Programiranje v okolju Unity 3D

Rok Kos

17. april 2016

Kazalo

Slike

Povzetek

V svoji projektni nalogi bom predstavil rokovanje z programskim okoljem Unity 3D. Kot primer bom vzel svojo aplikacijo, ki sem jo ustvaril z Unity-jem, ki predstavlja vodiča po Gimnaziji Vič.

Ključne besede: Unity3D

1 SET UP

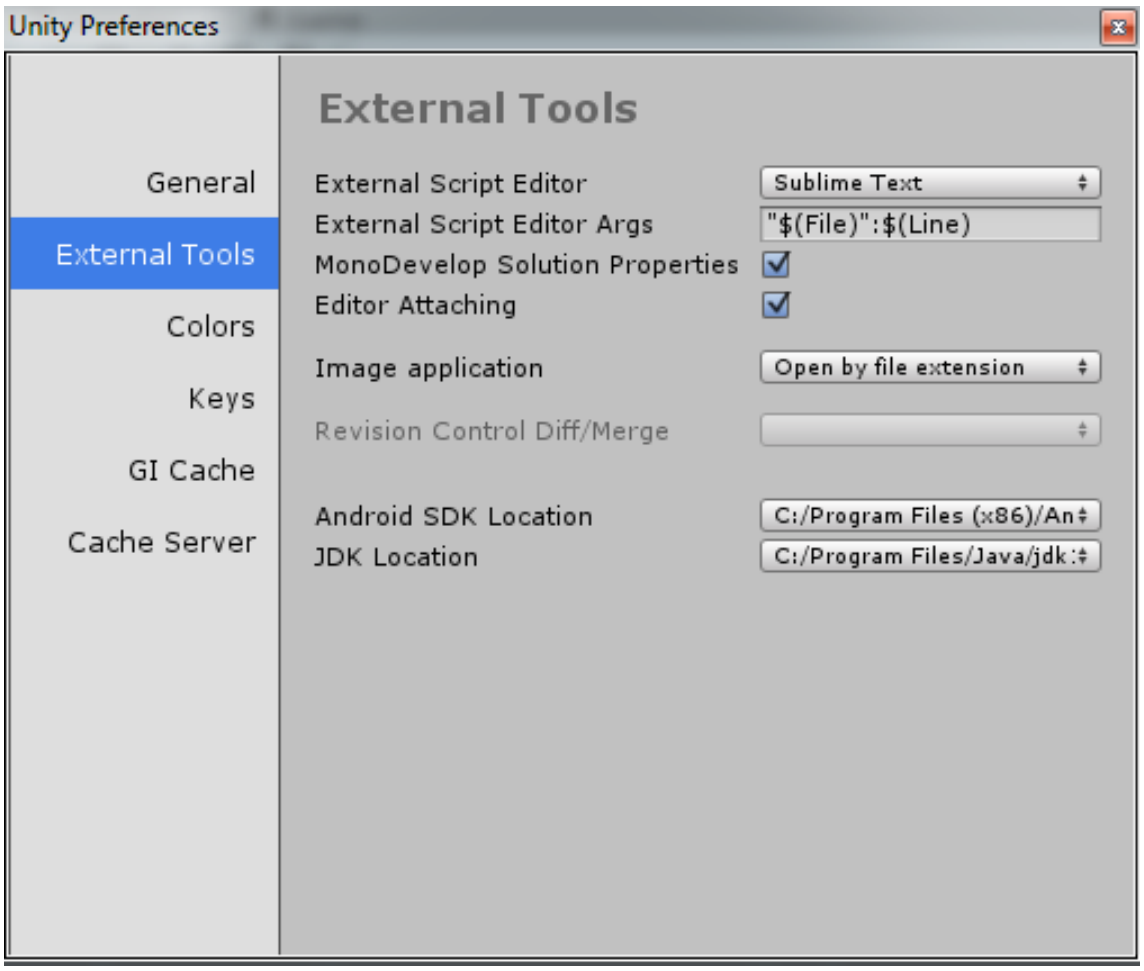
1.1 Editor

Kot vsako razvojno okolje ima tudi Unity 3D svoj privzeti editor. To je MonoDevelop, ki podpira večplatformski razvoj (to pomeni, da lahko isto kodo pišemo za različne operacijske sisteme kot so Linux, Windows, Mac, Android, IOS, PS3 itd.). Podpira naslednje jezika:

- C#
- F#
- VisualBasic
- . . .

