

- Семинарски Рад А -

Видео игра

**Професор  
др Ђорђе Херцег**

**Студент  
Тојагић Растко 72/15**

Table of Contents

[Увод 4](#_Toc442465623)

[Задатак 5](#_Toc442465624)

[Основни појмови 6](#_Toc442465625)

[Едитор 6](#_Toc442465626)

[Ресурси 9](#_Toc442465627)

[Организација фолдера 9](#_Toc442465628)

[Сцене 10](#_Toc442465629)

[Класа Monobehaviour и главна петља игре 10](#_Toc442465630)

[Game Object 11](#_Toc442465631)

[Component 11](#_Toc442465632)

[Спрајт 12](#_Toc442465633)

[Колајдер 12](#_Toc442465634)

[Rigidbody 12](#_Toc442465635)

[Delta Time 12](#_Toc442465636)

[Класа PlayerPrefs 13](#_Toc442465637)

[Particle System 13](#_Toc442465638)

[Идеје за имплементацију 13](#_Toc442465639)

[Играч 13](#_Toc442465640)

[Кретање 13](#_Toc442465641)

[Умирање 14](#_Toc442465642)

[Пуцање 14](#_Toc442465643)

[Непријатељи 14](#_Toc442465644)

[Кретање 14](#_Toc442465645)

[Детаљан преглед изворног кода 15](#_Toc442465646)

[Singleton pattern 15](#_Toc442465647)

[Class PlayerController 15](#_Toc442465648)

[Class IEnemy 18](#_Toc442465649)

[Методе ApplyDamage() и Death() 19](#_Toc442465650)

[Class KerbanautInvader 19](#_Toc442465651)

[Class Score 21](#_Toc442465652)

[Закључак 23](#_Toc442465653)

# Увод

Циљ овог пројекта јесте креирање видео игре помоћу алата Unity3D и програмског језика C#, који Unity користи као скриптинг језик.

Unity3D је бесплатан „game engine“ коришћен од стране девелопера из целог света, како индивидуалних, малих „indie” тимова, тако и великих озбиљнијих компанија као што јe „Blizzard Entertainment”.

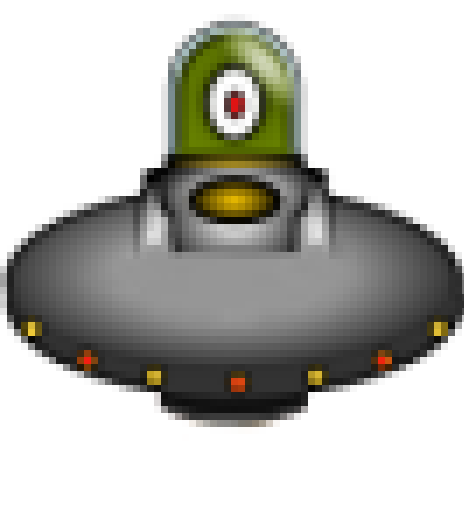




Unity је веома популаран због могућности лаког пребацивања апликација на друге платформе. Поред Windows, Mac OS X, Linux, подржане су и Android, iOs, Windows Phone, PS3, Web GL и многе друге платформе, а људи који раде на овом „game engine”-у труде се да пруже подршку и за нове уређаје који данас излазе на тржиште великом брзином.

# Задатак

Оно што такође даје предност Unity-у у односу на друге софтвере овог типа, јесте лакоћа коришћења, па је стога могуће веома брзо склопити сасвим солидан демо пројекат за кратко време и без много муке. Дакле задатак јесте направити игру! Прецизније, дводимензоналу свемирсуе пуцачину, или бар део исте.



Па, хајде да поделимо овај један задатак на неколико мањих целина, које су уједно „могућности“ и „механике“ ове игре.

