Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Toni Mervar

Izdelava 3D igre z uporabo pogona Unity za platformo Android

diplomsko delo na univerzitetnem študiju

Mentor: dr. Saša Divjak

Ljubljana, 2013

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01869/2012

Datum: 05.09.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **TONI MERVAR**

Naslov: **IZDELAVA 3D IGRE Z UPORABO POGONA UNITY ZA PLATFORMO ANDROID**

3D GAME DEVELOPMENT USING THE UNITY ENGINE FOR ANDROID PLATFORM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Proučite postopek izdelave 3D igre za platformo Android z uporabo pogona Unity. Opišite osnovne koncepte 3D grafike in modeliranja, s katerimi se srečamo pri pri izdelovanju iger. Podajte ozadje in primerjavo nekaterih fizikalnih pogovov, ki pridejo v poštev. Predstavite platformo Android in pogon Unity, predvsem njegove značilnosti in funkcionalnosti, ki nam jih ponuja. Delo naj dopolnjuje praktičen primer razvoja in preskusa 3D igre za pametni telefon z operacijskim sistemom Android.

Mentor:

prof. dr. Saša Divjak

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Toni Mervar, z vpisno številko **63070207**, sem avtor diplomskega dela z naslovom:

Izdelava 3D igre z uporabo pogona Unity za platformo Android

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Saše Divjaka,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 6. marca 2013 Podpis avtorja:

Zahvalujem se mentorju prof. dr. Saši Divjaku za vso izkazano pomoč ob nastajanju tega diplomskega dela. Posebna zahvala gre tudi moji družini, ki me je vsa leta spodbujala in mi stala ob strani.

Svojim dragim staršem.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Osnovni koncepti 3D grafike **3**

2.1 Vektorji 3

2.2 Modeliranje 6

2.3 Osvetljevanje 8

3 Fizikalni pogoni **11**

3.1 PhysX 13

3.2 Havok 14

3.3 Bullet 14

4 Izdelava 3D igre **17**

5 Android **21**

5.1 Android pogoni za igre 25

6 O pogonu za igre Unity **27**

7 Izdelava iger s pogonom Unity **31**

7.1 Programiranje v Unity 32

7.2 Osvetljevanje 33

7.3 Grafični uporabniški vmesnik 36
7.4 Senčniki (*shaders*) 37

8 Lastno delo **41**
8.1 Vodni tobogan 43
8.2 Igralec 44
8.3 Cekini 46
8.4 Voda 47
8.5 Hobotnica 47
8.6 Testiranje 49

9 Sklepne ugotovitve **51**

Povzetek

Cilj diplomske naloge je spoznati postopek izdelave 3D igre za platformo Android z uporabo pogona Unity. V prvem delu naloge so opisani določeni osnovni koncepti, ki nam pridejo prav pri izdelovanju iger, saj se z njimi srečamo pri izdelovanju skoraj vseh iger. To so vektorji in določene operacije nad njimi, 3D modeliranje, s katerim izdelamo večino objektov, ki jih vidimo v igri, in načini osvetljevanja ter objektov. Nato spregovorimo o fizikalnih pogonih in zaznavanju trkov ter predstavimo tri popularnejše fizikalne pogone. V naslednjem delu diplomske naloge spregovorimo o načinih izdelovanja iger in podamo nekaj splošnih informacij o pogonih za igre. Temu delu sledi predstavitev platforme Android. Tukaj povemo kakšna je njena arhitektura in podamo nekaj uporabnih informacij ter spregovorimo nekaj besed o pogonih za igre, s katerimi lahko izdelujemo igre za to platformo. Sledeče poglavje podaja informacije o pogonu za igre Unity, predvsem njegove značilnosti in funkcionalnosti, ki nam jih ponuja. V naslednjem poglavju ponovno spregovorimo o pogonu Unity, tokrat o tem, kako poteka izdelovanje iger z uporabo tega pogona. Zadnji del diplomske naloge pa je praktičen del. Tukaj opišemo, kako smo izdelali našo 3D igro, ki smo jo nato tudi testirali na Androidnem pametnem telefonu.

Ključne besede: Unity, Android, 3D igre, pogon za igre, izdelava iger

Abstract

The aim of this thesis is to get to know the process of making a 3D game for the Android platform using the Unity game engine. In the first part of this thesis we describe some basic concepts, which come in handy when making games, as we come across them when making almost all games. These are vectors and certain operations over them, 3D modeling, which is used to make the majority of objects, we see in a game, and ways of lighting these objects. Then we talk about physics engines, collision detection and introduce three popular physics engines. In the next part of the thesis we talk about different ways of making games and give some general information about game engines. After this part we represent the Android platform, here we talk about its architecture, give some useful information and talk a few words about game engines, which can be used to make games for this platform. The following chapter gives information about the Unity game engine, especially its characteristics and functions it provides. In the next chapter we again talk about the Unity game engine, this time about the process involved when making games using this game engine. The last chapter of the thesis is dedicated to our work in practice. In it we describe how we made our 3D game, which we tested on an Android smart phone.

Keywords: Unity, Android, 3D games, game engine, game making

Poglavje 1

Uvod

S prihodom pametnih telefonov se je igričastvo preselilo tudi na te naprave. Dobili smo igre, ki so bile narejene posebej za delovanje na pametnih telefonih in so pri tem izkoriščale specifičnosti teh naprav. Posebnost teh iger je njihovo upravljanje. Medtem ko igre na konzolah upravljamo s priloženimi igralnimi ploščki, igre na običajnih računalnikih pa z uporabo tipkovnice in miške, igre na pametnih telefonih običajno upravljamo z dotiki zaslona, ki znajo zaznavati dotike, ali pa z nagibanjem same naprave, saj imajo vgrajene merilce, ki zaznavajo premikanje naprave.

Trg z igrami za pametne telefone se v marsičem razlikuje od trga z igrami na konzolah in računalniku. Sredstva potrebna za izdelavo iger na pametnih napravah so dosti manjša. To velja tako za denarna sredstva kot za razvojno ekipo. Igre za pametne telefone lahko izdelujejo ljudje z vsega sveta, v razvojni ekipi je lahko tudi samo ena oseba z osebnim računalnikom, ustreznim znanjem in programsko opremo. Tudi distribucija iger je pri igrah za pametne telefone enostavnejša, saj se skoraj vse igre in aplikacije prodajajo preko uradnih spletnih trgovin posameznih platform (npr. Google Play za platformo Android in App Store za platformo iPhone), kjer lahko svoje igre objavlja praktično vsak. Edina slabost je ta, da si te trgovine vzamejo določen odstotek od vsake prodane igre.

Z večanjem popularnosti iger na pametnih telefonih je prišlo tudi do

večanja števila orodij, s katerimi lahko te igre izdelujemo. Pojavljati so se pričeli pogoni za igre, s katerimi lahko izdelujemo igre za pametne telefone. Eden izmed teh je tudi pogon za igre Unity, ki med drugim omogoča izdelovanje iger za pametne telefone na platformi Android. Današnji pametni telefoni so dovolj zmogljivi, da lahko na njih poganjamo 2D igre in tudi preprostejše 3D igre. Tem zadnjim se bomo posvetili v tem diplomskem delu. Seznanili in preizkusili se bomo v izdelavi 3D igre za mobilni telefon na platformi Android z uporabo pogona Unity.

Poglavje 2

Osnovni koncepti 3D grafike

2.1 Vektorji

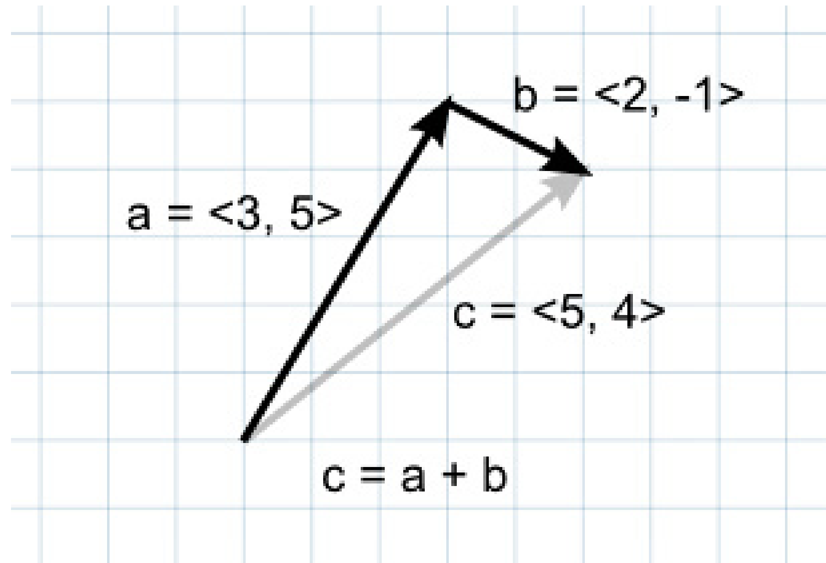
Pri izdelovanju iger imamo pogosto opravka z vektorji in operacijami nad njimi, npr. smer v kateri se giblje igralec v igri predstavimo z vektorji, površino po kateri se giblje predstavimo z normalnim vektorjem, pri računanju fizike uporabljamo vektorje za predstavitev različnih sil itd. Pomembnejše operacije z vektorji so:

- Seštevanje vektorjev (Slika 2.1). S seštevanjem dveh vektorjev dobimo tretji vektor, ki je enak poti iz začetne točke po obeh vektorjih do končne poti.

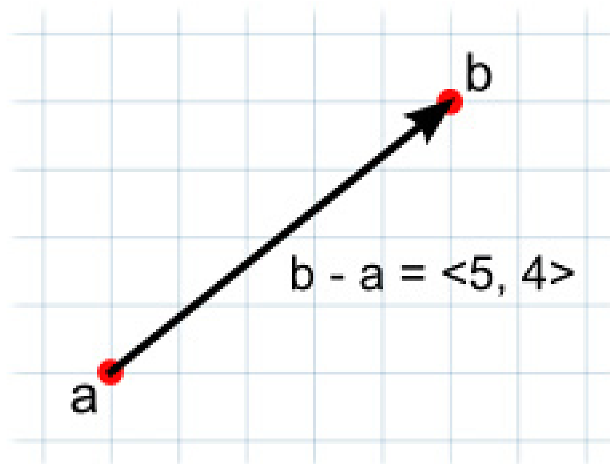
- Odštevanje vektorjev (Slika 2.2). S tem lahko ugotovimo vektor smeri, s katerim se premaknemo z ene točke v drugo.

- Skalarni produkt(Slika 2.3). Z njim lahko izračunamo koliksen del enega vektorja se razteza v smeri drugega vektorja.

