
    // ne smeju biti vece od maksimalne i minimalne podesene vrednosti.
    targetX = Mathf.Clamp(targetX, minXAndY.x, maxXAndY.x);
    targetY = Mathf.Clamp(targetY, minXAndY.y, maxXAndY.y);

    // Setujemo poziciju kamere na novu ciljnu poziciju dok z osu ostavljamo po starom.
    transform.position = new Vector3(targetX, targetY, transform.position.z);
}
```

Слика 11: Функција која помера камеру ка играчу

3.3.4 Имплементација пуцања

У игри ЈЈД има доста ефеката, али најбитнији је могућност играча да убије противника. Играч пуца из базуке која има анимирани трзај. Ова акција је направљена из више делова. Прво, се ослушкује корисников улаз и када је притиснуто дугме за пуцањ, инстанцира се ракета, пушта аудио клип и активира анимација. На Слици 12 се види тренутак када играч пуца и убија противника.



Слика 12: Приказ пуцања играча и убијања противника

Детаљније, да би анимација пуцања могла бити активирана преко осталих играчевих анимација, креиран је посебан слој у аниматору тако да ће се ан-

имација пуцања извршити преко осталих анимација из основног слоја. Слојеви у аниматору се најчешће користе за управљање сложеним аутоматима стања анимације за различте делове тела објеката игре. На пример, када објекат игре користи доњи део тела за ходање, трчање или скакање а горњи део, независно од доњег, за пуцање.
Анимација се активира из било ког другог стања играча тако што се из кôда, помоћу *SetTrigger* функције, окида *shoot* параметар, који је предходно креиран у аниматору. *Unity* окружење подржава креирање 4 различита типа параметара: *Float*, *Int*, *Bool* и *Trigger*, којима се може приступити и доделити вредност из скрипти. Ово је начин на који скрипта може да контролише или утиче на активирање анимација.
У скрипти *GunController* (Слика 13), где се контролише пуцање, прво се проверава да ли је притиснуто дугме за пуцање. Ако јесте, извршава се *Shoot* параметар и шаље информација аниматору. *Shoot* параметар, који је окидач, је типа *boolean* и када аниматор искористи новоподешену вредност *true* за неки прелаз у друго стање анимације, атутоматски га враћа на *false*, да би се могао позвати поново, што је адекватно за акције какво је пуцање. Додатно, у скрипти се пушта аудио клип и инстанцирају се ракете које ће, за своје понашање, бринути саме. Скрипта *GunController* је додељена празном објекту игре који је дете играча. Кôд је додељен празном објекту игре јер то даље дозвољава позиционирање објекта на жељено место, у нашем случају на крај цеви базуке, а затим се користи његова позиција за инстанцирање ракета.

```
public class GunController : MonoBehaviour {

    private Animator anim;  // Referenca ka igracevoj animator komponenti

    public float canShootAgainAfterTime = 0.5f;

    private bool canShoot = true;

    [SerializeField]
    private GameObject bulletPrefab;

    void Start () {
        anim = transform.root.gameObject.GetComponent<Animator>();
    }

    void Update () {
        // Ako je pritisnuto dugme za pucanj...
        if (Input.GetButtonDown("Fire1"))
        {
            // ... setujemo okidac animacije i pustamo audio klip.
            anim.SetTrigger("Shoot");
            int i = Random.Range(0, shootClips.Length);
            AudioSource.PlayClipAtPoint(shootClips[i], transform.position);
            // pozivamo funkciju Fire()
            Fire();
        }
    }

    void Fire () {
        // Ako je pucanje dozvoljeno ...