1. Главни мени – Наравно, свака игра га садржи, и из њега бирамо понуђене опције, које ће нас:
   1. одвести у игру
   2. приказати најбољи скор
   3. или ипак изаћи из ње.
2. Демо ниво – Овде испробавамо нове ствари, пре него што их имплементирамо у игру
3. Главни ниво – Ово је наша игра, а механике су:
   1. Кретање, односно померање по екрану, у четири правца
   2. Пуцање
   3. Освајање бодова
   4. Могућност умирања, те крај игре

# Основни појмови

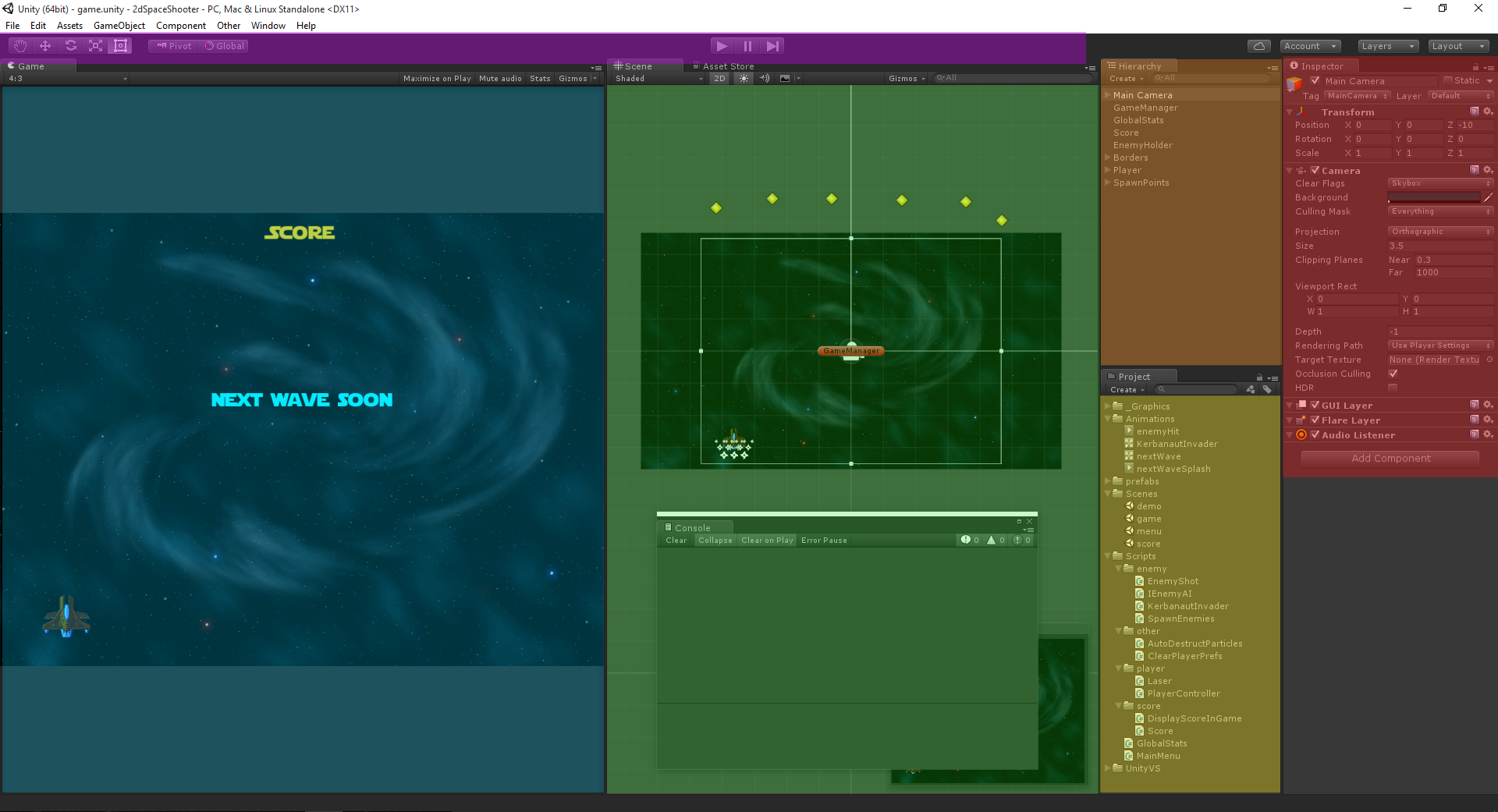
Алат који користимо, поред осталог, обезбеђује нам визуелно окружење или едитор како ћемо га даље називати. Целокупна апликација, односно игра подељена је на сцене, а ресурси су организовани по фолдерима, и Unity их аутоматски конвертује у потребне формате, или само компресује. Наредних неколико поднаслова бави се детаљнијим објашњењем ових појмова, а након тога залазимо дубље у саму игру, односно појмове као што су главна петља игре, идеја иза кретања карактера, непријатељска вештачка интелигенција...