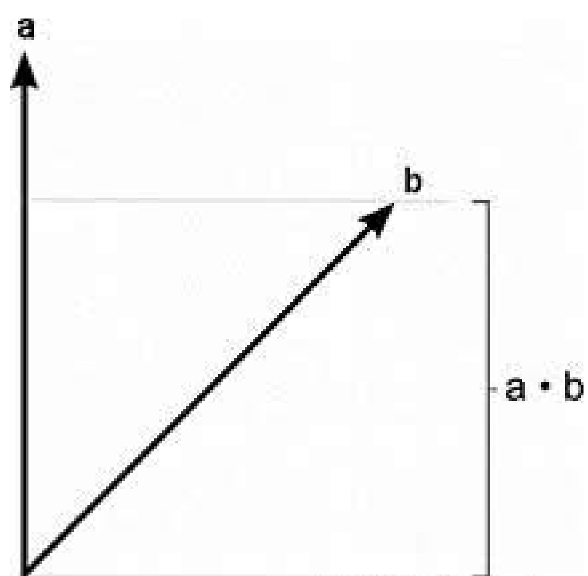
- Križni produkt (Slika 2.4). Z njim izračunamo vektor, ki je pravokoten na dva druga vektorja.



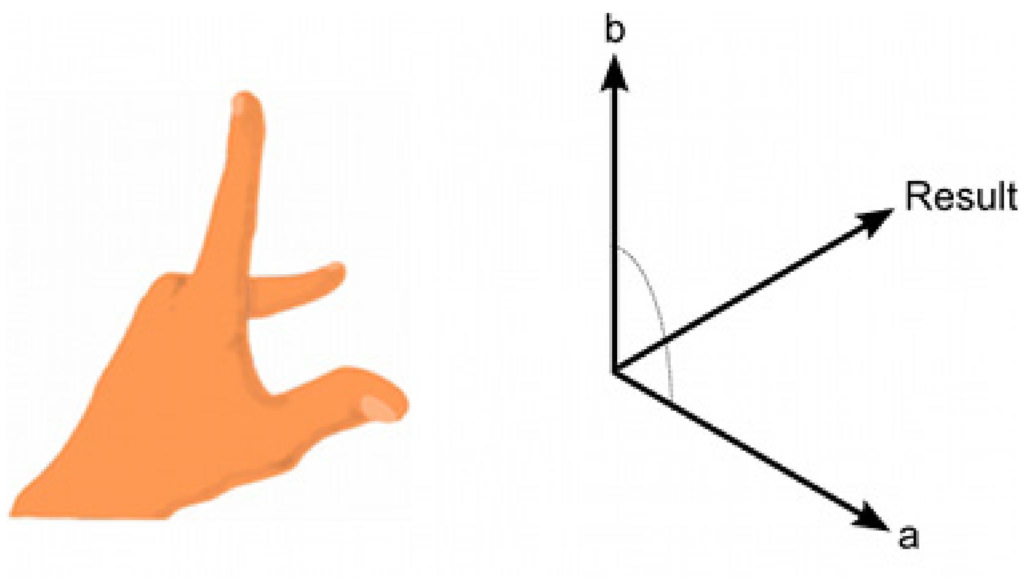
Slika 2.1: Seštevanje vektorjev.



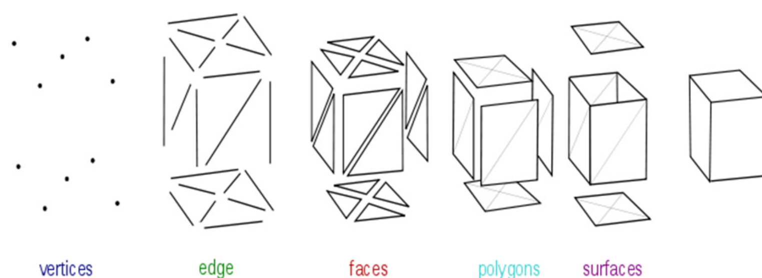
Slika 2.2: Odštevanje vektorjev.



Slika 2.3: Skalarni produkt.



Slika 2.4: Križni produkt.



Slika 2.5: Predstavitev 3D modelov.

2.2 Modeliranje

Igre vsebujejo veliko 3D modelov, kot so karakterji v igri, orožja, bonusi, vozila in raznorazni drugi predmeti. Vse te predmete je potrebno nekako zmodelirati in spraviti v digitalno obliko. Do 3D modelov lahko pridemo na različne načine. Lahko najamemo nekoga, da za nas naredi 3D model, model lahko poiščemo v obstoječih bazah modelov (npr. http://www.turbosquid.com) , kjer so na voljo plačljivi in brezplačni 3D modeli, model lahko z uporabo posebnih naprav pridobimo s skeniranjem resničnih predmetov, če pa nič od tega ni možno, bomo morali sami ustvariti 3D modele s pomočjo programov za 3D modeliranje.

Popularnejši plačljivi programi za modeliranje so 3ds Max, Maya, Softimage, Cinema 4D, od brezplačnih pa je najbolj popularen Blender. Vsi ti programi so obsežni in vsebujejo veliko funkcij, od različnih načinov modeliranja, orodij za izdelavo materialov, s katerimi oblečemo naše 3D modele, do orodij za animacijo. To je razlog, da traja kar nekaj časa, da se začetnik privadi dela s temi programi.

V igrah so 3D modeli običajno predstavljeni s mnogokotniškimi mrežami (ang. polygon mesh). Te so sestavljene iz oglišč (ang. vertex), robov (ang. edge), ki povezujejo dve oglišči, ploskev (ang. face), ki so zaključene množice robov, kjer ima trikotna ploskev tri robove, štirikotna ploskev pa ima štiri robove. Množici ploskev pravimo mnogokotnik (ang. polygon). Grafično predstavitev teh pojmov si lahko ogledamo na Sliki 2.5.

Slika 2.6: 3D model z okostjem.

Poleg samega modeliranja nas pri 3D modelih osebkov, ki jih bomo animirali ali upravljali, čaka še eno delo in to je izdelava okostja. Osebki so tako sestavljeni iz dveh delov: površine s katero osebke izrisujemo ter hierarhične množice povezani kosti, ki jih uporabimo za animacijo [1]. S premikanjem kosti se ustrezno premakne in deformira površina (ki je predstavljena z mnogokotniškimi mrežami). Zato je pomembno, da na mestih, kjer se bo površina bolj deformiramo (npr. pri sklepih) povečamo natančnost površine, torej da tam predstavimo površino s čim več ploskev, tako da dodamo dodatna oglišča. Kosti so med seboj v hierarhiji, kar pomeni, da ko premaknemo eno kost se ustrezno premaknejo tudi vse kosti, ki so podrejene tej kosti. Primer 3D modela, ki ima vgrajeno okostje si lahko ogledamo na Sliki 2.6.

Za posebno realistični videz je mogoče poleg okostja simulirati tudi mišice, vendar to v večini primerih ni potrebno, v poštev bi prišle pri realističnih bližnjih posnetkih osebkov.

2.3 Osvetljevanje

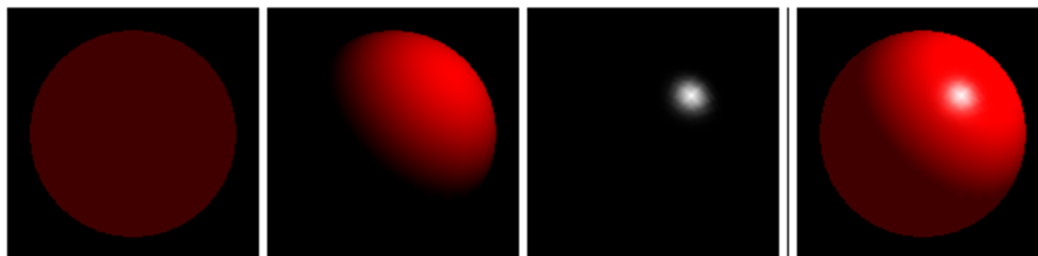
Osvetlitev v naravi je matematično gledano kompleksna stvar. Ko sije sonce je osvetljeno skoraj vse, čeprav sonce neposredno ne sveti na vse, kar lahko vidimo. To je zato, ker se svetloba odbija sem ter tja in tako zadane skoraj vse. Poleg tega svetloba zadene tudi delce v zraku, ki razkropijo to svetlobo in vse to pomeni, da bi bilo simuliranje resničnega modela osvetlitve zelo kompleksno in verjetno neizračunljivo, vsaj ne v realnem času. Zato se v računalniški grafiki uporabljajo modeli, ki poskušajo samo približno posnemati končni videz osvetlitve v resničnem življenju. To se naredi z delitvijo osvetlitve na tri različne tipe osvetlitve, ki združeno dovolj natančno posnemajo resnično osvetlitev. Tem trem osvetlitvam pravimo ambientalna osvetlitev (*ambient light*), difuzna osvetlitev (*diffuse light*) in svetloba sijaja (*specular light*).

Ambientalna osvetlitev je osvetlitev, za katero pravimo, da je povsod. To je osvetlitev, ki je ne moremo pripisati točnemu izvora. Ne glede na to kje se predmet nahaja v prostoru, vedno bo osvetljen z enako količino ambientalne osvetlitve.

Difuzna osvetlitev je osvetlitev, ki neposredno sveti na predmet. Intenziteta difuzne svetlobe je odvisna od kota med izvorom svetlobe in osvetljenim predmetov, ni pa odvisna od kota iz katerega mi opazujemo predmet. Ta svetloba nam omogoča, da si predmet predstavljamo kot 3 dimenzionalen predmet v prostoru (deli predmeta, ki so neposredno obrnjeni proti svetlobi, so bolj osvetljeni, medtem ko so tisti deli, ki so obrnjeni stran od izvora osvetlitve oziroma so zakriti, tudi manj osvetljeni).

Svetloba sijaja je svetloba, ki je odvisna od kota med izvorom svetlobe in osvetljenim predmetom ter hkrati tudi od kota iz katerega gledamo na predmet. Uporablja se pri predmetih, katerih površina je gladka, da dosežemo učinek sijaja (npr. pri predmetih kovinskega videza).

Na Sliki 2.7 si lahko ogledamo 3D kroglo, ki je osvetljena z različnimi osvetlitvami. Na prvem delu slike vidimo 3D kroglo, ki je osvetljena samo z ambientalno osvetlitvijo. Vidimo, da krogla izgleda kot 2D krog in ne kot 3D



Slika 2.7: Krogla osvetljena z različnimi vrstami osvetlitve.

predmet. Na drugem delu slike lahko vidimo to isto kroglo, ki je osvetljena samo z difuzno osvetlitvijo. Vidimo, da je zdaj že razvidna 3D oblika krogle. Na tretjem delu slike lahko vidimo ponovno isto kroglo, ki pa je osvetljena samo s svetlobo sijaja. Na zadnjem delu slike pa ponovno vidimo kroglo, ki jo osvetlimo z vsemi temi osvetlitvami hkrati.