        if (canShoot) {
            // ... instanciramo raketu
            Instantiate(bulletPrefab, transform.position, new Quaternion(90, 90, -10, 0));
            // zabranjujemo pucanje
            canShoot = false;
            // posle odredjenog vremena mogucnost pucanja je vracena
            StartCoroutine("canShootAgain", canShootAgainAfterTime);
        }
    }

    IEnumerator canShootAgain (float afterTime) {
        yield return new WaitForSeconds(afterTime);
        canShoot = true;
    }
}
```

Слика 13: *GunController* скрипта

3.3.5 Ракета

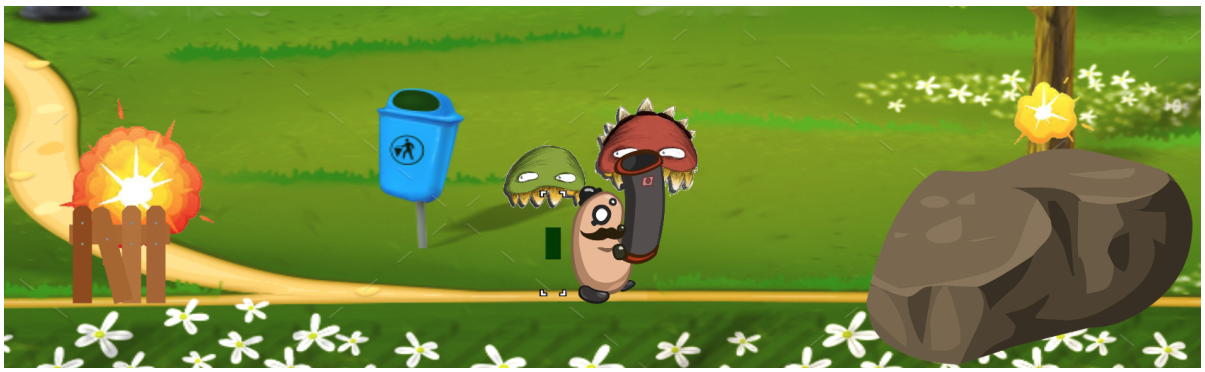
Ракета је посебан објекат у игри и њој се додаје физичко убрзање да би се покренула. Поседује издувни систем који је анимација, креирана од две сличице пламена типа *sprite*, и систем честица (енг. particle system) којим се представља дим. Систем честица прихвата графику типа *sprite*, па додавањем *sprite* сличица, које представљају делове дима, материјалу, који се даље може доделити аниматору система честица, добија се инстант анимација тих сличица. Када ракета додирне противника или неки део околине, уништиће саму себе и инстанцирати нову анимацију експлозије. Екпслозија је обичан објекат игре са аниматором као једином компонентом, који, при креирању објекта, активира анимацију експлозије. На Слици 14 се види ракета која лети ка противнику као и тренутак експлозије приликом додира са противником.



Слика 14: Ракета

3.3.6 Живот играча и наношење штете

Живот играча се чува као реална променљива типа *float*, и са сваким додиром са противником, позива се функција која одузима неки део живота играча. Ово се може десити само ако је прошло довољно времена од последњег додира са противником, да би било избегнуто брзо убијање играча. Како би играчу било дозвољено да што лакше побегне од противника и показати да је повређен, направљено је да сам чин наношења штете утиче на играча и физички. Да би ово било постигнуто, израчунавањем вектора од противника до играча, играч се помера у том смеру додавањем физичке силе. Када играч остане без живота, играч умире уз пратећу анимацију експлозије. На Слици 15 се види тренутак када противник додирује играча и наноси му штету.



Слика 15: Наношење штете играчу

3.3.7 Сакупљање поена

Скупљање поена приказује се са две анимације. Најпре се активира анимација доделе поена изнад играча а друга је анимација повећања укупних поена. Анимација доделе поена је направљена коришћењем две *sprite* слике,

нуле и јединице. Оне су додате празном објекту игре и креирана је анимација привидног одскакања бројева, а коришћењем једноставне скрипте биће избачени из сцене на крају анимације. Преглед укупних поена је обична GUI текст компонента са изабраним фонтом и скриптом која контролише поене играча. *Score* променљива, променљива која чува број поена, је *public* што значи да јој се може приступити из других скрипти и из њих може мењати.