## Едитор

Поред великог броја класа које имплементирају рендеровање, осветљење, виртуелни простор и кретање кроз њега, физике, главну петљу игре, овај софтвер нам даје и свој едитор који веома олакшава ствари попут постављања објеката у простор, додељивања понашања(односно наших класа) ТИМ објектима, лакше организације аудио и визуелних елемената по фолдерима у пројекту, и најбитније омогућава нам тестирање игре унутар едитора без потребе за прављењем „.exe”-а апликације.

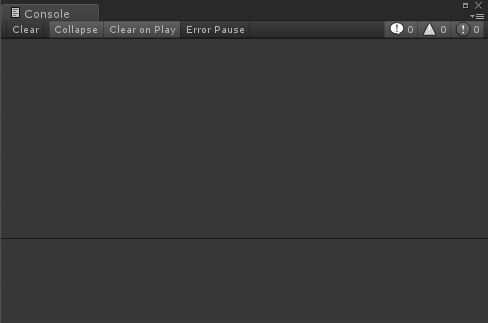
Следећа слика приказује едитор, и по бојама је подељена на целине.

*\*Напомена: Распоред прозора у едитору може да се мења превлачењем на жељену позицију, стога почетни распоред истих може и не мора да се разликује од ове слике.*

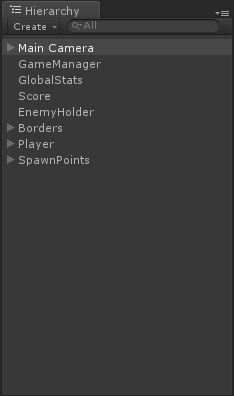
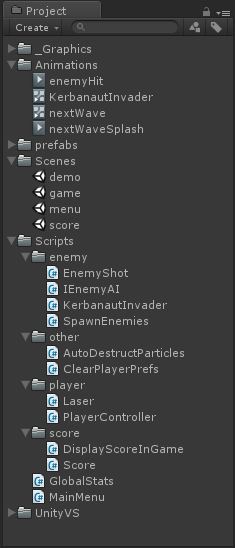


Едитор је подељен на два велика прозора, и неколико мањих целина са стране. Оба ова прозора приказују нашу „сцену“ на којој се одвија радња игре, али плави прозор приказује сцену унатар саме игре, односно како би то изгледало у готовој игри, док у зеленом прозору можемо да манипулишемо објектима, померамо их, обележавамо графички(симболима, текстом), додајемо и одузимамо.

У оквиру зеленог прозора, налази се још и конзолни прозор(који је сасвим случајно завршио баш на овоме месту). Он нам олакшава дебаговање апликације(омогућава исписивање порука услед извршавања одређених делова кода) и исписује могуће грешке .



Даље имамо прозор који приказује главни фолдер и подфолдере нашег пројекта, и њихов садржaj.



(десно)Прозор који садржи све објекте који се тренутно налазе на сцени, и као што видимо, тренутно обележени објекат јесте Главна камера.Наредни прозор, даје нам увид у подешавања тренутно обележеног објекта. Овде можемо да видимо позицију, ротацију и величину објекта у простору, као и компоненте(или наше класе) закачене на њега.



И коначно, у горњем делу слике, означено љубичастом бојом, налази се „трака“ са алатима за померање, ротацију и скалирање објекта, као и дугмићи за покретање и паузирање игре.