Poglavje 3

Fizikalni pogoni

Fizikalni pogon je programska oprema, ki se uporablja za približno simulacijo določenih fizikalnih sistemov, kot so dinamika togih teles, dinamika mehkih teles (telesa, ki jih lahko deformiramo) in dinamika tekočin [7]. V osnovi poznamo dva razreda fizikalnih pogonov, realnočasovnega in visoko natančnega. Realnočasovni se uporablja pri igrah in drugih interaktivnih aplikacijah. Zanj je značilno, da uporablja poenostavljene izračune in zmanjšano natančnost, da lahko izvede izračune dovolj hitro, da igra deluje še vedno dovolj tekoče. Visoko natančni fizikalni pogoni pa se uporabljajo pri filmih in različnih znanstvenih izračunih, kjer je tem pogonom na voljo več procesorske moči in več časa, da lahko fizikalne izračune izvedejo s čim večjo natančnostjo. Eden izmed prvih programskih fizikalnih pogonov je bil računalnik ENIAC, ki je bil uporabljen za simuliranje topovskih granat glede na spremenljivke za maso, kot, pogon in veter.

Fizikalni pogon je simulator, ki se ga uporablja za izgradnjo virtualnega okolja, ki vsebuje zakone iz fizičnega sveta. To virtualno okolje lahko vsebuje objekte in spremljajoče sile, ki delujejo nad temi objekti (npr. gravitacija), poleg tega pa tudi interakcije med temi objekti, kot so trki. Fizikalni pogon simulira Newtonovo fiziko v simuliranem okolju ter upravlja s temi silami in interakcijami [13].

Računanje fizike je za fizikalne pogone kar zahtevno, zato uporabljajo

različne trike, da si to poenostavijo in tako omogočijo tekoče izvajanje igre. Eden izmed takih trikov je ločevanje objektov na dva tipa, statične objekte (se nikoli ne premaknejo) in dinamične objekte (se lahko premaknejo). Če pravilno označimo statične in dinamične objekte lahko fizikalni pogon izvaja določene optimizacije glede na dejstvo, da statični objekti nikoli ne trčijo v druge statične objekte. Fizikalni pogoni si olajšajo delo tudi s tem, da imamo dve predstavitvi vsakega objekta. Prva predstavitev je mnogokotniška mreža, ki se uporablja za izris samega objekta, medtem ko se za potrebe računanja fizike uporabi druga preprostejša oblika, ki je take velikosti, da zaobjame resnično obliko objekta. Ta preprostejša oblika je običajno krogla, elipsoid, cilinder, kapsula, kvader ali konveksna ogrinjača. Fizikalni pogoni se pogosto poslužujejo tudi hierarhije algoritmov. Tako lahko npr. pri iskanju trkov najprej poženejo hitre algiritme nad temi enostavnejšimi predstavitvami objektov ter tako pridobijo seznam potencialnih trkov, ki ga nato uporabijo pri počasnejših algoritmih za iskanje resničnih trkov.

Obstajata dva načina, ki ga fizikalni pogoni uporabljajo za zaznavanje trkov, diskretno (*diskrete*) in zvezno (*continuous*). Pri diskretnem zaznavanju trkov se na določen časovni interval izvaja računanje trkov. Sam trk se zazna po tem, ko do njega pride, saj se najprej izvede premik objektov in se šele nato pogleda, ali se zdaj kakšna objekta sekata med seboj. Ker se zaznavanje trkov pri takem načinu izvaja v določenem časovnem intervalu, se lahko pri objektih, ki se hitro premikajo ali pa so zelo ozki, zgodi, da se trka ne zazna, saj se je med tem časovnim intervalom objekt lahko že premaknil skozi objekt, v katerega je trčil. Primer, kjer se lahko to zgodi, je streljanje naboja pištole skozi list papirja. Prednost diskretnega zaznavanja trikov je enostavnost algoritma za zaznavanje trkov, saj se algoritem ne rabi zavedati različnih fizikalnih spremenljivk. Algoritmu se samo posreduje seznam fizičnih teles, ta pa vrne seznam teles, ki se sekajo med seboj. Pozicije in poti teh objektov je potrebno nato nekako popraviti, tako da se upošteva ugotovljene trke. Sam algoritem za zaznavanje trkov ne rabi razumeti trenja, elastičnih trkov, neelastičnih trkov in deformabilnih teles. Pri

diskretnem zaznavanju trkov se problemi pojavijo v fazi popravljanja, ko je potrebno popraviti vsa mesebojna sekanja (ki niso fizikalno pravilna) [15].

Zvezno zaznavanje trkov je drugačno od diskretnega. Tukaj ne čakamo, da se sam trk že zgodi, ampak natančno izračunamo trenutek trka. Algoritem za zaznavanje trkov mora biti v tem primeru sposoben natančno napovedati poti fizičnih teles. Pri takem zaznavanju trkov nikoli ne pride do tega, da bi se fizična telesa sekala med seboj, saj algoritem trke prej napove. Pri zveznem zaznavanju trkov torej popravimo konfiguracijo fizičnih teles po tem, ko smo izračunali trenutke trkov. Prednost zveznega zaznavanja trkov je povečana stabilnost. Pri zveznem zaznavanju trkov je težko (ni pa popolnoma nemogoče) ločiti fizikalno simulacijo od algoritma za zaznavanje trkov.

Fizikalni pogoni lahko podpirajo oba načina zaznavanja trkov. V tem primeru uporabimo diskretno zaznavanje trkov, če želimo čim večjo hitrost izvajanja, v kolikor pa se nam pojavijo problemi, kot je nezaznavanje trkov, pa je bolje uporabiti zvezno zaznavanje trkov.

3.1 PhysX

PhysX je lastniški realnočasovni fizikalni pogon, ki je od leta 2008 v lasti podjetja Nvidia [8]. To, da je PhysX v lasti proizvajalke grafičnih procesorjev, je ena izmed prednosti tega fizikalnega pogona, saj je Nvidia zanj omogočila strojno pospeševanje na njenih grafičnih procesorjih. To ima dve prednosti, hitrejše računanje fizike in obenem razbremenitev centralne procesne enote, ki lahko opravljajo drugo delo, namesto računanja fizike. PhysX je najbolj uporabljen fizikalni pogon pri igrah za Windows, saj je licenca za njegovo uporabo brezplačna za nekomercialno in komercialno uporabo. Drugače velja za Linux, Mac OS X in Android, kjer je uporaba brezplačna samo za izobraževalno in nekomercialno uporabo ter komercialno uporabo za razvijalce z manj kot 100.000 $ letnega prometa [12].

Fizikalni pogon PhysX podpira Windows, Mac OS X, Linux, PlayStation 3, Xbox 360, Wii in glavne mobilne platforme. Uporabljajo ga tudi mnogi

pogoni za igre, kot so Unreal Engine, Unity, Gamebyro, Vision, Instinct Engine, Panda3D, Diesel, Torque, HeroEngine in BigWorld.

PhysX podpira fiziko togih teles, nadzor osebkov (uporabno predvsem pri prvoosebnih in tretjeosebnih igrah), simuliranje vozil, na večini platform podpira tudi večprocesorske sisteme, podpira tudi simulacijo tekočin, blaga (tudi trganje blaga in trke samega vase), simulacijo mehkih teles in simulacijo volumenskih polj sil (uporabno za veter ali cone brez gravitacije) [9].

3.2 Havok

Tudi Havok Physics je lastniški realočasovni fizikalni pogon. Razvija ga irsko podjetje Havok, ki pa je od leta 2007 v lasti podjetja Intel. Popularnejši je pri igrah narejenih za konzole ter pri kakovostnejših igrah, za kar naj bi bil zaslužen obsežen nabor orodij, orientiranost h konzolam in odlična podpora za razvijalce [10]. Čeprav je Havok komercialen fizikalni pogon, lahko kupci licence po potrebi dobijo tudi vpogled v določene dele izvorne kode tega pogona.

Fizikalni pogon Havok deluje na Windows, Windows RT, Windows Phone, Unix, Linux, Android, Mac OS X, iOS, Xbox, Xbox 360, PlayStation 2, PlayStation 3, PlayStation Portable, PlayStation Vita, GameCube, Wii in Wii U [11].

3.3 Bullet

Bullet je eden izmed popularnejših fizikalnih pogonov, ki je odptrokoden, razvija pa ga bivši Havok zaposlenec. Uporabljen je bil pri mnogih igrah (npr. Grand Theft Auto 4) in tudi za vizualne efekte v filmih (npr. Shrek 4, Hancock, 2012) [14]. Uporablja ga tudi Blender, odprtokodni program za modeliranje. Pogon podpira simulacijo togih in tudi mehkih teles (blago, vrv in telesa, ki jih lahko deformiramo). Trke med objekti zna zaznati diskretno in tudi zvezdno.

Bullet deluje na Windows, Linux, Mac OS X, iOS, Android, PlayStation 3, XBox 360, Wii in drugje. V primerjavi s komercialnimi pogoni je dokumentacija tega pogona manj obsežna in manj prijazna do uporabnika, ima pa uporabnik to prednost, da lahko sam pogleda v njegovo izvorno kodo in po potrebi vnese kakšne spremembe.

Poglavje 4

Izdelava 3D igre

Izdelovanja 3D iger se lahko lotimo na dva načina, z direktno uporabo grafičnih knjižnic ali pa uporabimo pogone za igre, ki v ozadju namesto nas dostopajo do grafičnih knjižnic.

Izdelava iger preko pogonov za igre je dosti lažja in hitrejša, med izdelovanjem igre pa imamo tudi boljšo predstavo, saj nam pogoni za igre sproti vizualizirajo igro in nam omogočajo, da svet v igri sestavljamo vizualno. 3D modelov nam ni potrebno premikati z računanjem koordinat, ampak lahko vse premike, obračanja in povečavo modelov izvajamo vizualno preko grafičnega vmesnika. Pogoni za igre nam v primerjavi z grafičnimi knjižnicami poleg samega izrisa 2D ali 3D grafik nudijo dosti dodatnih funkcionalnosti. Tipične dodatne funkcionalnosti so vgrajen fizikalni pogon ter zaznavanje trkov (in odziv na trke), zvok, skripte, animacija, umetna inteligenca, mrežna podpora, upravljanje s pomnilnikom, večnitnost, lokalizacijska podpora in scenski graf [16]. Med funkcionalnostmi najdemo tudi sistem za delce in urejevalnike za gradnjo terena. Pogoni za igre omogočajo tudi dostop do različnih naprav, ki so značilne za platforme, ki jih pogon podpira (npr. miške, tipkovnice, giroskopa in merilca pospeškov v mobilnih napravah, spletne kamere). Prednost mnogih pogonov za igre je tudi podpora več platformam, kar nam omogoča, da pri prenašanju igre na drugo platformo večji del kode ponovno uporabimo, popravimo pa samo tiste dele, ki so specifični

za določeno platformo. S tem prihranimo na stroških in naredimo razvoj iger bolj ekonomičen. Prihranimo tudi pri razvoju več različnih iger, saj lahko ponovno uporabimo enak pogon za igre oziroma ga prilagodimo tudi za izdelavo drugih iger. Pogoni za igro nam zmanjšajo tudi kompleksnost ter čas do plasiranja igre na trg. Še ena prednost pogonov za iger so programski jeziki, v katerem pišemo igro, ki so pogosto visokonivojski, pogoni pa nam pogosto omogočajo tudi delo z več različnimi programskimi jeziki hkrati.