Ima tudi integrirano dopolnjevanje kode (code auto-completion) in svoj razhroščevalec (debugger). Sam sem kar nekaj časa uporabljal MonoDevelop in v njemu razvijal, ampak mi je pri njem vedno nekaj manjkalo. Zdelo se mi je, da je njihov dopolnjevalec kode (code auto-completion) vsiljeval svoje stvari in da mi osvetljevalec (highliter) ni vedno obarval kakšnih klasov. Zato sem se odločil, da bom začel uporabljati meni ljubši urejevalnik besedila, to je Sublime Text 3. Pri tem urejevalniku sem dobil veliko svobodo pri urejanju, iskanju, popravljanju ter pri samem urejanju urejevalnika. To Sublime omogoča s svojim Package Controlom, ki se ga da inštalirati preko te strani: https://packagecontrol.io/installation. Preko Package Controla lahko namestimo dodatne vtičnike (plugine) in snippete za Sublime, ki izboljšajo samo delovanje urejevalnika. O tej temi bi se dalo še veliko govoriti zato jo bom pustil za kdaj drugič. Seveda moramo Unity-ju povedati, da naj svoje datoteke odpira z sublimom. To naredimo tako:

1. Naložimo Sublime Text 3/2 preko te http://www.sublimetext.com/3
2. Gremo v **Edit → Preferences → External Tools**(kot lahko vidimo v spodnji sliki) in spremenimo na[urejevalnik
3. Spremenimo **External Script Editor Args** v "$(File)":$(Line) zato, da bo Sublime skočil v vrstico, kjer je error
4. Sedaj bi nam moral Unity odpreti Sublime, ko dvokliknemo na datoteko, ki je skripta
5. V Sublimu bomo še spremenili, katere datoteke hočemo videti v projeknem drevesu. To naredimo tako, da gremo v **Project → Save Project As** in notri vtipkamo tole kodo:



Slika 1: Nastavitev Sublime Text 3

```
{
        "folders":
        [
                {
                        "path": "Assets/Scripts",
                        "file_exclude_patterns":
                        [
                                "*.dll",
                                "*.meta"
                        ]
                }
        ]
}
```

in potem shranimo datoteko. To nam skrije vse nepotrebne .meta in .dll datoteke

6. S Package Controlom na koncu še namestimo dodatne snippete, ki nam pomagajo pri dopolnjevanje kode (auto-completion) in barvanju kode. Vse potrebne package najdemo na tej strani http://wiki.unity3d.com/index.php/Using_Sublime_Text_as_a_script_editor, kjer so tudi bolj podrobna navodila za celoten namestitev Sublime Text-a.

Kot zanimivost pa bi vam rad pokazal, kako enostavno se da narediti snippete v Sublime Text-u, ki se potem sprožijo ob določenem zaporedju tipk in nakoncu tabulatorja.

```
<snippet>
        <content><![CDATA[
//Author: Rok Kos <rocki5555@gmail.com>
//File: $TM_FILENAME
//File path: $TM_FILEPATH
//Date: $1
//Description: $2
]]>
        </content>
        <!-- Optional: Set a tabTrigger to define how to trigger the snippet -->
        <tabTrigger>sig</tabTrigger>
        <!-- Optional: Set a scope to limit where the snippet will trigger -->
        <!-- <scope>source.python</scope> -->
</snippet>
```

1.2 Plugini

V Unity-ju si lahko pomagamo z različni skriptami, ki jih ustvarimo izven Unity-ja. Tem skriptam pravimo vtičniki (plugins). Te nam lahko pomagajo pri samem urejanju projekta, olajševanju in avtomatiziranju nekaterih stvari ali pa nam dodeljuje kakšne funkcije za prav posebno platformo. Prvim pravimo Managed plugins, drugim pa Native plugins [Tec]. Managed plugine ponavadi pišemo v C#, saj uporavljajo samo knjižnico .NET. Te vtičniki se prevedejo v dinamične knjižnice (dynamical linked library) oz. DLL, ki jih potem vključimo v projekt. Native plugini pa so napisani v C, C++, Objective-C in ostalih jezikih. To pomeni, da damo možnost Unity, da lahko npr. kliče kodo Java ali C++. Seveda je glavna prednost tega, da napišemo svoje knjižnice za določeno platformo npr. za IOS svojo in za Android svojo.

2 SHADERS

2.1 Definicija

Shaderji so skripte, ki nam povejo, kako naj se vsak piksel na zaslonu zrendera. To seveda ni samo od tega kako so napisani shaderji ampak tudi od tega kakšne materiale in teksture uporabljamo. V shaderjih so zapisani algoritmi, ki uporabljajo vektorje za svetlost in barvo in s tem povejo kako naj se obarva piksel. Z verzijo Unity 5 je prišel ven tudi njihov Standard Shader, ki je zelo uporaben in se lahko zelo prilagaja. Ampak v tem projektu bom predstavil, kako napisati lasten shader.

2.2 Lastnosti shaderjev

Tole je prikaz enostavnega shaderja [Jos]:

```
Shader "DT\Basic\SimpleShader"{
        SubSHader{
                Tags = {"RenderType" = Opaque}
                CGPROGRAM
                #pragma surface surf Lambert
                struct Input{
                        //(1.0, 1.0, 1.0, 1.0) R, G, B, A
                        float4 color : COLOR;
                };
                void surf(Input IN, inout SurfaceOutput o){
                        o.Albedo = 1;
                }
                ENDCG
        }
        Fallback "Diffuse"
}
```