## Ресурси

Ресури, или Assets, јесу графички елементи, звук, анимације, и скрипте односно класе које сами пишемо. Организовани су по фолдерима.

## Организација фолдера

Укратко, преглед фолдера из пројекта...

* \_Graphics
  + Овај фолдер садржи, као што и само име наговештава, све графичке елементе који се користе у игри, а направљени су помоћу неког од програма за едитовање графике(Photoshop, GIMP).
* \_Sound
  + Овде се налази звук, такође направљен ван Unity-а.
* Animation
  + Овде се налазе анимације неких објеката у игри, направљене у Unity-у.
* Prefabs
  + Постоји могућност чувања објеката(и даље не причамо о објектима класа) у префабе, што касније омогућава њихово стварање на сцени. На пример, желиимо да стварамо непријатеље истог типа, спаковаћемо непријатеља у префаб, и одатле стварати копије истих.
* Scenes
  + Игра је организована по сценама, што би било исто што и екрани који се појављују у игри(Главни мени, скор и сама игра).
* Scripts
  + У овом фолдеру налази се сав код наше игре, односно све класе које смо написали.
* UnityVs
  + Ово је додатак који омогућава комуникацију Unity-а са MS Visual Studio софтвером за развој Windows апликација(Visual Studio није неопходан, већ се користи само као едитор за код и дебагер ако се закачи за Unity).

## Сцене

Сваки пројекат можемо поделити на сцене, првенствено ради лакше организације. На сцене онда додајемо ресурсе који су потребни. Обично имамо главну сцену у којој се одвија сама игра, и помоћне сцене попут главног менија, опција...

Сцене су засебне целине, и могуће је кратње кроз њих, али постоји један проблем, а то је очувавање вредности променљивих. Ако бисмо имали неколико нивоа у нашој игри, и за сваки ниво засебну сцену, и желимо да играч пређе из једне сцене у другу а да се при том не изгуби, рецимо његов тренутни скор, ово би нам направило велики проблем. Срећом, Unity решава ово без много муке, коришћењем класе PlayerPrefs.

## Класа Monobehaviour и главна петља игре

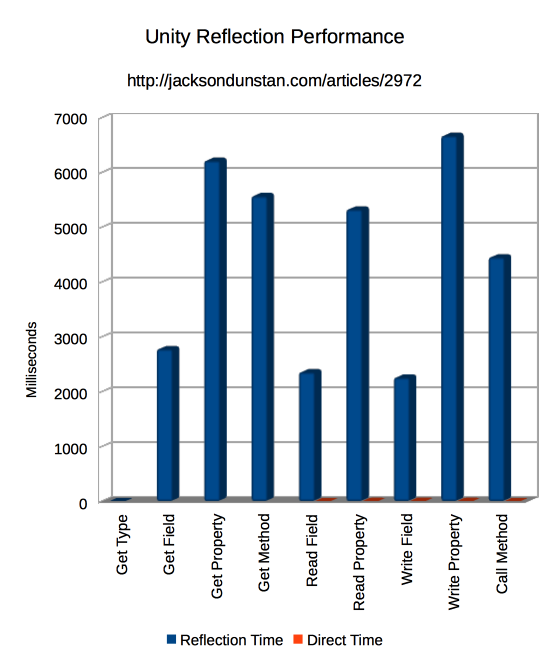
Основна класа, коју обично наслеђују све класе које ми пишемо, јесте Monobehaviour. Ова класа садржи своје методе, и такозване „поруке“ односно методе које се позивају када се догодила одређена радња.

Тако се, метода Awake() позива када се направи инстанца, односно објекат наше класе. За разлику од ове методе, метода Start() позива се првог фрејма након што објекат наше класе „оживи“. И сада долазимо до главне петље игре, а то је метода Update().

Ова метода, односно главна петља игре, позива се сваког фрејма. Unity нам омогућава да у свакој класи имамо Update() методу, тако да не морамо да мислимо много како ћемо уклопити све наше функционалности које зависе од времена, тј ове методе.