Imajo pa pogoni za igre tudi nekaj negativnih lastnosti. Določeni pogoni so specializirani in optimizirani samo za izdelavo določenih žanrov iger, npr. prvoosebnih strelskih iger ali dirkaških iger, kar pomeni, če mi izdelujemo drugačno igro, da bomo imeli več dela ali pa sploh ne bomo mogli narediti igre tako, kot smo si jo zamislili. Prav tako so določeni pogoni narejeni samo za izdelava 2D iger, drugi samo za 3D igre, nekateri pa za 2D in 3D igre. Slabost pogonov za igre je tudi cena, saj je uporaba dovršenih in uporabniku prijaznih pogonov na določenih platformah plačljiva, vendar če upoštevamo znesek inženirskih ur, ki bi jih brez pogonov porabili za izdelavo določenih funkcionalnosti iger, se nam ta znesek poplača v hitrejšem in lažjem delu. Obstajajo tudi brezplačni ali odprtokodni pogoni za igre, vendar so ti pogosto uporabniku manj prijazni, manj dovršeni, učenje dela z njimi pa nam vzame več časa. Pogosto so pogoni tudi delno plačljivi oz. obstajata dve verziji pogona, brezplačna okrnjena verzija pogona z manj funkcijami ter plačljiva verzija z več funkcijami.

Pogoni za igre imajo za seboj kar pestro zgodovino. Pred pogoni za igre so morali izdelovalci iger za vsako novo igro začeti ponovno od začetka. Eden izmed razlogov je bila tudi strojna oprema, ki je predstavljala resno oviro pri izdelovanju iger, saj je bilo potrebno na vsakem koraku izdelovanja iger imeti v glavi vse njene omejitve in pomankljivosti, kot je majhna količina pomnilnika. To je tudi onemogočalo izdelovanje iger po principu uporabe pogonov za igre. Tudi na platformah, ki niso bile tako omejujoče, je bilo le malo stvari, ki se jih je dalo ponovno uporabilo med igrami. Hiter napredek v strojni opremi arkad, ki so bile tisti čas vodilne na trgu, je pomenil, da se je

moral večji del kode vseeno zavreči, saj so igre poznejših generacij uporabljale drugačne zasnove iger, ki so izkoriščale prednosti dodatnih resursov.

Prve oblike pogonov za igre so se začele najprej pojavljati znotraj samih podjetij, ki so izdelovale igre. Ti so za potrebe določene igre izdelali nekakšen pogon za igre, ki so ga lahko kasneje uporabili tudi pri izdelovanju kakšnih drugih iger ali pa za nove verzije obstoječih iger. Pogoni tretjih oseb niso bili običajni do vzpona 3D računalniških grafik v desetletju 1990, medtem ko so že v desetletju 1980 obstajali 2D sistemi za ustvarjanje iger. Med njimi so Pinball Construction Set (1983), War Game Construction Kit (1983), Thunder Force Construction (1984), Adventure Construction Set (1984), Garry Kitchen's GameMaker (1985), Wargame Construction Set (1986), Shoot'Em-Up Construction Kit (1987) in Arcade Game Construction Kit (1988). Izraz 'pogon za igre' se je uveljavil v sredini 1990-ih, posebej v povezavi s 3D igrami, kot so prvoosebne strelske igre. Popularnost iger Doom in Quake je privedla do tega, da so drugi razvijalci, namesto da bi začeli iz nič, licencirali jedrne dele te programske opreme in dizajnirali svojo grafiko, like, orožja in stopnje. Separacija delov, ki so specifični za igro od osnovnih konceptov (kot je zaznavanje trkov), je pomenila, da se je ekipa lahko povečala in specializirala. Poznejše igre, kot je Quake 3 Arena, so bile zasnovane s tem konceptom v mislih, da se pogon in vsebina igre razvija ločeno. Praksa licenciranja takih tehnologij se je za določene razvijalce iger izkazala kot uporaben dodaten vir zaslužka. Ponovno uporabljivi pogoni tudi pohitrijo in olajšajo proces izdelovanja nadaljevanj iger, kar je cenjena prednost v konkurenčni industriji video iger.

Moderni pogoni za igre so eni izmed najbolj kompleksno napisanih aplikacij, ki pogosto vsebuje mnogo dovršenih sistemov, ki medsebojno komunicirajo, da zagotovijo natančno nadzorovano uporabniško izkušnjo. Neprestana evolucija pogonov za igre je ustvarila strogo ločitev med izrisovanjem, programiranjem, igrino umetnostjo in načrtovanjem stopenj. Zdaj je npr. običajno, da tipično skupino, ki razvija igro, sestavlja več umetnikov kot pravih programerjev [17]. Danes je tudi čisto normalno, da pogone za igre izdelujejo

podjetja, ki so specializirana samo za to in sama ne razvijajo iger, razen krajših demonstracijskih iger, s katerimi prikažejo sposobnosti in prednosti njihovega pogona. Uporaba pogonov za igre se je razširila tudi na bolj resna področja: vizualizacije, trening, medicina in vojaške simulacije. Da bi še povečali dostopnost, moderni pogoni za igre ciljajo tudi na nove platforme, med njimi tudi na mobilne telefone (npr. Android, iPhone) in spletne brskalnike (npr. WebGL, Flash).

Poglavje 5

Android

Android je operacijski sistem namenjen predvsem mobilnim napravam z zaslonom na dotik, kot so pametni telefoni in tablični računalniki. Njegova priljubljenost od dneva predstavitve leta 2007 ves čas hitro narašča. Do tretjega kvartala leta 2012 je bilo aktiviranih že 500 milijonov naprav in v povprečju je bilo vsakega dne tega kvartala 1,3 milijona novih aktivacij. To pomeni, da si lasti že 75 % tržni delež med pametnimi telefoni [2].

Android operacijski sistem razvija Google in je osnovan na Linux operacijskem sistemu ter je izdan pod odprto kodo. Poleg funkcionalnosti samega operacijskega sistema pa za Android obstaja tudi že preko 700.000 plačljivih in brezplačnih aplikacij, ki jih pišejo razvijalci z vsega sveta. Aplikacije so različnih vrst, kot so igre v 2D in 3D pogledu, programi za navigacijo ter različni drugi programi, ki nam omogočajo določene funkcionalnosti in storitve. Največ aplikacij se distribuira preko uradne spletne trgovine Google Play. Ta je že naložena v obliki aplikacije na Android napravah, brskanje po trgovini pa je možno tudi preko brskalnika. Aplikacija Google Play skrbi tudi za posodabljanje obstoječih aplikacij na novejše verzije. Ko odpremo to aplikacijo le-ta pogleda, ali so na voljo novejše verzije aplikacij, ki jih že imamo na telefonu. Google Play nam lahko avtomatsko posodobi aplikacije, ali pa ročno izberemo aplikacije, ki jih želimo posodobiti. Poleg uradne trgovine obstaja še nekaj drugih trgovin (npr. Amazon Appstore), ki pa niso tako

popularne. Obstaja pa tudi ročna namestitev aplikacij na telefon v obliki datotek s končnico ".apk".

Programska arhitektura operacijskega sistema Android (Slika 5.1) sestoji iz jedra osnovanega na Linux jedru, ki pa ga je Google malenkost priredil, da deluje bolje v mobilnih napravah. Na nivoju višje od jedra najdemo programske knjižnice, med njimi tudi grafično knjižnico OpenGL ES, ki se uporablja za izdelovanje iger za Android in knjižnico SGL, ki je Googlova lastna 2D grafična knjižnica. Na tem nivoju najdemo tudi zagonsko okolje Android, ki sestoji iz navideznega stroja Dalvik in jedrnih knjižnic (*Core Libraries*). Na nivoju nad tem najdemo aplikacijsko ogrodje, ki je napisano v Javi in vsebuje nabor osnovnih komponent, ki jih uporabljajo vse aplikacije med svojim delovanjem. Na najvišjem nivoju pa najdemo aplikacije, ki se prav tako pišejo v Javi.

Vsaka aplikacija se v Androidu izvaja v svojem procesu v svoji instanci navideznega stroja Dalvik, ki je bi posebno zasnovan za delovanje v vgrajenih sistemih (*Embedded Systems*), kjer je treba upoštevati omejitve napajanja (zmogljivost baterije), procesorske moči in velikosti pomnilnika [3].

Sicer obstajata dve verzije grafične knjižnice OpenGL ES, in sicer OpenGL ES 1.1, ki jo najdemo na starejših in manj zmogljivih napravah, ter OpenGL ES 2.0 na novejših napravah (od Android verzije 2.2 dalje), vendar je delež naprav, ki podpirajo OpenGL ES 2.0 že preko 90 % (podatki iz oktobra 2012), tako da se lahko pri razvoju iger običajno osredotočimo na OpenGL ES 2.0, razen če želimo podpreti čisto vse Android naprave. Bistvena razlika med novejšo in starejšo verzijo OpenGL ES je to, da novejša podpira delo s senčniki (*shaders*).

Preden začnemo z izdelavo Android aplikacij se je potrebno odločiti, katero verzijo Androida bomo podprli oz. katerim bomo namenili največ pozornosti. Android je vnaprej kompatibilen, kar pomeni, da bodo vse aplikacije, ki so narejene za eno verzijo Androida podpirale tudi vse verzije Androida, ki tej verziji sledijo. V praksi to npr. pomeni, da bo aplikacija, ki je bila napisana za Android verzije 2.3 brez sprememb deloval tudi na Androidu verzije

Slika 5.1: Programska arhitektura Androida.

Verzija	Kodno ime	Porazdelitev
1.6	Donut	0,2%
2.1	Eclair	2,4%
2.2	Froyo	9%
2.3 - 2.3.2	Gingerbread	0,2%
2.3.3 - 2.3.7	Gingerbread	47,4%
3.1	Honeycomb	0,4%
3.2	Honeycomb	1,1%
4.0.3 - 4.0.4	Ice Cream Sandwich	29,1%
4.1	Jelly Bean	9%
4.2	Jelly Bean	1,2%

Tabela 5.1: Statistika uporabe posameznih verzij Androida (januar 2013).

4.1. Obratno pa seveda ne velja, zato je dobro razmisliti, katerim verzijam Androida bomo namenili največ pozornosti. Tu nam pride prav statistika uporabe posameznih verzij Androida. Iz statistike, ki si jo lahko ogledamo na Tabeli 5.1, lahko vidimo, da je Android Gingerbread najpogostejša verzija, saj jo najdemo na približno 47 % naprav Android (podatki iz januarja 2013) [4]. To kaže tudi na slabost Androida in sicer, da izdelovalci Androidov hitro pozabijo na starejše verzije naprav in ne pripravijo posodobitev programske opreme, ampak se osredotočijo samo na bolj aktualne modele, čeprav so pogosto lahko tudi starejši modeli strojno dovolj zmogljivi za poganjanje novejših verzij, kar je razvidno tudi iz tega, da obstajajo skupnosti, ki izdajajo tudi neuradne novejše verzije Androida tudi za starejše modele naprav.