3 CLASSES

3.1 Objektno programiranje

Kot nam že samo ime pove temelji objektno programiranje na objektih, ki programiranje bolj približajo človeku. Na tem principu deluje tudi Unity s svojim jezikom C#. V vsaki igri kot tudi v realnem svetu imamo stvari s podobnimi lastnostmi ali pa celo z enakimi atributi. Tukaj v igro vstopijo objekti. Ti nam pomagajo specificirati atribute določenega predmeta. Vzemimo npr. svetilko. Svetilka ima žarnico, baterijo in stikalo za vklop in izklop. Pozna tudi metodo vklop in izklop. Lahko bi vzeli tudi drugi klas npr. zrcalo, ki bi spet imelo drugačne atribute. Ampak glavna ideja za tem je, da imamo lahko sedaj v naši igri poljubno mnogo instanc teh svetilk in zrcal in zato nismo rabili pisati code za vsakega posebej. Lahko bi rekli, da imamo glavni klas in iz njega samo kopiramo ven nove objekte. Pri klasih pa pridemo tudi do dedovanja, dostopnosti (public, protected, private) in polimorfizma.

3.2 Pisanje classov

Primer mojega klasa, ki sem ga uporabil v svoji aplikaciji in predstavlja šolo.

```
//Author: Rok Kos
//Date: 05.12.2015
//Description: Class for classroms

using UnityEngine;
using System.Collections;

public class Sola : MonoBehaviour {

        public class Ucilnica{
```

```
11                  /*
12                   * To je class v katerem bomo definirali
    ↪ pozicijo vsake ucilnice,
13                   * ime ucilnice, nfc tag ucilnice.
14                  */
15                  public Vector3 pozicija;
16                  public Quaternion rotacija;
17                  public string ime_Ucilnice;
18                  public int NFC_tag;
19
20                  //Construktor, to so default vresnosti
21                  public Ucilnica(){
22
23                          pozicija =  new Vector3(0f,0f,0f);
24                          rotacija = new Quaternion(0f,0f,0f,0f);
25                          ime_Ucilnice = "UNKNOWN";
26                          NFC_tag = -1;
27                  }
28                  //Construktor z vsem
29                  public Ucilnica(Vector3
    ↪ vnesena_pozicija,Quaternion vnesena_rotacija,
30                          string vneseno_ime_Ucilnica, int
    ↪ vnesen_NFC_tag){
31                          pozicija = vnesena_pozicija;
32                          rotacija = vnesena_rotacija;
33                          ime_Ucilnice = vneseno_ime_Ucilnica;
34                          NFC_tag = vnesen_NFC_tag;
35                  }
36                  //Construktor brez rotacije
37                  public Ucilnica(Vector3 vnesena_pozicija,
38                          string vneseno_ime_Ucilnica,
39                          int vnesen_NFC_tag){
40                          pozicija = vnesena_pozicija;
41                          rotacija = new Quaternion(0f,0f,0f,0f);
42                          ime_Ucilnice = vneseno_ime_Ucilnica;
43                          NFC_tag = vnesen_NFC_tag;
44                  }
45
46                  public Vector3 vrni_pozicijo(){
47                          return pozicija;
48                  }
49
50                  public Quaternion vrni_rotacijo(){
51                          return rotacija;
52                  }
53
54                  public string vrni_ime(){
55                          return ime_Ucilnice;
56                  }
57
58                  public string vrni_class(){
```

```
59                          return ime_Ucilnice +
    ↪ pozicija.ToString ("G4") + "\n";
60                  }
61                  //Destructor
62                  ~Ucilnica(){
63
64                  }
65          }
66  }
```

4 PREFABS

4.1 Definicija

Prefab si v Unity okolju lahko predstavljamo, kot klas GameObject-a, na katerega so pripete skripte, teksture, slike, rigidbody-ji itd. To nam da možnost, da shranimo celotne GameObject-e, ki jih lahko kasneje uporabimo v sceni oz. naredimo poljubno število instanc. Za boljšo pojasnitev lahko vzamemo primer drevesa. Najprej naredimo eno drevo z vsemi atributi ga shranimo in nato lahko delamo iz tega drevesa nove instance v sceni in s tem dobimo efekt gozda. To lahko naredimo tudi med izvajanjem same igre, kar nam da še večje možnosti. Enako lahko naredimo za nasprotnike, zidove, zgradbe itd.