Али све већим бројем Update() метода долазимо до проблема. Корисници су уочили(кроз разне провере и тестирања) да се позиви интернтих метода односно „порука“ врше помоћу рефлексије, System.Reflection namespace се користи. Ово наравно није званично, и девелопери који раде на развоју Unity-a су покушали ово да заташкају али тестирања показују супротно.

Наредна слика показује да су овакви позиви метода далеко захтевнији временски, чак неколико хиљада пута, па ако би имали стотину класа које садже Update(), знатно би утицало на перформансе. Ово су резултати теста чији код се може наћи на линку, записаном на слици:



*\*Напомена: Крајем Децембра 2015., на официјалном сајту Unity-a, порекли су коришћење рефлексије, и објаснили нови начин на који су имплементирали позиве метода. Више на <http://blogs.unity3d.com/2015/12/23/1k-update-calls/>*

## Game Object

Основна класа за све ентитете(објекте) на сцени.

## Component

Ово је основна класа за све што качимо на наше ентитете(објекте), односно све скрипте које напишемо и додамо на објекте.

## Спрајт

Свака 2D сличица коју користимо у игри назива се спрајт. Она се на екран рендерује помоћу истоимене класе, која садржи и нека додатна подешавања као што је транспаренција.

## Колајдер

Колајдер је компонента која, што се и из самог назива може закључити, омогућава детектовање судара међу објектима на сцени. Деле се на 2D и 3D колајдере, који могу бити различитих облика: коцка, лопта, круг, квадрат, и такозвани mesh collider који поприма облик самог објекта(3D модела или спрајта). Да би се детектовао судар потребно је да објекат на себи има закачену и компоненту rigidbody.

## Rigidbody

Додавањем ове компоненте на објекат, он постаје „физичко“ тело, односно на њега утичу закони физике, дефинисани у самом Unity-у. Ова класа садржи и бројне методе помоћу којих се манипулише кретањем објекта, омогућава детектовање судара и симулира понашање тела при судару са другим.

Да би уопште могли да детектујемо судар, објекти на сцени морају имати закачену ову компоненту на себе, са могућношћу да искључимо физику тако што чекирамо опцију Kinematic, на самој компоненти.

## Delta Time

Time.deltaTime је време у секундама које је било потребно за извршавање последњег фрејма. Ако померамо објекат сваког фрејма, множећи вектор правца са брзином, потребно је да помножимо и са ово променљивом, како би се наш објекат кретао ЈК/секунд уместо ЈК/фрејм.

ЈК – Јединица кретања

## Класа PlayerPrefs

Ова класа нам омогућава да сачувамо потребне нам вредности, и запишемо их на диск, наравно ово је запаковано и заштићено од измене ван игре. Уместо да користимо базу података, што би додатно компликовало ствари, једноставно користимо ову класу.

Конкретно, у овом пројекту, класу PlayerPrefs користићемо да сачувамо најбољи скор.

## Particle System

Particles(честице) користе се при изради графичких ефеката као што је експлозија, киша, ватра и слично. У нашем пројекту, партикле користимо да бисмо нагласили да је играч погођен од стране непријатеља и да је непријатељ уништен(експлозија).

Ова компонента емитује честице на основу задатих параметара

# Идеје за имплементацију

## Играч

Основне механике јесу кретање, пуцање и умирање.

Наша игра има један ниво, који траје све док је играч жив, а кретање му је ограничено величином екрана, али треба напоменути да је играчу ограничено кретање по Y оси до отприлике половине екрана.

### Кретање

Кретање је могуће ограничити на неколико начина. Један од начина био би коришћење физике, односно колајдера и он је можда једноставнији за имплементацију зато што је потребно додати четири „невидљива“ зида и компоненту Rigidbody на играча. Али да би се решили непотребних калкулација које захтева коришћење физике, кретање ћемо ограничити на други начин, мало компликованији за имплементацију, али и даље прилично једноставан. Да би знали како ово да урадимо морамо се упознати са идејом кретања.