Ena izmed slabosti razvijanja aplikacij za platformo Android je podpiranje ogromnega števila naprav. Vsaka naprava ima drugačne specifikacije in tudi kakšne hrošče, čemur se morajo razvijalci prilagoditi. Nekateri razvijalci aplikacij si želijo, da bi Google izdelovalcem Android naprav postavil omejitve, ki bi pomagale pri standardizaciji naprav [18]. Ko kupec kupi aplikacijo

ali igro pričakuje, da ta deluje. Pomanjkanje kompatibilnosti ima za posledico slabe ocene in posledično slabšo prodajo aplikacij. Ocene aplikacij so med glavnimi dejavniki, ki vplivajo na število nakupov. Pri Androidu je težje zagotoviti kompatibilnost kot pri Applovi platformi iOS. Skoraj vsak dan izzide kakšna nova Android naprava, ki jih izdelujejo različni proizvajalci. Pri Applu v letu dni izzide morda ena ali dve novi napravi. Android platforma je dosti bolj fragmentirana. Imamo mobilne telefone in tablične računalnike. Zaslon na napravah so različnih velikosti, razmerij in ločljivosti. Imamo naprave z enojedrnimi in večjedrnimi procesorji. Naprave se razlikujejo tudi v grafičnih zmogljivostih, saj imajo različne grafične procesorje. Različne so tudi velikosti pomnilnikov. Nekatere naprave imajo vgrajene več različnih senzorjev, druge manj. Večina naprav ima za upravljanje samo zaslon na dotik, nekaj jih ima tudi fizično tipkovnico. Vse te razlike je potrebno pri razvijanju aplikacij za Android upoštevati.

5.1 Android pogoni za igre

Obstaja veliko število pogonov za igre, ki podpirajo razvoj tudi za Android. Večji del teh pogonov podpira samo izdelovanje 2D iger (npr. AndEngine, Cocos2d-x, App Game Kit). Med njimi je nekaj dobrih tudi brezplačnih. Ponudba 3D pogonov, ki podpirajo tudi Android platformo, pa je bolj skopa, najbolj dovršeni pogoni (npr. Unity, ShiVa, SIO2) so plačljivi, brezplačni (npr. jPCT-AE, libGDX) pa niso tako dovršeni in enostavni za uporabo. Tudi ponudba Android iger je podobna ponudbi Android pogonov za igre, dosti več je 2D iger kot pa 3D iger.

Poglavje 6

O pogonu za igre Unity

Unity je pogon za igre primarno namenjen 3D igram, čeprav se da z njim izdelovati tudi 2D igre. Pogon podpira izdelavo iger za več različnih platform, Windows, Mac, Unity Web Player (vtičnik za spletni brskalnik), iOS, Android, Nintendo Wii, PlayStation 3, Xbox 360, v bodoče pa bo podpiral ali pa že testno podpira še platforme Adobe Flash Player, Linux, Windows 8, Windows Phone 8, Nintendo Wii U. Unity nam omogoča programiranje igre v treh različnih jezikih, Javascript, Boo in C# [5].

Unity je v osnovi na voljo v dveh izvedbah, brezplačni Unity Free in Unity Pro, ki stane 1875 $. V osnovnih izvedbah ni vključeno izdelovanje iger za Android, tako da moramo za izdelovanje Android iger pri Unity Free kupiti dodatek za 500 $, pri Unity Pro pa dodatek za 1875 $. Z obema izvedbama pogona lahko izdelamo popolno igro, razlika med Unity Pro in Unity Free je v tem, da Unity Pro vsebuje dodatne funkcionalnosti, ki lahko predvsem pripomorejo k bolj realističnemu videzu igre (npr. dinamične sence).

Unity ima vgrajen ustvarjalec terena. Z njim lahko preko grafičnega vmesnika ustvarjamo teren v naši igri, torej nam ga ni potrebno ustvariti v posebnem programu za modeliranje in ga nato uvažati v Unity. Teren ustvarjamo z risanjem. Imamo različne čopiče s katerimi dvigujemo in nižamo teren, ga gladimo ali pobarvamo. Za barvanje imamo že vgrajene osnovne teksture, kot so npr. trava, pesek, mivka, kamen. Seveda lahko uporabimo tudi svoje

teksture in z njimi pobarvamo teren. Enostavno lahko s čopiči postavljamo tudi drevesa, travne bilke, kamne in podobne predmete.

S pogonom za igre Unity lahko enostavno dodamo tudi oceane in jezera v našo igro. Za reke in potoke, ki tečejo po terenu pa v osnovi ni vgrajene podpore, obstajajo pa orodja tretjih oseb, ki jih lahko dodamo v Unity preko vgrajene trgovine sredstev (*Asset Store*). V njej najdemo tudi druge orodja (npr. za risanje cest), teksture in materiale, 3D modele, projekte iger in druga sredstva. Nekateri sredstva so plačljiva, najdejo se pa tudi taka, ki so brezplačna. V pogon je vgrajen tudi sistem za delce, s katerim lahko ustvarjamo efekte kot so ogenj, eksplozija, dim, megla, dež, slap, prah, fontane in podobno. Nekaj osnovnih delcev (npr. ogenj, dim, prah) je že vgrajenih, še več jih lahko dobimo v vgrajeni trgovini ali pa uporabimo svoje.

Pri izdelovanju realističnih iger se moramo posluževati fizike. Unity ima podporo za fiziko vgrajeno (v ozadju za fiziko skrbi Nvidiin fizikalni pogon PhysX). Objekt v igri je lahko tako podvržen različnim silam in se giblje realistično, če pa si tega ne želimo, pa lahko objekt upravljamo tudi sami. Tudi za zaznavanje trkov je ustrezno poskrbljeno, saj lahko objekt v igri ovijemo v enega izmed trkalnikov. Na voljo so osnovni trkalniki oblike kvadra, krogle in kapsule, za teren in nepremične predmete, pa lahko uporabimo tudi trkalnike, ki se popolnoma prilegajo obliki terena oziroma nepremičnega predmeta. Kompleksnejši objekt lahko razbijemo tudi na manjše objekte in vsakega od njih ovijemo v enega izmed osnovnih trkalnikov. Na voljo je tudi posebni trkalnik za kolesa, ki ga uporabimo pri kolesih vozil, da se ta obnašajo realno glede na sile trenja. Objektom v igri lahko določimo tudi lastnosti trenja in odbojnosti, tako da lahko enostavno realiziramo ledeno površino, po kateri predmeti drsijo ali pa kroglico, ki se odbija. Zaznavanje trkov je lahko izvedeno diskretno ali zvezno.

Unity ima vgrajen tudi sistem za animacijo. Z njim lahko animiramo humanoidne objekte v igri. Poskrbljeno je tudi za iskanje poti. Generiramo lahko navigacijsko mrežo, ki je poenostavljena predstavitev sveta v igri, po kateri znajo nato agenti v igri navigirati. Agentu določimo ciljno lokacijo

in agent bo sam poiskal ustrezno pot, pri čemer bo upošteval tudi različne dinamične elemente v igri.

V Unity je vgrajenih tudi nekaj načinov za pohitritev igre. Vgrajena je podpora za različne nivoje detajlov, kar nam pride prav pri velikih scenah. S tem lahko prikažemo bolj podroben model objekta v igri, ko mu je kamera blizu, ko pa objekt gledamo bolj od daleč, pa je prikazan manj natančen model objekta. Unity pospeši izris igre tudi s tem, da ne izrisuje objektov, ki so izven vidnega polja kamere. V Unity lahko izris igre pospešimo tudi s tem, da ne izrisujemo objektov, ki jih kamera trenutno ne vidi, ker jih drugi objekti zakrivajo.

Poglavje 7

Izdelava iger s pogonom Unity

Izdelovanje igre v Unity se začne z ustvarjanjem novega projekta, ki predstavlja našo igro. Vsaka igra ima nato eno ali več scen. V to sceno nato vnašamo 3D modele, ki sestavljajo našo igro. Vsem objektom se v Unity reče GameObject. Vsakemu objektu v osnovi pripada komponenta Transform, s katero nastavljamo pozicijo, rotacijo in skaliranje. Za vsako dodatno funkcionalnost objektu dodamo dodatne komponente. Pomembnejše komponente so:

- Mesh Filter – dodamo objektu mrežo (ki jo zmodeliramo s programom za modeliranje, npr. 3ds Max, Maya, Blender).
- Mesh Renderer – s to komponento postane mreža resnično vidna. Če uporabimo samo Mesh Filter mreža ni vidna v igri, se pa uporablja npr. pri trkih.
- Particle System – dodamo objektu sistem za delce.
- Rigidbody – z dodajanjem te komponente se objekt začne obnašati po pravilih fizike. Nastavimo mu lahko maso in zračni upor.
- Character Controler – to komponento dodamo karakterjem, ki jih upravljamo pri prvoosebnih ali tretje osebnih igrah, če ne želimo, da so pod nadzorom fizike (da se lahko npr. karakter v trenutku ustavi in začne premikati v drugo smer). Še zmeraj pa deluje zaznavanje trkov in pri

premikanju hodimo po podlagi in stopnicah ter ob trku s steno drsimo ob steni.

- Box Collider, Sphere Collider in Capsule Collider – trkalniki v obliki kvadra, krogle in kapsule.
- Mesh Collider – trkalnik, ki prevzeme obliko mreže objekta. Uporabljamo ga lahko samo na nepremičnih objektih.
- Wheel Collider – trkalnik, ki ga uporabljamo pri kolesih vozil.
- Camera – s to komponento spremenimo objekt v kamero.
- Light – s to komponento spremenimo objekt v izvor svetlobe.
- Script – tudi programske kode, ki upravljajo z objektom, dodamo kot komponento.

7.1 Programiranje v Unity

Igro v Unity programiramo s pisanjem skript, ki jih nato kot komponente pripnemo objektom v igri. Pisanje samih skript se vrši izven Unity vmesnika. Ob namestitvi pogona Unity se namesti tudi MonoDevelop IDE, čeprav lahko po želji uporabimo poljuben program za pisanje kode. Pri programiranju v Unity sta bistveni funkciji *Update* in *FixedUpdate*. Update je funkcija, ki se kliče pred posodobitvijo izrisa igre. Tukaj naj bi bila večina kode, ki upravlja z obnašanjem igre. Funkcija FixedUpdate pa se kliče enkrat na določen časovni interval, kjer naj bi se izvajala koda, ki ima opravka s fiziko. Poleg teh dveh funkcij imamo se funkcije, ki se kličejo ob določenih dogodkih (npr. funkcija OnCollisionEnter se kliče, ko objekt trči v drug objekt) [6].