4.2 Uporaba

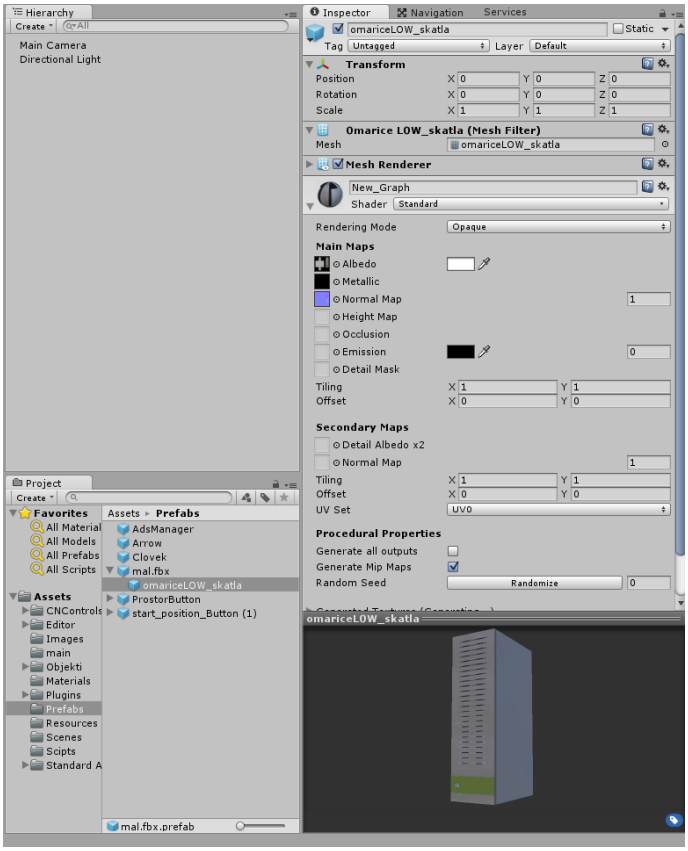
Ena izmed najbolj uporabnih lastnosti prefabov je, da če spremenimo en element in potem te sprembe shranimo se spremenijo vsi objekti. Z enim gumbom lahko naredimo globalno spremembo. To naredimo tako, da gremo na prefab v sceni ali v mapi, kjer se nahaja, ter v pregledovalniku (inspektorju) zgoraj pod imenom, kjer vidimo 3 možnosti: Select, Revert, Apply. Naredijo ravno to. Revert skoči na prejšno globalno spremembo, apply pa naredi gloabno spremembo. Prefabe se da zelo enostavno instancirati med igro. Kot spodaj vidimo. Narejenim objektom lahko tudi spreminjamo njihove lastnosti med igro.

5 IMPORTING

5.1 Lokacija in tipi datotek

Unity pokriva velik spekter različnih formatov iz različnih programov, kot so npr. Blender, Maya, Photoshop itd. Vse datote se nahajajo v Asset mapi, ki je potem tudi vidna v samem Unity urejevalniku. V spodnji sliki lahko tudi vidimo tipično strukturo map, ko se naredi nov Unity projekt.

Pri tipičnih formatih datotek kot so npr. JPG, PSD, PNG, MA, BLED, FBX, WAW itd. lahko datoteko, kar kopiramo v Asset mapo ali jo celo povlečemo v prav specifično mapo, kjer jo hočemo in se bo prikazala v Unity urejevalniku. Tako so potem urejene datoteke v samem Unity-ju.



Slika 2: Prefab

Ko mi kopiramo datoteko v Asset mapo, Unity poskrbi, da glede na končnico oz. tip se pravilno vključi v projekt, kompresira in nakoncu postane tudi presentativna v urejevalniku. Pri tem lahko tudi sami spreminjamo določene parametre:

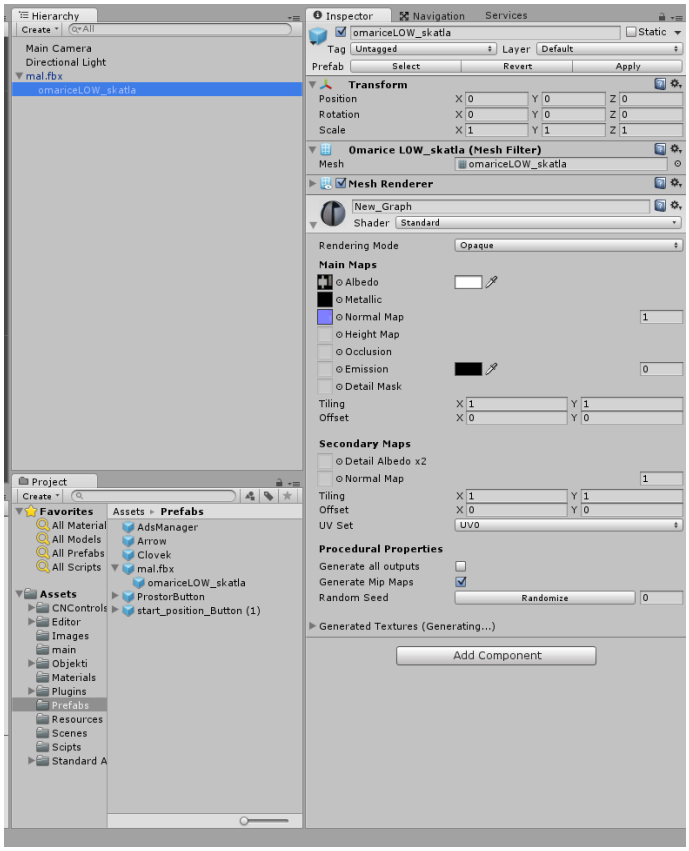
- kakšnega tipa je datoteka (sprite, tekstura, normal map itd.)
- kako gosto naj bojo piksli (če je slika)
- kaliko prostora lahko zasede na disku
- kako naj se kompresira

Nakoncu je ponujena možnost, da za različno platformo (Android, IOS, PC, PS3) različno nastavimo import nastavitve. Spodaj lahko na primeru vidimo kakšne opcije so nam na voljo.