Потребно је да знамо правац у коме играч треба да се креће. Овај правац имамо у облику вектора, чије координате зависе од инпут-а корисника. Вектор се нормализује, и тако се губи његов интензитет те остаје само правац који се множи са жељеном брзином и добијамо померање кроз простор.

Након тога, ограничавамо позицију методом Clamp() у дозвољеним границама кретања.

### Умирање

Када је играч погођен непријатељским пројектилом, одузимају му се животни бодови, а када они спадну на 0 играч умире. Као индикатор судара, појављује се графички ефекат. Иста логика примењује се за непријатеље, с тим да је графички приказ другачији.

### Пуцање

Притиском на одређено дугме, инстанцира се нови објекат(нови објекат класе, и нови објекат у смислу графичког приказа) на сцену, коме се задаје(а овде сада користимо физику) правац(увек равно горе) и брзина, односно јачина силе која га покреће, јер сада користимо методу AddForce(). На исти начин је имплементирано и пуцање непријатеља, само у супротном смеру.

## Непријатељи

Основна идеја јесте да се непријатељи стварају у „таласима“ или налетима, где сваки следећи налет садржи једног непријатеља више од претходног. У неком моменту непријатеља ће бити много и скоро неизводљиво избећи све њихове пројектиле, и ту се игра завршава.

### Кретање

Кретање непријатеља разликује се од кретања играча, зато што овог пута немамо унос са тастатуре, па морамо да испрограмирамо логику кретања. Да би поједноставили ствари, једноставно генерисаћемо насумичне координате у неким границама, и сваки пут када непријатељ изврши кретање односно дође на задату позиицју, генерисаћемо нову.

# Детаљан преглед изворног кода

## Singleton pattern

На пар места у пројекту користимо Singleton. Користимо га онда када нам је потребна имитација статичке класе. Имитација, зато што нам није дозвољено да класа буде статичка ако наслеђује Monobehaviour, а ипак не желимо да радимо са објектима.

Да би ово извели, у нестатичкој класи правимо статичку променљиву која броји инстанце класе, односно не дозвољава да се прави нова инстанца ако једна већ постоји, па референцу на ту инстанцу додељујемо статичкој променљивој типа те класе. Ево како то изгледа:

Метода Awake() као што је пре напоменуто позива се у тренутку када се наша класа инстанцира.

## Class PlayerController

Ова класа садржи код који контролише играча, његово кретање, пуцање и умирање.

Пре свега, декларишемо неколико мањих класа које садрже информације о границама кретања, о животним боодвима, оружју...

Ове информације издвојене су у мање целине, односно класе ради лепшег приказа у Inspector-у. Па ево како изгледа класа кроз коју подешавамо границе:

Вредности уписујемо кроз едитор. Даље је потребно да вредности из ових класа доделимо променљивама у главној класи(PlayerController). То радимо на следећи начин:

Даље у главној петљи:

Учитавамо унос са тастатуре, и правимо вектор кретања. Након тога померамо брод са Translate() методом. Множимо са променљивом Time.deltaTime да нам брзина кретања не би зависила од фрејмова.

Ротирамо брод у односу на смер кретања, да би смо добили утисак скретања у страну.

Позивамо методу која испаљује ласер ако је регистрован одређени инпут.

И на крају, позивамо методу која ограничава кретање.



## Class IEnemy

Ова класа је интерфејс за непријатеље. Садржи само декларације али не и имплементације. Ако бисмо убацивали више типова непријатеља сваки би требао да наследи ову класу, и имплементира пре-декларисане методе.

Имамо неке променљиве неопходне за имплементирање понашања, и три методе које имплементирају одређено понашање.

Методе ApplyDamage() и Death(), могле су да се напишу као евент-методе, што би значило да се позивају када се одређене ствари догоде, односно када се промени вредност животних бодова. Али постоји једноставнији начин да се ово регулише, тако да пројектил сваки пут када се судари са непријатељем позива методу ApplyDamage() а она даље позива Death() ако је то потребно.



## Методе ApplyDamage() и Death()



Имамо интеракцију са класом GlobalStats, у њој чувамо тренутни скор, као и вредности које су фиксиране, као што је штета која је нанета играчу када бива погођен, као и колико штете играч наноси непријатељу.