3.4 Противници

У игри ЈЈД, постоје три врсте противника, зелени и црвени ванземаљац и ванземаљац са бродом као чувар. Зелени и црвени противник деле исту скрипту, јер им је понашање врло слично, а у скрипти се дефинише различита брзина кретања и количина потребних удараца да би противник умро. Противник са бродом користи другу скрипту јер су му кретање и анимација кретања другачије од осталих противника. Црвеном и зеленом противнику је креирана анимација падања помоћу различитих *sprite* сличица и активира се при креирању објекта. Анимација која се односи на противника са бродом је једноставнија јер само треба да се промени ротација објекта. Што се тиче механике померања, зелени и црвени противници падају слободно под утицајем гравитације физичке симулације окружења, док се противник са бродом помера додавањем физичког убрзања његовом физичком телу.
Да би противник био убијен, сваки пут када ракета дође у додир са њим, позива се функција која одузима 1 од дозвољеног броја погодака за одређеног противника, затим се води рачуна о животу противника и ако је једнак или је мањи од нула позива се функција која уништава противника. У овој функцији, код зеленог и црвеног противника, одрађује се више ствари. Прво се инстанцира анимација смрти. Затим, ако је противник убијен од стране играча, додају се поени за убијеног противника и активира се анимација поена. Ако противник није убијен од стране играча, то значи да је ударио у неки део позадине, тада се одузимају поени играчу, јер је противник на-

нео штету околини. Убијање противника са бродом додаје већи број поена играчу, затим деактивира анимацију кретања и повећава гравитацију да би противник пропао кроз ниво. Да би мртав противник био уклоњен из сцене, други објекат игре, правоугаоник који је постављен испод видокруга камере, детектује објекте и уништава њихове инстанце.

3.5 Декорација терена

Позадина је декорисана са разним покретним објектима да би се осећај уживања у игри подигао на већи ниво. Додати су летећи лабудови, као и трактор који пролази путем. Прво је додат лабуд, који је, на стандардан начин, креиран у алату *Photoshop* а заим убачен у *Unity*. Предност окружења *Unity* је што, када се превуку више сличица једног објекта, он аутоматски креира анимацију тог објекта мењајући те сличице једну за другом. Точкови трактора су одвојени од остатка возила па је анимација кретања креирана из више делова. Креирана је и скрипта која се може искористити више пута и која је задужена за инстанцирање ових објеката. Скрипта такође брине о времену појављивања сваког од објеката, брзини и где на екрану треба да се појави објекат.

3.6 Звук

Звук долази као позадинска музика, дијалози или звучни ефекти са циљем стварања већег доживљаја током игре, повећањем квалитета и тако даље.
Досег музике је велики и варира у разним играма од једноставне музике до оркестралне. Интерактивна мелодија у игри се мења у зависности од тога шта играч ради у датом тренутку, те тако ствара бољу атмосферу унутар игре. За велике пројекте у индустрији игара, музика се снима посебно и захтева скупу опрему и студио за обраду.
Звучни ефекти стварају већу реалност при одређеним догађајима, на пример, удар грома, кочење аутомобила, пуцање из оружија, цвркут птица и слично. Звучни ефекти се активирају помоћу скрипти у зависности од потребе.

Дијалози у игри се понајвише односе на снимљене и обрађене говоре у студију који се пуштају уз анимације у игри. Говори се снимају према сценарију игре.
Чланови развојног тима који брину о звуку су инжењери звука (енг. sound engineers). Снимање и обрада звука често укључује импровизацију и имитацију како би се добио реалистичан звук жељеног ефекта.
У окружењу *Unity*, за репродукцију звука су задужене две компоненте:

- Извор звука (енг. Audio Source)
- Пријемник звука (енг. Audio Listener)

3.6.1 Извор звука

Извор звука је сложенија компонента од пријемника и задатак јој је да пушта задати звук унутар свог домена у игри.
У игри ЈЈД, креиран је празни објекат игре и њему додата позадинска музика као компонента. За сваки извор звука *Unity* омогућава понављање (енг. loop) звучне датотеке. Тако ће се, са датотеком у трајању од неколико секунди, добити сталан звучни ефекат понављањем истог. Такође, *Unity* омогућава и подешавање гласноће и приоритета.