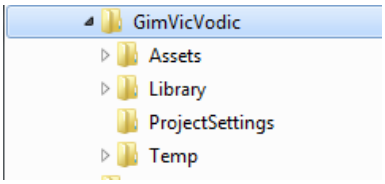
5.2 Asset store

To je trgovina, kjer so brezplačni in plačljivi asseti narejeni s strani Unity Technologies in velike skupnosti razvijalcev. Pod te assete spadajo vse od tekstur, 3D modelov, slik, projektov itd. Po trgovini se lahko zelo lahko navigira in išče za prav specifične assete.

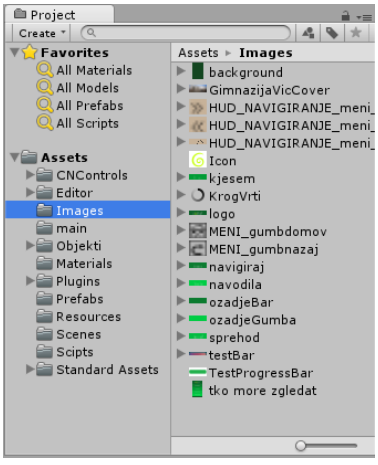
Vsak asset ima potem tudi opis kaj vsebuje, ceno, kakšne slike in mogoče tudi posnetek implementacije ali primer projekta.



Slika 3: Prefab spremembe

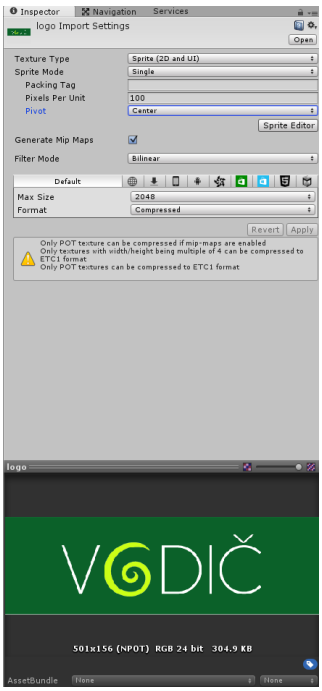


Slika 4: Struktura map

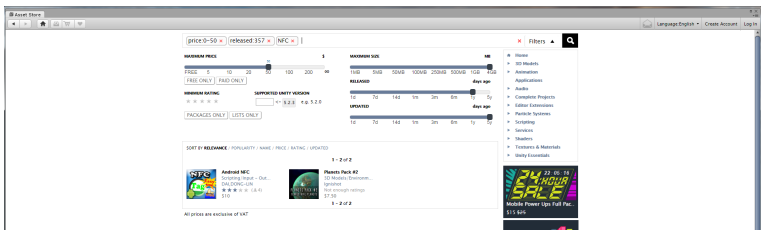


Slika 5: Pregled map v Unity-ju

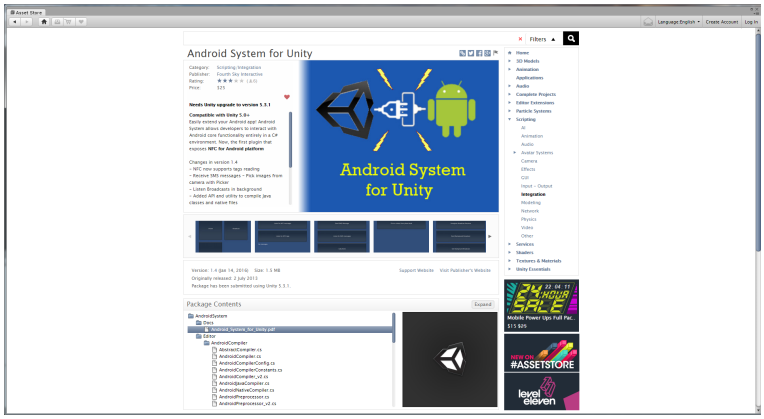
Glavna prednost je da se zelo hitro vključijo v trenutni projekt in za njih ni potrebno spreminjati nastavitev. Razvijalec lahko tudi izbira katere stvari