*Напомена: Ове две методе могле су, (у овом случају би то и било боље) да буду статичке, али овако ипак остављамо простора за разлике у понашањима.*

## Class KerbanautInvader

Ово је једини тип непријатеља који имамо у игри, да не бисмо додатно компликовали ствари. Наслеђује интерфејс за непријатеље, и имплементира потребне методе.

Следећа метода ће генерисати координате нове позиције ка којој непријатељ треба да се креће када изврши претходно задато кретање.



Генеришемо координате у оквиру граница кретања. Кретање вршимо методом Lerp(), која лерпује, односно мења вредност од тренутне ка задатој у јединици времена.



У методи EnemyBehaviour() дефинишемо кретање, и онда ову методу позивамо из главне петље игре, односно методе Update(). Метода Lerp() прима три параметра, стару позицију, нову, и пређену путању процентуално(0.0-1.0). Е сада, овде имамо један проблем, а то је да брзина кретања неће бити константа, односно варираће у односу на дужину путање, што може бити оно што желимо да постигнемо, али ипак у већини случајева не бисмо то овако урадили. Свакако, ово је најједноставнији начин имплементације кретања, и у овој једноставној игри ће послужити.

На интернету се може наћи у много туторијала погрешно коришћење методе Lerp(), тако што уместо рачунања процента кретања, само проследимо променљиву Time.deltaTime, а уместо почетне позиције задајемо увек тренутну, ово ради, али проблем је то што наш објекат никада неће достићи 100% задате путање, и то нас спречава да проверимо да ли смо завршили кретање.

Начин на који је Lerp() имплементиран овде је мало комплкикованији, али је исправан. Идеја је, да имамо методу која ће да иницијализује лерповање, тако што закуца вредности, почетну и крајњу, и узме тренутно време Time.time, а то је укупно време од покретања игре и постави га као почетно. Даље настављамо да меримо време, и ако је разлика новог и почетног времена једнака јединици(1), значи да смо прешли 100% путање, па прекидамо лерповање, и позивамо поново методу која га иницијализује да бисмо добили нову позицију.



## Class Score

Помоћу ове класе управљамо скоровима. Имамо неколико метода, од којих две главне, једна која уписује скор и друга која га исписује на екран.