3.6.2 Пријемник звука

Пријемник звука се користи како би играчу било омогућено да чује све унутар сцене. На улаз прима сигнал од било којег извора звука који је у домету и репродукује звук на звучницима. У *Unity* окружењу, пријемник звука је компонента која може бити додељена било којем објекту. Са обзиром да се она понаша као ухо играча, најчешће је то компонента главне камере. Тако је и у игри ЈЈД додат пријемник управо главној камери. Важно је напоменути да унутар сцене може бити само један пријемник звука.

3.7 Скрипте и контроле

Иако се сматрају компонентама, скрипте су кључан појам у развоју игара. Као што је раније напоменуто, *Unity* подржава C#, UnityScript (UnityScript се заснива на JavaScript-у), а пре верзије 5.0, подржавао је још један језик, *Boo*. За овај део посла, у развојном тиму су задужени програмери великом већином. Игра ЈЈД је писана у C# па ће сви делови кода бити приказани у том језику. Као што је показано у раду, скриптама је описано понашање играча, противника, ракета, позадине и других. Важно је поменути неколико функција у животу једне скрипте:

- *Awake*( ) - Ова функција се позива пре било које *Start*( ) функције, без обзира да ли је инстанца скрипте омогућена или не, али после инстанцирања *prefab* објеката. Навише се користи за постављање било какавих референци између скрипти и иницијализације.

- *Start*( ) - Позива се након *Awake* функције али пре првог исцртавања, само ако је инстанца скрипте омогућена. Користи се за иницијализацију променљивих приликом стварања објекта којем је скрипта придружена. Ово омогућава одлагање било ког дела кôда иницијализације све док не буде потребан. На пример, противник може да буде прикључен игри и користити *Awake* функцију да му додели број муниције, али само да добије могућност да пуца, користећи *Start* функцију у тренутку када је та скрипта омогућена.
  Треба напоменути да ће функције *Awake* и *Start* бити позване само једном у животу скрипте која је прикључена неком објекту игре.

- *Update*( ) - Позива се једном по исрцртавању. Садржи сву логику која се извршава у реалном времену. Померања објеката на које не утичу физичке симулације, једноставни тајмери и детекција уноса корисника су само неке од примера које се обично извршавају у *Update* функцији. Треба приметити да се *Update* функција не позива у једнаким времен-

ским интервалима. Ако једном кадру треба више времена за исцртавање него следећем, онда ће време позивања *Update* функције бити различито.

- *FixedUpdate*( ) - функција слична *Update* функцији али са неколико битних разлика. *FixedUpdate* се позива у једнаким временским интервалима. Сва израчунавања физичког симулатора се дешавају непосредно пре ове функције. Дакле, сва логика која утиче на померање чврстог тела објекта игре, треба бити извршена у *FixedUpdate* функцији пре него *Update* функцији.

У скрипти је могуће приступити свим елемнтима објекта игре па самим тим га и мењати и управљати његовим садржајем.

3.8 Интеракције

У овом делу ће бити размотрени важни концепти за интеракцију играча и осталих објеката у сцени. Биће разјашњено шта је колизија и како се врши детекција колизије у *Unity* окружењу.

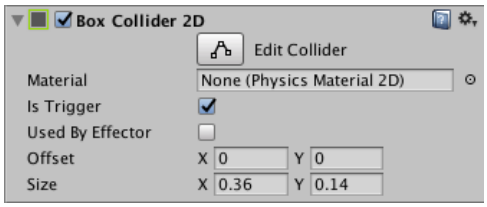
3.8.1 Судар

Судар [6] је појам за контакт два објеката и деловање односно размена сила између њих. У игри ће се пратити појава контакта између објеката и информације везане за тај догађај. На основу тих информација о судару, *Unity* може прикладно реаговати те нам приказати реалан одговор на колизију који имитира понашање по физичким законима у природи.
Рачунање детекције колизије захтева знање линеарне алгебре и геометрије. Све прорачуне о томе како ће елементи реаговати обавља *Unity*. Кориснику остаје да узима информације о догађајима колизије и складно томе реагује. Појава детекције и сакупљања информација о контакту са другим објектом позната је као препознавање судара (енг. Collision detection).