Slika 6: Import nastavitve



Slika 7: Iskanje assetov

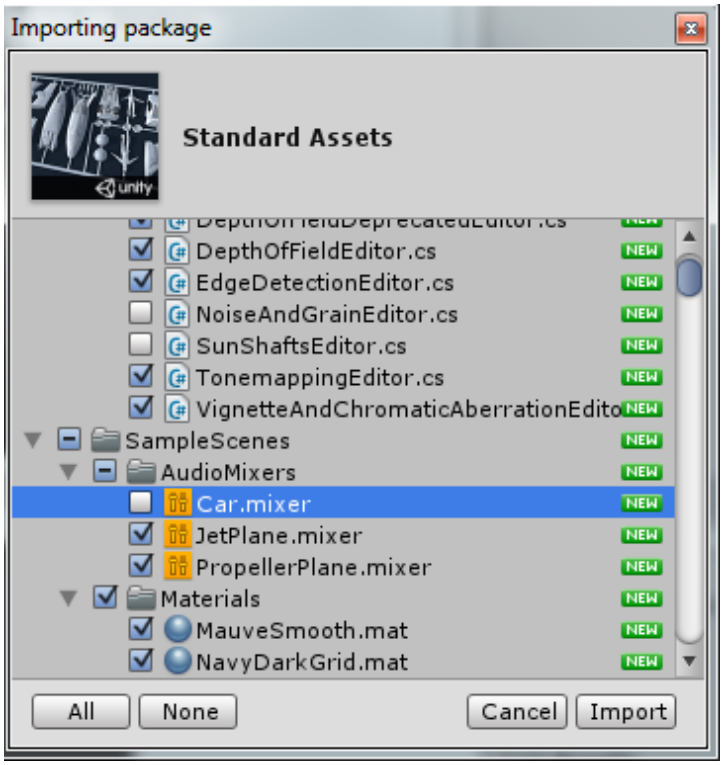


Slika 8: Predstavitev asseta

hoče prenesti v projekt in katere ne.

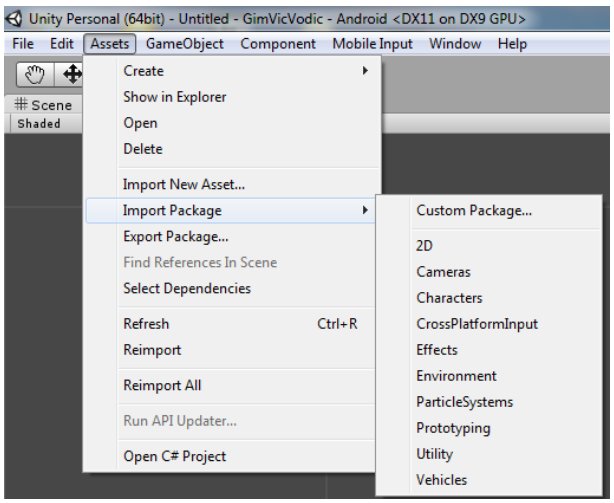
5.3 Standard Assets

To je velika skupina assetov, ki je spravljena v en paket. Unity je naredil ta paket, da lahko začetniki hitro in brez večjega truda naredijo enostavno



Slika 9: Izbira kaj vključiti

aplikacijo. V tem paketu je tudi veliko primerov uporabe teh assetov. Zelo olajšajo kakšne stvari, za katere bi enemu samemu razvijalcu vzelo veliko časa, da bi jih implementiral. V mojem projektu sem tudi uporabila Joystick implementacijo iz standard assetov. To sem naredil tako, da sem šel **Assets → Import Package** in izbral področje katerega sem želel (v mojem primeru je to bilo CrossPlatformInput).
In nato se odpre okno, ki ga lahko vidimo tudi na sliki 9.



Slika 10: Standard Assets

Je pa tudi velika past saj z vsako novo verzijo Unity-ja pridejo novo standard asseti, ki se lahko obnašajo drugače, kot prejšnji zato moramo vedno pogledati za kakšno verzijo, so bili izdani asseti.

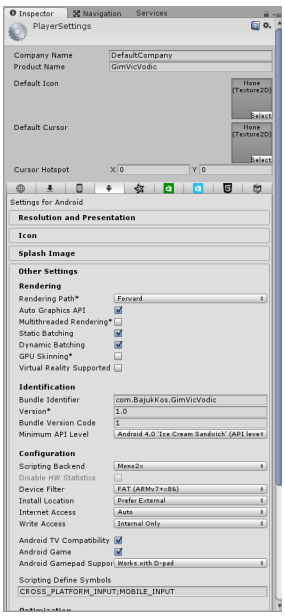
6 BUILDING PROJECT

6.1 Nastavitve

Kot smo že omenili v uvodu je Unity večplatformsko razvijalno orodje. Z Unity-jem lahko razvijamo aplikacije za:

- Web Player
- Windows
- Mac OS X
- Linux
- IOS
- Android
- Tizen
- Windows Store
- Windows Phone 8
- WebGL
- Samsung TV
- BlackBerry
- Xbox 360
- Xbox One
- PS3
- PS Vita
- PS4

Seveda za nekatere platforme potrebujemo posebne licence ali Unity PRO verzijo ter seveda za posamezne platforme posebne SDK knjižnice.
Do vseh te nastavitev pa dostopamo tako, da odpremo **File → Build Settings...**. Nato se nam pojavi to okno:
Na sredini nam pokaže imena scen, ki se bojo zgradila v tej verziji. Imamo možnost, da dodamo trenutno sceno ali pa povlečemo sceno iz mape. Z lahkoto pa tudi odstranimo sceno tako, da jo odznačimo. Na levi strani vidimo katere platforme so nam na voljo. Levo spodaj je gumb **Switch Platform**, ki zamenja platformo za katero hočemo buildati. Zraven je gumb **Player Settings...**, ki po samem imenu malce zavaja. V bistvu nam odpre novo okno, kjer lahko za vsako platformo specificiramo na kakšen način se bo zgradila aplikacija. Imamo različne nastavitve od imena aplikacije, izbira ikone, resolucije, renderinga in tudi optimizacije. Več o



Slika 11: Build settings

Slika 12: Build settings

tem bo povedal v naslednjem poglavju, ko se bom tudi osredotočil samo na android platformo.