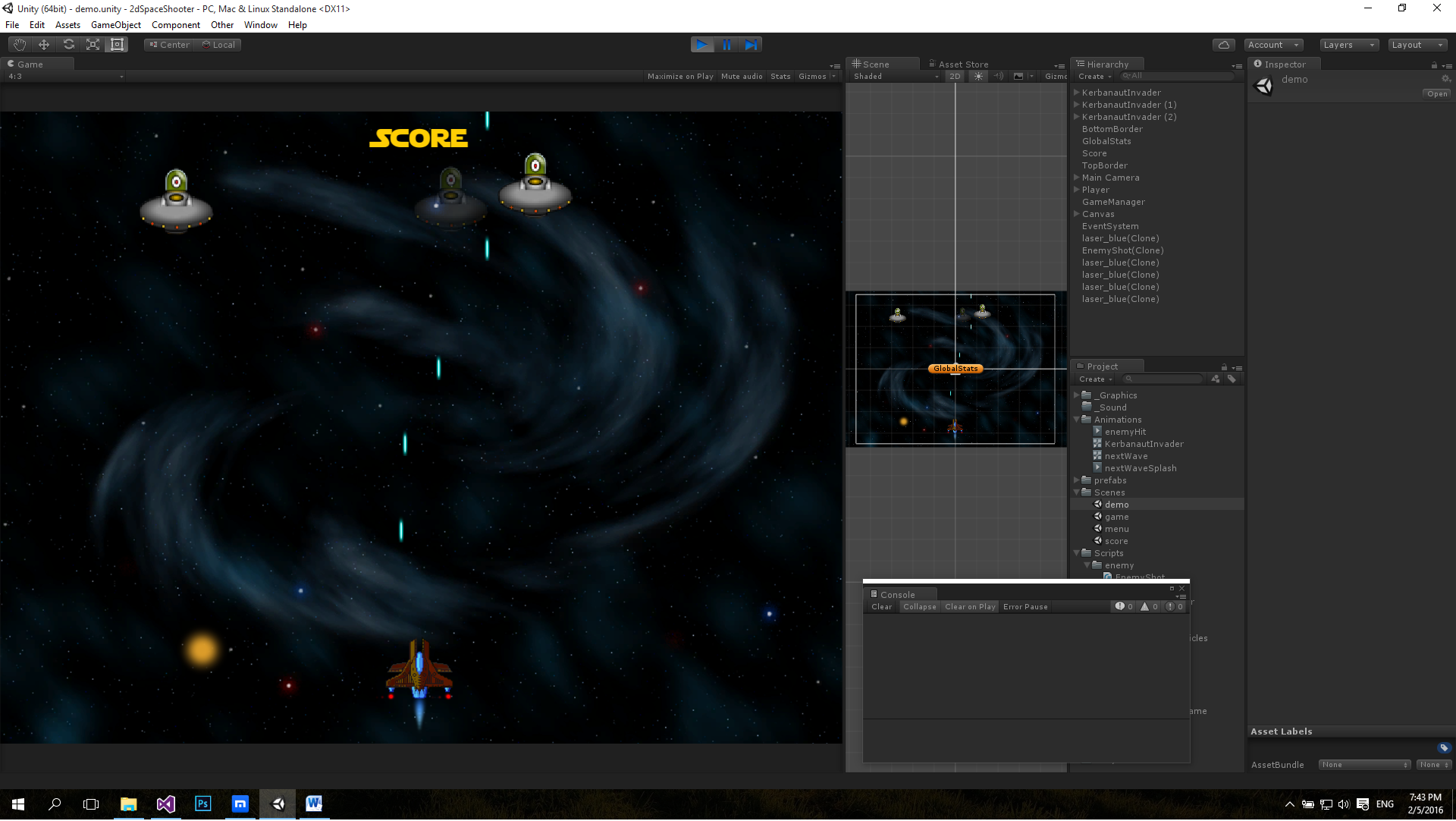
Хајде да прегледамо прво методу која омогуићава чување скора. Идеја је да прво пронађемо дотадашњи минимални скор, и заменимо га са тренутним, наравно, ако је тренутни већи од тадашњег минимума, а ако није не дирамо ништа. Из претходно поменутих PlayerPrefs-а, методом GetMinScoreIndex() која враћа стринг, односно назив променљиве из PlayerPrefs, добијамо њен назив, што нам омогућава да искористимо PlayerPrefs.GetString() методу да бисмо добили вредност променљиве са тим називом.

Након тога, правимо нови низ, који сортирамо, па уписујемо назад у PlayerPrefs. Ово можда делује компликованије него што је потребно, али овако мора, јер не можемо да чувамо низове у PlayerPrefs.

И на крају једностанво исписујемо скор на екран, тако што инстанцирамо 3D текст објекат на екран за сваки скор, и упишемо у њега вредности.

# Закључак

Добили смо солидан демо игре, који изгледа овако:



Unity је веома моћан алат, како за мале једноставне и брзе пројекте, тако и за оне веће. Успели смо да направимо игрив демо, релативно брзо и са не превише куцања.

На крају, ако хоћемо да ову игру билдујемо за неку од платформи, односно омогућимо коришћење ван едитора, довољно је да одемо на **File>Build**, изаберемо једну од многобројних платформи и кликнемо **Build**. За платформе попут Андроида и иОС-а потребне су мање измене у коду, и то једино око инпута корисника, што и није неки проблем, уместо да очитавамо са тастатуре, очитаваћемо инпут са телефона, односно помоћу accelerometer-a.

Алати коришћени:

[Unity3D](https://unity3d.com/)

[Visual Studio 2015 Community Edition](https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx)

[Visual Studio for Unity plugin](https://www.visualstudio.com/en-us/features/unitytools-vs.aspx)