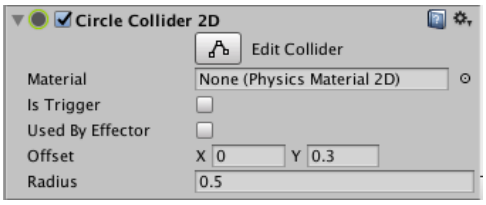
3.8.2 Препознавање судара

Препознавање судара најчешће се реализује преко *Collider* компоненте. *Collider* компонента је невидљива мрежа која окружује објекат игре и служи за препознавање контакта са другим објектима у сцени.
Препознавање судара се дели на 2 врсте: **дискретно** и **континуирано**.
Дискретно препознавање судара - код оваквог приступа препознавања, дефинише се мали интервал времена након ког се проверава да ли је дошло до судара објеката у сцени. Након сваког корака, ствара се листа објеката који се пресецају. Овакав тип се назива дискретним јер проверу врши у одређеним временским интервалима. Последица овога је што није могуће препознати судар тачно у тенутку догађања већ након одређеног временског интервала. Ово може довести до пропуста детекције судара неких ситних тела.
Континуирано препознавање судара - користи алгоритам за препознавање који је у могућности да врло прецизно предвиди путање физичких тела. Овај тип препознавања има предности јер онемогућава појаву пролетања једног тела кроз друго при великој брзини, познатију као *frame miss*.

У игри ЈЈД, препознавање судара се користи при откривању судара између ракете и противника, противника и играча, играча и помоћних кутија и тако даље. На пример, да би се успешно открило да ли је ракета погодила противника, ракети је додат *Box Collider* (Слика 16) који има облик квадрата док противник садржи *Circle Collider* (Слика 17) који, као што име каже, има облик круга. Постављањем атрибута *Is Trigger* се осигурава да *Unity* дозовљава пролаз ракете кроз *Circle Collider* противника и не рачуна даља физича својства судара. *BoxCollider* ракете само даје информацију да је дошло до пресецања која се може искористити како би се покренула анимација или уништио објекат игре.
У скрипти која контролише ракету, у функцији *OnTriggerEnter2D* (Слика 18), која се позива ако грничник (енг. collider) ракете дође у контакт са неким



Слика 16: *Box Collider* компонента



Слика 17: *Circle Collider* компонента

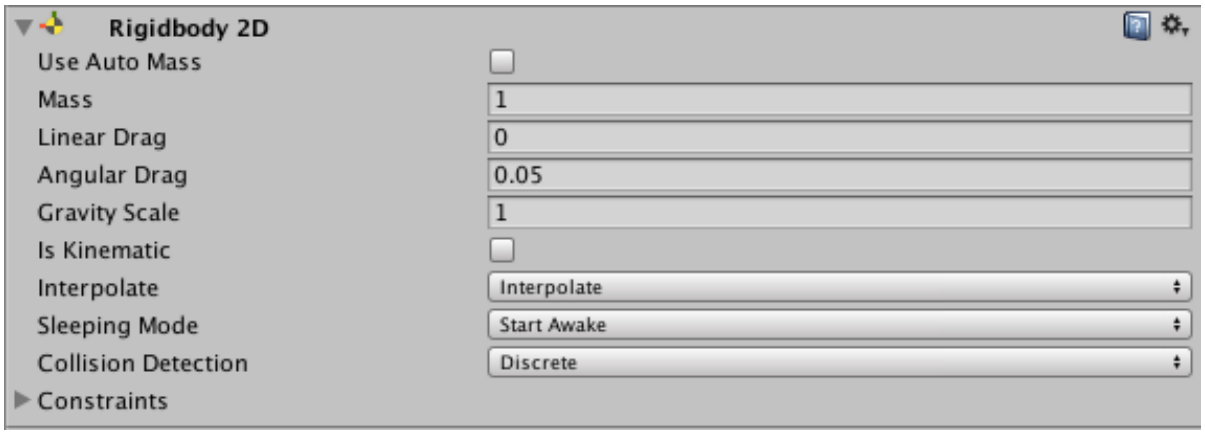
другим објектом, проверава се да ли је тај објекат противник и ако јесте, из противникове скрипте се позива функција која противнику наноси штету, а затим се инстанцира експлозија и уништење ракете.