Primer kode, ki obrača objekt za 5 stopinj na sekundo okoli osi Y si lahko ogledamo spodaj:

```
var speed = 5.0;
```

```
function Update () {
    transform.Rotate(0, speed*Time.deltaTime, 0);
}
```

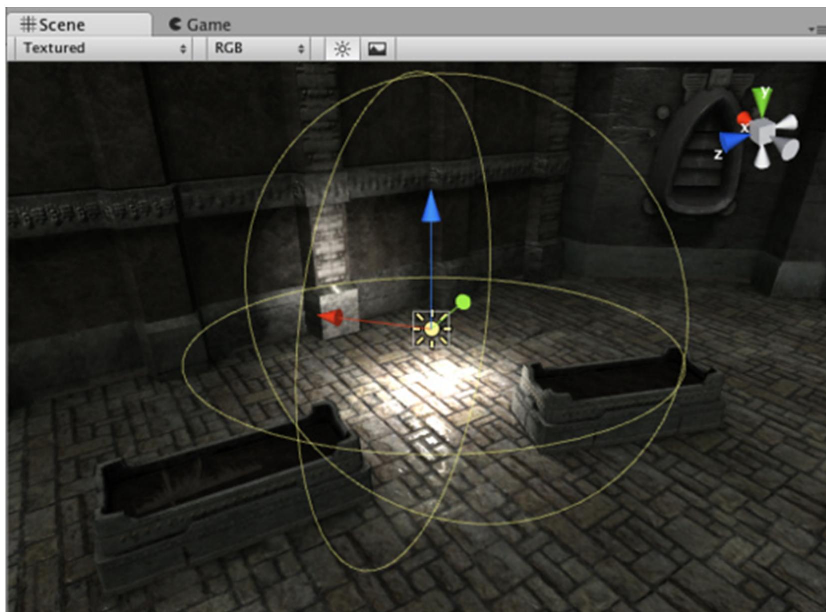
Spremenljivka *Speed* iz primera kode se po tem, ko to kodo pripnemo objektu, pojavi tudi v Unity urejevalniku, kjer lahko to spremenljivko spreminjamo, brez da bi morali spreminjati kodo, spreminjamo jo lahko celo med samim izvajanjem igre in se takoj uveljavi, brez da bi morali ugasniti in ponovno zagnati igro. Če se želimo v kodi sklicevati na nek drug objekt deklariramo spremenljivko, ki se nam nato pokaže v Unity urejevalniku, kjer lahko grafično izberemo želeni objekt (iz seznama ali pa z miško povlečemo objekt).

7.2 Osvetljevanje

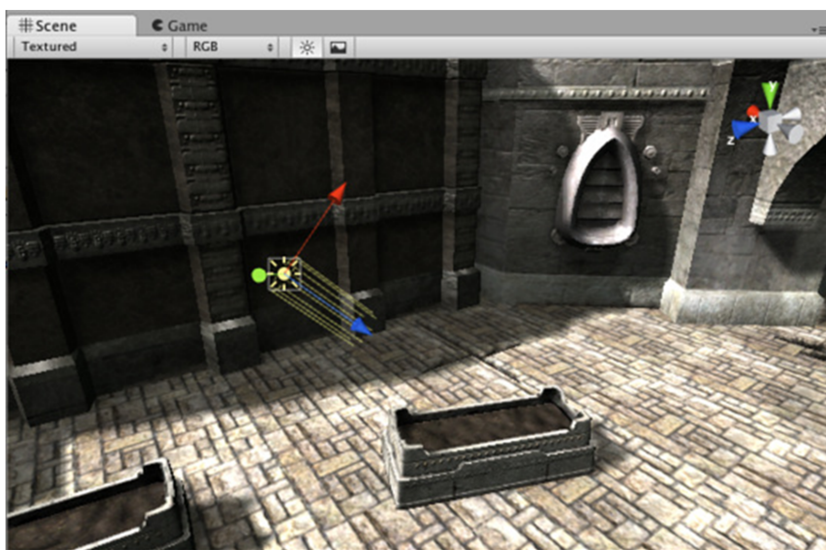
V primeru, da prostora v igri ne osvetlimo z lučmi vidimo samo temo. Osvetljevanje ima velik vpliv tudi na vzdušje v igri, saj lahko z različnimi načini osvetljevanja dosežemo drugačno vzdušje. Z lučmi lahko simuliramo sonce, gorečo vžigalico, svetilko, eksplozije in mnoge druge stvari.

Unity pozna 4 tipe luči. Točkovna luč (Slika 7.1) je luč, ki sveti iz ene točke z enako močjo v vse smeri, kot običajna žarnica. Usmerjena luč (Slika 7.2) je luč, ki je postavljena neskončno stran in vplivajo na vse v sceni, podobno kot sonce. Tretji tip luči je svetlobni slop (Slika 7.3), ki sveti iz ene točke v eni smeri in osvetli predmete znotraj stožca, kot žarometi pri avtomobilu. Četrta luč pa je luč, ki sveti iz ene strani pravokotne ravnine in osvetli predmete znotraj dosega luči.

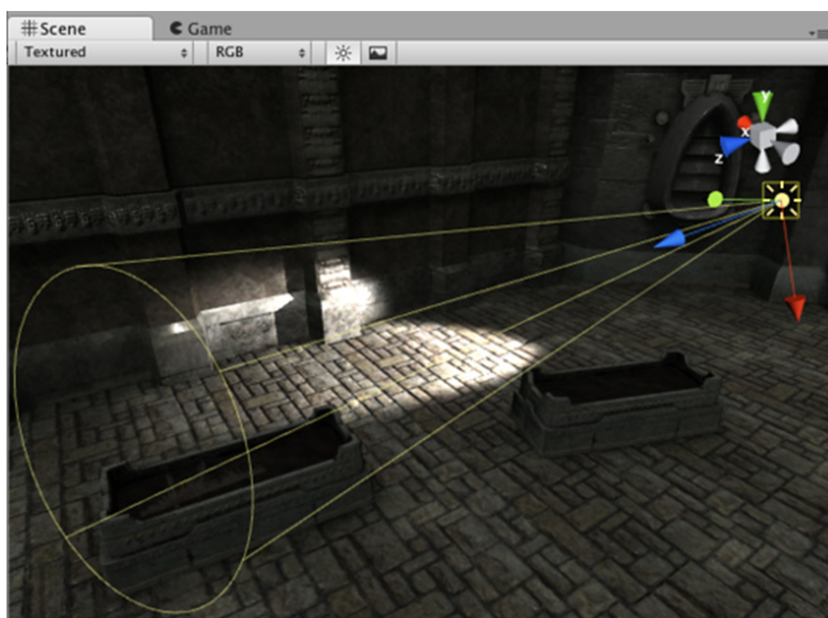
Lučem v Unity lahko dodamo tako imenovane piškote (*Cookies*), ki nam dajo efekt, kot da svetloba sveti skozi nevidno kocko s teksturami na stranicah in osvetli prostor z vzorcem teh tekstur. Primer si lahko ogledamo na Sliki 7.4.



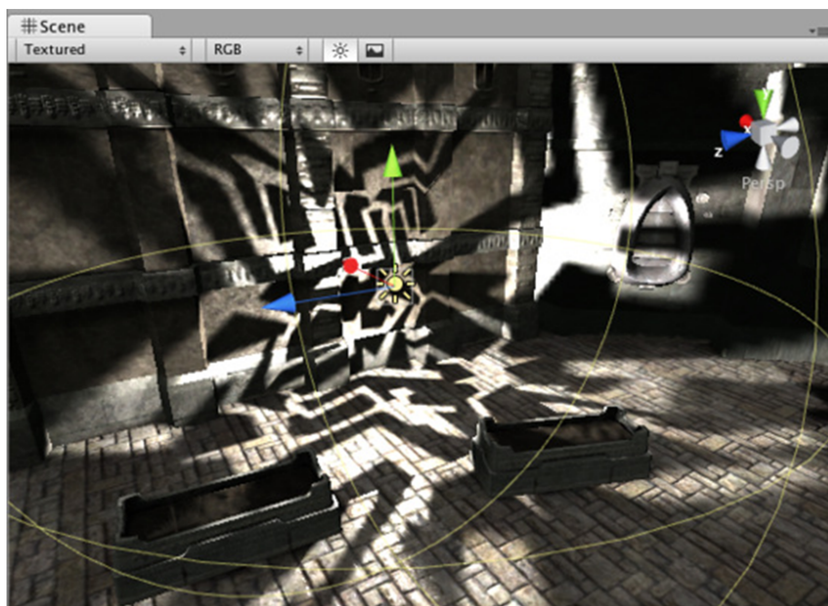
Slika 7.1: Točkovna luč.



Slika 7.2: Usmerjena luč.



Slika 7.3: Svetlobni snop.



Slika 7.4: Točkovna luč s piškotom.

7.3 Grafični uporabniški vmesnik

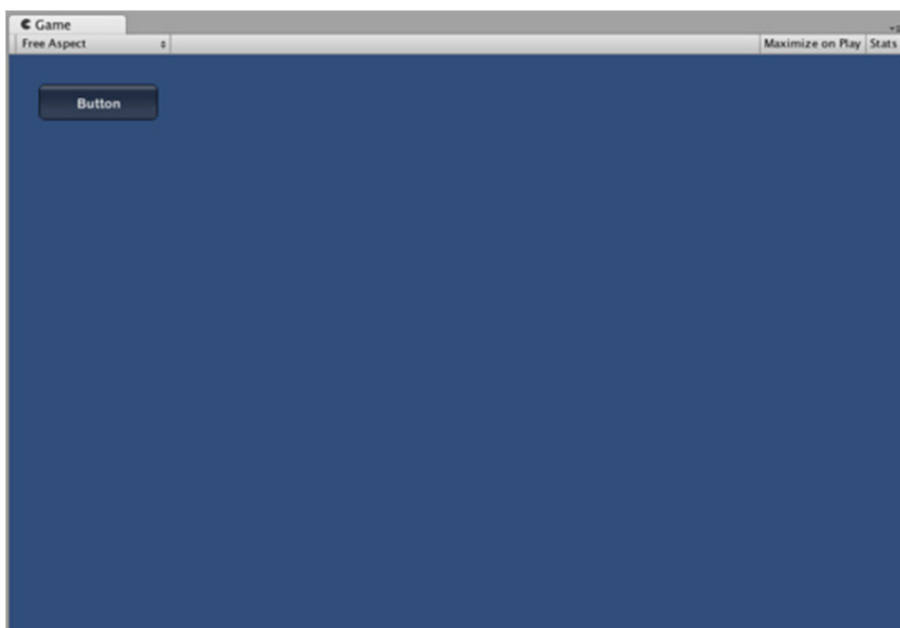
Skoraj vsaka igra mora imeti tudi grafični uporabniški vmesnik. Ta je sestavljen iz različnih menijev in gumbov, s katerimi izberemo stopnjo v igri, ki jo želimo igrati, nastavljamo različne nastavitve v igri, začasno zaustavimo ali zapremo igro itd. Sestavni del grafičnega vmesnika so tudi različni prikazi, kot je prikaz igralčevega trenutnega rezultata v igri, prikaz trenutnega časa, prikaz števila streliva v strelski igri itd.