Ostaneta nam samo še dva gumba, ki pa naredita točno to kar piše na njih. To sta **Build**, ki zgradi aplikacijo(naredi npr. .exe ali .apk datoteko) in **Build and Run**, ki jo še požene. In če izbrana platforma računalnik potem se zažene na računalniku, če gre za mobilno platformo, pa mora biti na računalnik povezan telefon in na računalniku mora biti SDK knjižnica, ko so izpolnjene vse te specifikacije se na telefonu namesti apk od aplikacije in se tudi požene.

6.2 Android

Preden lahko zbuildamo kateri koli projekt za android moramo najprej izpolniti nekaj korakov preden lahko dejansko vidimo aplikacijo na telefonu.

1. Prenesemo Android SDK knjižnico s tega naslova: http://developer.android.com/sdk/index.html

2. Namestimo knjižnico SDK in gledamo, da smo namestili saj API level 4.0 ali več kot to tako, da bojo naše aplikacije kompatibilne z večino telefonov.

3. Nastavimo pot do SDK knjižnice v Unity-ju, tako da lahko Unity dostopa do nje med buildanjem aplikacije. To naredimo tako, da gremo **Edit → Preferences → External Tools** in dobimo enako okno kot na sliki 1. Če tega ne naredimo se nam to okno pojavi, ko poskušamo prvič zbuildati aplikacijo za android.

4. Povežemo telefon z računalnikom. Namestimo potrebne driverje ter omogočimo USB Debugging (to je ključno saj drugače ne bomo mogli dostopati do telefona). To omogočimo tako, da gremo v **Settings → Developer options** in tam omogočimo to opcijo.

6.3 Unity Remote 4

Ker je konstantno grajenje in nameščanje aplikacij zamudno in počasno opravilo, so pri Unity-ju naredili aplikacijo Unity Remote 4. To je zelo uporabna in skoraj nujno potrebna aplikacija za razvijanje na telefonih, kar prihrani ure in ure čakanja. Ta aplikacija se dobi na Google Play Storu in App Storu. Pomaga pri simuliranju:

- Dotika
- Accelerometer-a
- Gyroscopa
- Kamere
- Kompasa
- GPS-a

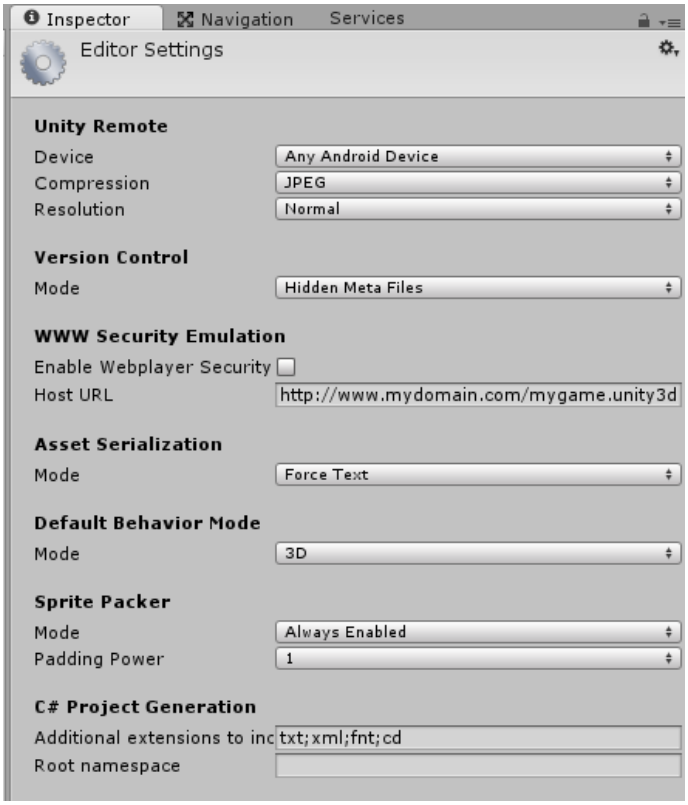
Aplikacija deluje tako, da se preko USB kabla poveže z računalnikom in ko gremo v play mode v Unity urejevalniku pošilja sliko na zaslonu na telefonu. Nazaj pa na računalnik pošilja podatke o zgoraj navedenih funkcijah. Deluje kot nek emulator za telefone. Seveda pa ni še čisto popolna, včasih ima težave z povezavo in resolucija, ki je prikazana na telefonu ni dejanska končna resolucija. Vse razen tega pa dela zelo dobro.