```
private void OnTriggerEnter2D(Collider2D other)
{
    // Ako je pogodjen neprijatelj ...
    if (other.gameObject.CompareTag("Enemy"))
    {
        // ... pronadjemo EnemyController skriptu i pozivamo funkciju Hurt
        other.gameObject.GetComponent<EnemyController>().Hurt();
    } else if (other.gameObject.CompareTag("EnemyBoss"))
    {
        // ... pronadjemo EnemyBossController skriptu i pozivamo funkciju Hurt
        other.gameObject.GetComponent<EnemyBossController>().Hurt();
    }
    // instancira se eksplozija
    Instantiate(
        explosionPrefab,
        gameObject.transform.position,
        gameObject.transform.rotation
        );
    // Unistava se raketa
    Destroy(gameObject);
}
```

Слика 18: *OnTriggerEnter* функција у *BulletController* скрипти

3.9 Чврста тела и инстанцирање

Перформансе игре зависе од тога на шта све делује физички симулатор. Што је више објеката на које физички симулатор утиче, то је захтевније рачунање а самим тим су и перформансе слабије. Када се нови објекат дода у игру, он сам по себи није чврсто тело. Да би на објекат утицао физички симулатор, тада се објекту мора додати *Rigidbody* компонента (Слика 19).
Rigidbody или чврсто тело омогућава објекту игре да буде укључен у прорачуне физичког симулатора. *Rigidbody* може да делује под утицајем силе и помера се на реалистичан начин. Сваки објекат игре мора садржати *Rigidbody* компоненту да би био под утицајем гравитације, физичких сила додатих преко скрипте или интераговао са другим објектима кроз физички симулатор.



Слика 19: *Rigidbody* компонента

Rigidbody компонента нам омогућава подешавања која се узимају у обзир при рачунању физичких одзива:

- *Use Auto Mass* - Ако је омогућено, симулатор аутоматски препознаје масу тела из његовог граничника;
- *Mass* - Маса тела изражена у килограмима;

- *Linear Drag* - Коефицијент отпора при померању објекта;
- *Angular Drag* - Коефицијент отпора при ротацији објекта;
- *Gravity Scale* - Одређује колико гравитација утиче на објекат игре;
- *Is Kinematic* - Ако је омогућено, физички симулатор неће деловати на објекат али ће и даље имати све информације о овом телу;
- *Interpolate* - Дефинише како се интерполира померање објекта између физичких освежавања;
- *Sleeping mode* - Дефинише како објекат "спава" да би сачувао процесорско време кад није активан;
- *Collision Detection* - Дефинише како ће се препознавати судар између два објекта. О овоме смо причали у предходним поглављима;
- *Constraints* - Дефинише сва ограничења у померању објекта.

Да би се тело у игри понашало у складу са законима физике, подешавања морају одговарати реалним својствима елемената у природи. На пример, ако се каменчићу зада велика маса, несразмерна његовој величини, тада он неће летети далеко.

На самом почетку игре нису сви објекти унутар сцене. Неки настају касније, током игре, на одређеним местима или насумично а неке ствара сам играч. Сама појава стварања новог објекта у сцени током игре се назива инстанцирање. Инстанцирање се врши дефинисањем тачке стварања (енг. spawn point) и шаблона објекта који се треба створити.
Пример инстанцирања објекта у игри ЈЈД се види код противника, где су направљене три тачке стварања и у њима се креирају противници насумично (Слика 20).