V Unity je običajen način izdelovanja grafičnega vmesnika s pisanjem kode. Obstajata dva razreda, ki ju lahko uporabimo za izdelovanje grafičnega vmesnika, *GUI* ter *GUILayout*. Razlika med njima je v tem, da pri prvem fiksno postavljamo grafične gradnike (rečemo na kateri X in Y poziciji na zaslonu naj se gradnik postavi in kakšno višino ter širino naj ima) pri drugem pa je postavitev grafičnih gradnikov avtomatska, na podoben način kot HTML tabele. Fiksna postavitev nam pride v poštev, ko imamo definiran vmesnik in vemo koliko določenih grafičnih gradnikov želimo imeti, medtem ko nam avtomatska postavitev pride prav takrat, ko ne vemo točno koliko gradnikov bomo imeli oziroma se to število med izvajanjem spreminja, ali pa ko ne želimo na roke postavljati vsakega gradnika. Z različnimi grafičnimi gradniki, ki so nam na voljo, lahko na zaslon izrišemo tekst, ikone oziroma teksture, zaobljene pravokotnike, gumbe, polja za vnos teksta, potrditvena polja, vrsto gumbov, mrežo gumbov, navpične in vodoravne drsnike ter okna.

Primer kode, ki izriše gumb širok 100 in visok 30 točk oddaljen 25 točk od zgornje in leve strani zaslona s tekstom »Button«, ki ob kliku nanj izpiše obvestilo o tem v konzolo:

```
function OnGUI () {
    if (GUI.Button (Rect (25, 25, 100, 30), "Button")) {
        Debug.Log('Someone clicked the button!!!');
    }
}
```

Rezultat te kode si lahko ogledamo na Sliki 7.5.

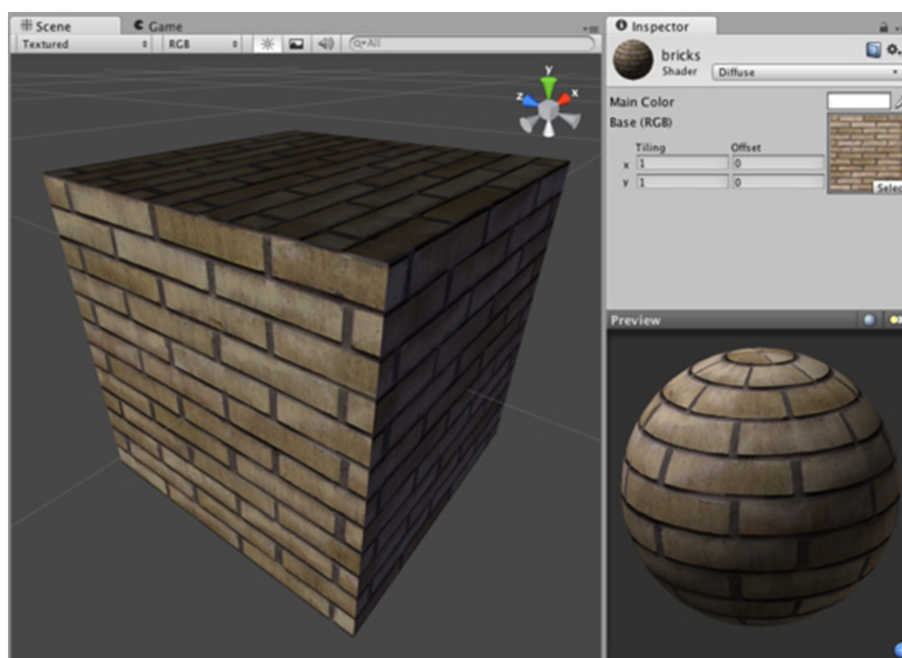


Slika 7.5: Izris gumba z Unity grafičnim vmesnikom.

Stil grafičnih gradnikov lahko prilagodimo svojim željam s spreminjanjem lastnosti, kot so barve, ozadja, razmiki itd. Stil lahko določimo vsakemu grafičnemu gradniku posebej, s pomočjo instance razreda *GUIStyle*, ali pa ustvarimo instanco razreda *GUISkin*, s katerim nastavimo stil vsem gradnikom.

7.4 Senčniki (*shaders*)

Izris vseh objektov se v Unity vrši s pomočjo senčnikov. Brez njih Unity ne ve, kako izrisati objekte v igri. Unity ima vgrajenih preko 60 senčnikov, vendar se najbolj pogosto uporablja samo nekaj od njih, medtem ko so preostali uporabni v posebnih primerih [19]. Poleg vgrajenih senčnikov je možno pisati tudi svoje senčnike, vendar je to primerno za bolj napredne uporabnike, saj je pisanje le-teh bolj nizkonivojsko v primerjavi s pisanjem običajnih Unity skript in je potrebno tudi dobro poznavanje delovanja grafičnih kartic.

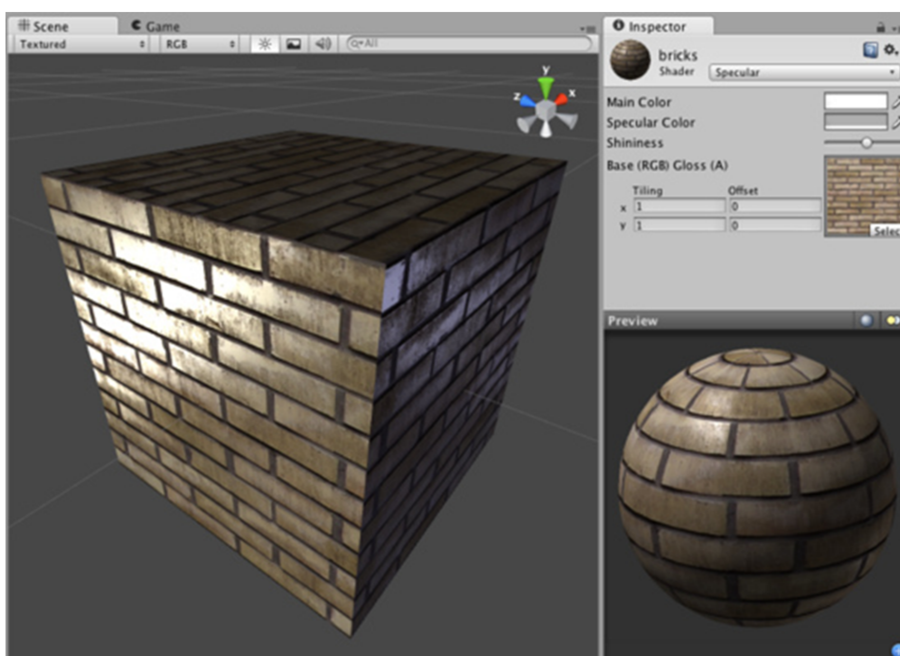


Slika 7.6: Senčnik Diffuze.

Senčniki se med seboj razlikujejo tudi v hitrosti izvajanja, tako da so nekateri primernejši za pospeševanje izrisa pri manj zmogljivih sistemih. Nekaj najbolj popularnih senčnikov:

- Diffuze – eden izmed preprostejših senčnikov, ki izriše objekte, kot da jih osvetljuje samo difuzna svetloba. Primer iz realnega življenja je neobdelan les, torej tak, ki se na lesketa. Ponuja nam dve možnosti, ki jo lahko izberemo. Prva je glavna barva, s katero nastavimo barvo objekta v kateri ga želimo izrisati. Druga možnost je pa tekstura, ki jo lahko izberemo in tako objekt izrišemo z izbrano teksturo. Primer uporabe tega senčnika si lahko ogledamo na Sliki 7.6.

- Specular – ta senčnik je podoben Diffuze senčniku s to razliko, da je izrisan predmet poleg difuzne svetlobe obsijan tudi s svetlobo sijaja. Položaj sijaja se ustrezno premakne, če se premakne kamera, s katero opazujemo objekt. Možnosti, ki nam jih ta senčnik ponudi so izbira

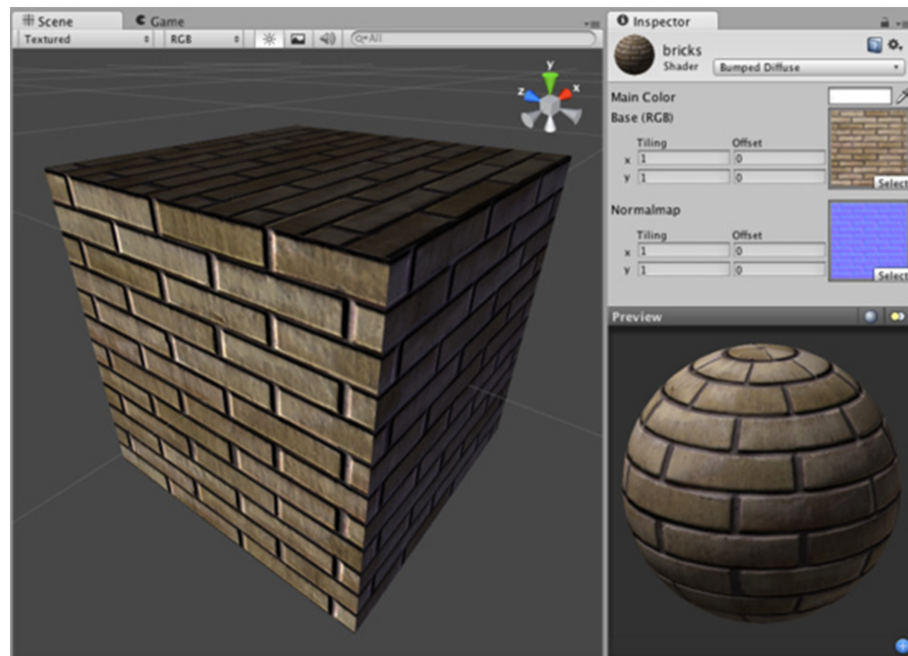


Slika 7.7: Senčnik Specular.

glavne barve, izbira barve sijaja, izbira velikosti sijaja in izbira teksture. Z alfa kanalom v teksturi lahko napravimo, da imajo deli objekta več sijaja, del objekta pa manj. Primer, kjer nam to pride prav, je zarjavela kovina, kjer bi nastavili manj sijaja, pri polirani kovini pa več. Primer uporabe tega senčnika si lahko ogledamo na Sliki 7.7.

- Bumped Diffuse – podoben senčnik kot Diffuse, ki pa ima dodatno možnost, da lahko izberemo še drugo teksturo, s katero lahko simuliramo manjše detajle (grbine in vdolbine), namesto da naredimo model z več poligoni. To je uporabno npr. za simuliranje rež pri opekah ali tlakovanih poteh. Primer uporabe tega senčnika si lahko ogledamo na Sliki 7.8.