7 VERSION CONTROL

7.1 Integracija v Unity-ju

Unity ponuja že v samem editorju dva različna version control sistema. To sta Perforce(https://www.perforce.com/) in Plastic SCM(https://www.plasticscm.com/). Sedaj se pojavi tudi vprašanje zakaj bi uporabljali kateri koli version control sistem. Ena izmed ključnih lastnosti je, da lahko spremljamo potek sprememb in da se lahko kadarkoli v procesu vrnemo na prejšnjo rešitev. Da nam pa tudi funkcije, kot npr. kdo v ekipi je naredil spremembo, kakšne so spremembe med samimi spremembami (commit-i) in lažje vidimo kje so nastale napake ali hrošči. V moji seminarski nalogi ne bom uporabljal teh dveh orodji, ampak bom uporabljal čisto osnovni GIT, kot stran oz. server kjer bo držal bo uporabil GitHub(https://github.com/). Najprej bomo seveda nastavili Unity tako, da bomo lahko primerjali datoteke med seboj. To naredimo tako da **Edit → Project Settings → Editor menu**. Tukaj nastavimo, da se vse datoteke nujno spremenijo v .txt datoteke.

Saj ima Unity po primarnih (default) nastavitvah nekatere objekte zapi-



Slika 13: Namestitev version control

sane v binarnem sistemu, pri katerih je skoraj nemogoče spremljati spremembe.

7.2 GIT

Git je brezplačni in odprtokodni version control sistem, ki ima veliko uporabnih funkcij. Navedel bom samo nekatere, ki so najbolj pogoste za uporabo. Vse funkcije imajo pred ukazom besedo git. Funkcije:

- **init** : Naredi prazen git mapo ali pa na novo reinicializira že obstoječo mapo.
  Komanda bi zgledala nekako takole:
  ```
  $ git init
  ```
- **clone** : Klonira že obstoječo mapo v trenutno mapo
  Najbolj pogosto se ta komanda uporablja, da kloniramo mapo ali projekt iz kakšne strani, na naš računalnik. Lahko pa tudi lokalno pri sebi kloniramo iz ene mapo v drugo.
  Za kloniranje map iz url naslova git podpira ssh, git, http in https. Url naslov mora biti v eni izmed takih oblik:
  - ssh://[user@]host.xz[:port]/path/to/repo.git/
  - git://host.xz[:port]/path/to/repo.git/
  - http[s]://host.xz[:port]/path/to/repo.git/

  In ko vse skupaj sestavimo bi bil primer komande takšen:
  ```
  $ git clone
    ↪  https://RokKos@bitbucket.org/RokKos/gimvicvodic.git
  ```
- **config** : Nastavi globalne ali lokalne spremeljivke mape.
  Ne bom se poglabljav v vse možnosti uporabe te komande, ampak bom naštel samo nekaj osnovnih. V bistvu bosta nas zanimali samo dve in to sta:
  ```
  $ git config --global user.name "Rok Kos"
  $ git config --global user.email RokKos@gmail.com
  ```
- **add** : Doda datoteko ali skupino datotek k delovnemu drevesu.
  Ta komanda doda datoteko, tako da se upoštevajo in beležijo vse spremembe narejene na njej. Na začetku, ko inicializiramo mapo, hočemo da se upoštevajo vse, nato pa dodajamo posamezne datoteke.
  Par primerov uporabe te komande:
  ```
  $ git add --all
  $ git add Documentation/\*.txt
  $ git add Assets/ControllerScript.cs
  ```
- **commit** : Zapiše spremembe v mapi
  To komando pokličemo, po komandi add ali ko združujemo skupaj dve veji. Poleg tega imamo možnost, da poleg commita dodamo tudi

sporočilo oz. poročilo kaj smo naredili. Primer:

```
$ git commit -m "Meseage of commit"
```

- **push** : Naloži datoteke na oddaljen strežnik
  To komando pokličemo čisto nakoncu, ko smo dodali vse datoteke, jih zapisali (commitali). Imamo pa tudi možnost, da pushamo na različne veje(branche).
  Primer:

```
$ git push origin master
```

- **pull** : Potegne datoteke iz oddaljenega strežnika in jih združi z lokalno mapo.
  Ta komanda požene git fetch komando, ki dobi podatke iz strežnika in potem za tem požene git merge, ki združi lokalno verzijo z verzijo iz oddaljenega strežnika. Primer:

```
$ git pull origin
```

- itd.

[Tec] [Ala14] [Tho15]

Viri in literatura

[Ala14] THORN Alan. *Pro Unity Game Development with C#*. Vol. 1. izdaja. New York: Apress, 2014. ISBN: 9781430267461.

[Tho15] FINNEGAN Thomas. *Learning Unity Android Game Development*. Vol. 1. izdaja. Birmingham: Packt Publishing, 2015. ISBN: 9781784394691.

[Jos] KINNEY Joshua. *Introduction to Scripting Shaders in Unity*. URL: http://www.digitaltutors.com/tutorial/1438-Introduction-to-Scripting-Shaders-in-Unity (visited on 2015).

[Tec] Unity Technologies. *Unity Manual*. URL: http://docs.unity3d.com/Manual/index.html (visited on 2015).