```
void InstantiateRandomEnemy()
{
    int probabilityFactor = 100;
    // izaberemo nasumicno index instanciranja protivnika
    float index = Random.Range(0, probabilityFactor);
    // izaberemo nasumicno x koordinatu gde treba instancirati protivnika
    float randomX = Random.Range(-2, 2);
    // ako je index manji od nekog zadatog broja
    // onda instanciramo zelenog protivnika, u suprotnom crvenog
    if (index / probabilityFactor <= greenEnemyProbability)
    {
        // instanciranje zelenog protivnika
        Instantiate(
            enemies[0],
            new Vector2(transform.position.x + randomX, transform.position.y),
            transform.rotation
            );
    }
    else
    {
        // instanciranje crvenog protivnika
        Instantiate(
            enemies[1],
            new Vector2(transform.position.x + randomX, transform.position.y),
            transform.rotation
            );
    }
}
```

Слика 20: Функција која инстанцира противнике

3.10 Систем честица

Систем честица [9] (енг. *Particle system*) је позната техника у рачунарској графици за симулирање феномена као што су ватра, дим, киша, ефекти експлозије и слични ефекти које је тешко израдити класичним техникама. Коришћењем једноставних материјала и текстура које се називају честице (енг. particles) могу се добити многи ефекти и појаве у стварном свету. Честице предсављају 2D слике (*sprites*).

Систем честица у окружењу *Unity* је компонента која се додаје попут новокреираног објекта игре у сцену или већ постојеће компоненте постојећем објекту. Због компликованости компоненте, *Inspector* је подељен на неко-

лико подсекција од којих свака садржи неки скуп повезаних својстава. Када се изабере објекат игре који садржи систем честица, *Unity* окружење приказује мали *Particle Effect* прозор са неким основним контролама које су корисне за мењање подешавања система. **Playback Speed** се користи за убрзавање или успоравање симулације, тако да се унапред види како изгледа у неком другом стању. **Playback Time** индицира време од кад је систем стартован. **Particle Count** је тренутни број честица у систему. Дугмићи са врха панела се користе за стопирање и настављање симулације. У игри ЈЈД, овај систем се користи за креирање дима који оставља ракета, додавањем *sprite* типа систему честица.

3.11 Графички интерфејс и главни мени

3.11.1 Графички интерфејс

Током игре се прате одређене информације о догађајима у игри па ће такође неке од тих информација бити приказане играчу. На пример, број поена и број убијених противника. Дакле, играчу је омогућен графички интерфејс односно HUD (енг. heads up display). За приказивање информација у игри се користе компоненте: **GUIText** за приказивање текста на екрану и **GUITexture** за приказивање слика.
ScoreUI текст који приказује поене играча је *GUIText* компонента са прилагођеним фонтом и скриптом која управља играчевим поенима.

3.11.2 Главни мени

При раду са *Unity* окружењем постоје два могућа приступа израде менија:

- Коришћење GUITexture компоненте и управљањем догађајима миша.
- Коришћење UnityGUI класе.

Код првог приступа се користе властите слике израђене у другим алатима, а затим ће се помоћу скрипти контролисати догађаји као што су измена слике,

учитавање нове сцене и тако даље.
Други приступ је нешто тежи јер захтева генерисање целог менија помоћу скрипти па се сам изглед менија описује другим компонентама.
У игри ЈЈД се користи први приступ. Главни мени је посебна сцена у игри (Слика 21). Након што се слике уреде и увезу у *Unity* могу се произвољно распоредити по екрану. Свакој опцији се додељује скрипта која описује њено понашање. Скрипта треба да садржи функције које се позивају на одређени клик дугмета.



Слика 21: Главни мени

3.12 Тестирање и дистрибуција

По завршетку израде свих планираних делова игре потребно је уклонити евентуалне грешке које нису примећење током самог развоја. Тестирање игара је процес тражења и исправљања грешака. Грешке у игри се најчешће деле на:

- Грешке типа А - критичне грешке које онемогућавају дистрибуцију игре због неисправног рада или могућег пуцања игре.
- Грешке типа Б - мање грешке, игра је исправна, могуће је играти иако одређени делови нису функционални.
- Грешке типа Ц - мањи проблеми који долазе у облику предлога за измену.

Након што је игра тестирана спремана је за дистрибуцију. Дистрибуција игре се најчешће врши путем интернета или путем CD/DVD медијума. Након саме дистрибуције, игру је потребно одржавати и, у зависности од игре, надограђивати. Одржавање игре подразумева одржавање сервера (односи се на мрежне игре) али и поправке након дистрибуције.

4 Демонстрација тока игре

Као што је поменуто у раду, игру чине играч, противници, специјални предмети, бројни објекти позадине као и разне анимације експлозија. Кретање играча се контролише притиском стрелица лево/десно за кретање по хоризонталној оси, стрелица горе за скакање и клик миша за испаљивање ракета. На Слици 22 се види тренутак када је играч у скоку док испаљује ракете. Ту су и противници који слободно падају после насумичног креирања. Види се и тренутак када ракета погађа и уништава противника, за шта се играчу додељује 100 поена. За уништавање зеленог противника потребан је један погодак, док су за црвеног потребна два поготка.



Слика 22: Испаљивање ракете

На сваких 15 убијених противника појављује се један тзв. чувар, противник у свемирском броду. Када се појави противник чувар, игра стопира креирање осталих противника. По уништењу противника чувара, наставља се креирање нових противника. За уништење противника чувара је потребно 10 погодака

и играч као награду добија 1000 поена. На Слици 23 приказан је противник чувар као и рањени црвени противник.



Слика 23: Појављивање противника чувара

На Сликама 24 и 25 приказано је рањавање и потпуно уништење противника чувара. Виде се и неки објекти позадине као што су летећи лабудови и трактор.

На почетку игре играч има 10 живота. Када противник додирне играча, играчу се одузима неки део живота. Зелени противник са собом носи 1 живот, црвени противник 2 живота а противник чувар 5 живота. Када играч остане без живота, долази до краја игре. На крају игре, играч има увид у број освојених поена и опцију да почне игру из почетка (Слика 26).



Слика 24: Рањавање противника чувара



Слика 25: Уништење противника чувара



Слика 26: Крај игре

Извршна верзија игре за *MacOS* се може скинути са линка https://drive.google.com/drive/folders/1dqRbZDk2pj1pzY-qdMR7hu02ajRStZgE?usp=sharing

5 Закључак

Развој игара је широка област која захтева многе професије и знања из разних других области. Програмирање, дизајн, снимање и обрада звука и други су веома тешки послови. У имплементацији саме игре сви елементи морају радити складно како би се добио реалан приказ света.

Мањи пројекти који се најчешће односе на мобилне платформе, развијани су од стране појединца или мањег тима док су за модерне квалитетне игре задужени велики развојни тимови те је време израде дуже. Пројекат великих размера или мањи пројекат, свака игра има исте концепте који су објашњени у овом раду.

Као што је показано, *Unity* окружење олакшава креирање игара и тај посао чини занимљивим. Омогућава доста својстава и компоненти што у неким играма може бити велика уштеда времена. Са *inspector* прозором је могуће и лакше тестирање игре.

Након што је игра израђена она се тестира и дистрибуира. Након дистрибуције, игру је потребно одржавати.

Велики проценат људи сматра видео игре забавним те оне постају платформа која се, осим за забаву, може користити за образовање па чак и у медицинске сврхе. На пример, игра *SIMS* је игра симулације живота, у којој корисник управља животом једног појединца а касније и породице, укључујући и тражење посла и изградњу куће. Едукативни део ове игре се састоји у томе што корисник у тренутку изградње куће за свог карактера, кроз слике намештаја, које поставља по кући, учи енглеске речи.

Постоје игре које су реалне симулације лета авиона, хеликоптера, операције пацијента и тако даље.

Литература

[1] ERIK BETHKE, "Game development and production", 2003

[2] RAY WENDERLICH, "Unity" https://www.raywenderlich.com/category/unity

[3] UNITY, "What is a game engine" https://unity3d.com/what-is-a-game-engine

[4] WILL GOLDSTONE, "Unity Game Development Essentials", 2009

[5] JOHN PILE JR, "2D Graphics Programming for Games", 2013

[6] UNITY, "Manual" https://docs.unity3d.com/Manual/index.html

[7] UNITY, "Tutorials" https://unity3d.com/learn/tutorials

[8] NINTENDO, "Nintendo Games" https://www.nintendo.com/games/detail/arcade-archives-vs-super-mario-bros-switch

[9] RAY WENDERLICH , "Introduction To Unity: Particle Systems" https://www.raywenderlich.com/138-introduction-to-unity-particle-systems