Poleg teh omenjenih senčnikov so še senčniki, ki kombinirajo lastnosti več drugih senčnikov. Nekateri senčniki so namenjenih za prosojne objekte, nekateri za objekte, ki osvetljujejo samo samega sebe, nekateri pa so taki, ki



Slika 7.8: Senčnik Bumped Difuze.

prikazujejo odseve na objektih.

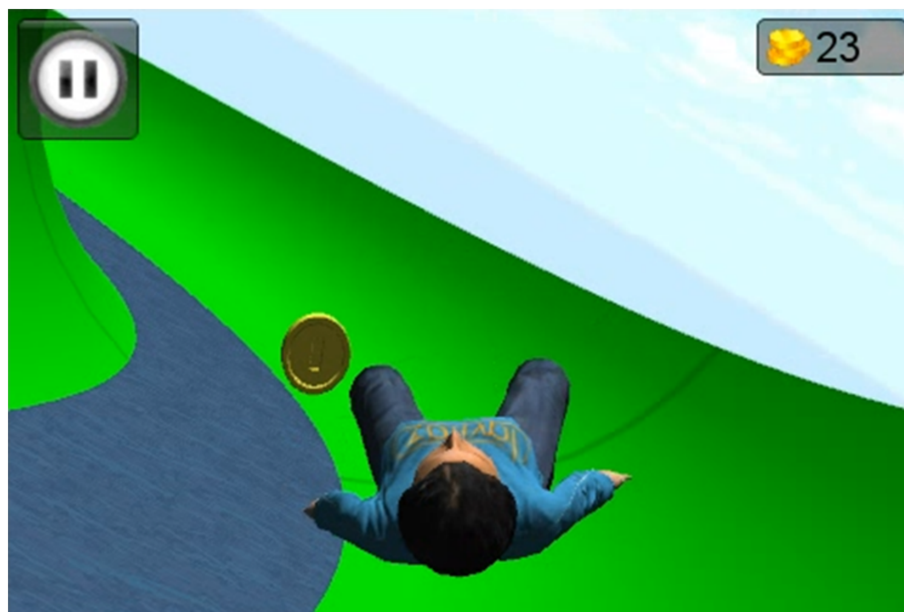
Poglavje 8

Lastno delo

Da bi tudi v praksi preverili, kako poteka izdelovanje 3D igre s pogonom Unity za platformo Android, smo razvili eno igro. Za inspiracijo smo vzeli igro Waterslide Extreme, ki obstaja za platformo iOS, ne obstaja pa za platformo Android. Odločili smo se, da bomo poskušali izdelati podobno igro, ki pa bo delovala na platformi Android. Na Sliki 8.1 si lahko ogledamo posnetek naše igre. V igri se spuščamo po vodnem toboganu in poskušamo pobrati čim več cekinov, ki so nastavljeni po toboganu. Igralec upravlja igralca v igri s premikanjem mobilnega telefona, in sicer, če zavrti telefon v levo, se bo tudi igralec v igri začel premikati proti levi strani tobogana, če pa telefon zavrti v desno, se bo igralec v igri premaknil proti desni strani tobogana.

Na začetku se je bilo potrebno najprej spoznati z dvema orodjema, ki smo ju uporabili pri izdelavi igre, to je pogon za igre Unity in program za modeliranje 3ds Max. Predhodnih izkušenj z izdelavo iger s pogoni za igre nismo imeli, imeli pa smo nekaj splošnega znanja o računalniški grafiki in grafični knjižnici OpenGL. Spoznavanje s pogonom za igre Unity in njegovim vmesnikom je bilo lažje kot spoznavanje s programom za 3D modeliranje 3ds Max.

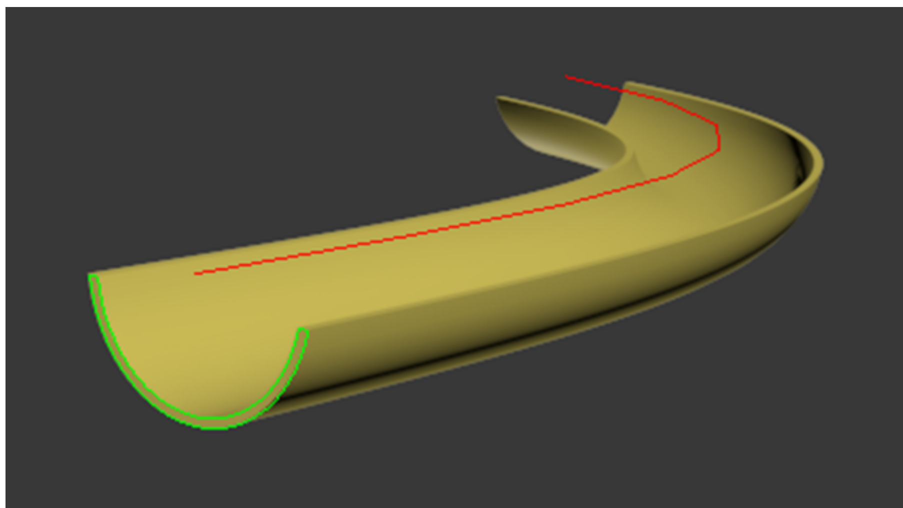
Spoznavanje z grafičnim vmesnikom pogona Unity ni pretirano zahtevno, saj vmesnik ni natrpan z različnimi funkcijami, ampak je logično razdeljen na posamezne dele, tako da se ga da hitro osvojiti. Tudi učenje programiranja



Slika 8.1: Posnetek naše igre.

Unity skript ni bilo preveč zahtevno, saj je nabor razredov in metod, ki jih ponuja programski vmesnik pogona Unity dovolj kratek, da se ga da preleteti in si ustvariti neko predstavo, kaj nam le-ta omogoča in kje poiskati določeno funkcionalnost, ki jo želimo realizirati. Na uradni strani je dobra dokumentacija, ki pomaga tako pri razumevanju Unity vmesnika kot tudi pri razumevanju Unity skript.

Spoznavanje s programom za modeliranje 3ds Max pa je malo bolj zapleteno. To je program, ki vsebuje ogromno nekih funkcij in posledično je uporabniški vmesnik precej natrpan. Vse to pomeni, da je za začetnika kar naporno, da se ga navadi. Funkcij programa je preveč, da bi se naenkrat seznanil z njimi, zato je spoznavanje z njim postopno. Dobro si je pogledati osnovne video vodiče, ki nam dajo neko osnovno znanje, ko pa potrebujemo poznati posamezne funkcije programa si lahko pogledamo specifične videe, ki nam predstavijo uporabo teh funkcij, ali pa si preberemo dokumentacijo za te funkcije.



Slika 8.2: Del tobogana narejen z orodjem Loft.

8.1 Vodni tobogan

Eden izmed sestavnih delov, ki sestavlja našo igro je tobogan, po katerim drsi lik v igri. Pri izdelavi vodnega tobogana si nismo mogli pomagati z izdelovalcem okolja, ki je vgrajen v Unity, saj je le-ta namenjen izdelavi naravnih okolij, kot so hribi in doline, trava in pesek itd. Tudi z obstoječimi 3D modeli, ki so na voljo na internetu, si nismo mogli pomagati, saj nobeden ni ustrezal našim potrebam. Ni nam preostalo drugega, kot da ga izdelamo sami s pomočjo programa za 3D modeliranje. Po nekaj poskusih z različnimi orodji se je za ta namen najbolje izkazalo orodje Loft v programu za modeliranje 3ds Max. Ta nam omogoča, da narišemo dve krivulji, ena predstavlja presek druga pa pot in orodje Loft nam iz tega naredi 3D model, ki gre po podani poti in ima po vsej dolžini podani presek. Primer lahko vidimo na Sliki 8.2, kjer rdeča krivulja predstavlja pot, zelena pa presek.

Zdaj, ko smo znali narediti tobogan, se je bilo treba odločiti, kako bomo to prenesli v Unity. Ali naredimo celoten tobogan iz enega dela in ga celega uvozimo v Unity ali pa naredimo sestavne kose, ki jih nato v Unity sestavljamo v večji tobogan. Sprva smo poskusili prvo možnost, ki pa se ni izšla.

Prva slabost je ta, da večji kot je 3D model dlje traja uvoz v Unity, ki včasih sploh ni uspel in je prišlo do sesutja programa. Tudi ko nam je v Unity uspelo v enem delu uvoziti celoten tobogan, je preveliko število trikotnikov, ki sestavljajo model tobogana, povzročilo, da ga Unity sploh ni izrisal, saj ima Unity omejitev v številu trikotnikov, ki lahko sestavljajo eno posamezno mnogokotniško mrežo.

Odločiti smo se torej morali za drugo možnost, da bomo v Unity uvozili manjše sestavne dele in tam sestavili celoten tobogan. Naredili smo torej tri dele, raven del, del, ki zavija v levo, in del, ki zavija v desno. Tudi ta postopek ni potekal brez problemov. Problem je bil v postavljanju sestavnih delov tobogana v 3D prostoru, saj je bilo potrebno vsak nov del poravnati z obstoječim toboganom. Pri tem je bilo potrebno zadeti pozicijo novega dela v vseh treh smereh prostora in tudi rotacijo novega dela okoli osi Y. Za pozicijo v prostoru smo s časom ugotovili, da ima Unity funkcijo imenovano Vertex Snapping, ki nam omogoča, da poljubno oglišče novega dela enostavno priključimo poljubnemu oglišču obstoječega tobogana. S tem smo si olajšali nastavljanje pozicije novega dela, še vedno pa je bilo potrebno ročno nastavljati rotacijo okoli osi Y. Pri izdelovanju delov tobogana, ki zavijajo, smo morali najti tudi pravi kot zavijanja, ki ni smel biti ne prevelik ne premajhen. Če je bil kot prevelik, igralec igre ni mogel dovolj hitro reagirati, da bi pobral cekine, saj je bil njegov pogled naprej zaradi močnega ovinka preveč omejen in ni mogel pravočasno opaziti cekinov. Tudi premajhen kot zavoja ni dober, saj bi bila igra preenostavna, ker igralca ne bi moglo nič presenetiti.

8.2 Igralec

3D model za igralca, ki se spušča po vodnem toboganu, smo našli na internetu. Bil je že opremljen z okostjem, ki smo ga ustrezno premaknili, da smo dobili igralca v ustreznem položaju za spust po toboganu. Zdaj je bilo potrebno ugotoviti, kako pripraviti igralca, da bo drsel po vodnem toboganu in bo obenem mogoče upravljati njegove premike navzgor v levo ali desno stran

tobogana, da bo lahko pobiral cekine. V resničnem življenju, ko se spuščamo po toboganu, nimamo skoraj nič nadzora ampak drsimo po tobogan glede na različne sile, ki delujejo na nas. Da bi imela naša igra tudi igralni faktor in ne bi bila samo računalniška simulacija drsenja po vodnem toboganu, smo morali realizirati gibanje našega igralca, ki ne deluje ravno po zakonih fizike. To je razlog, da nismo mogli uporabiti fizikalnega pogona, ki je vgrajen v Unity, ampak smo morali premikanje igralca izvajati sami.

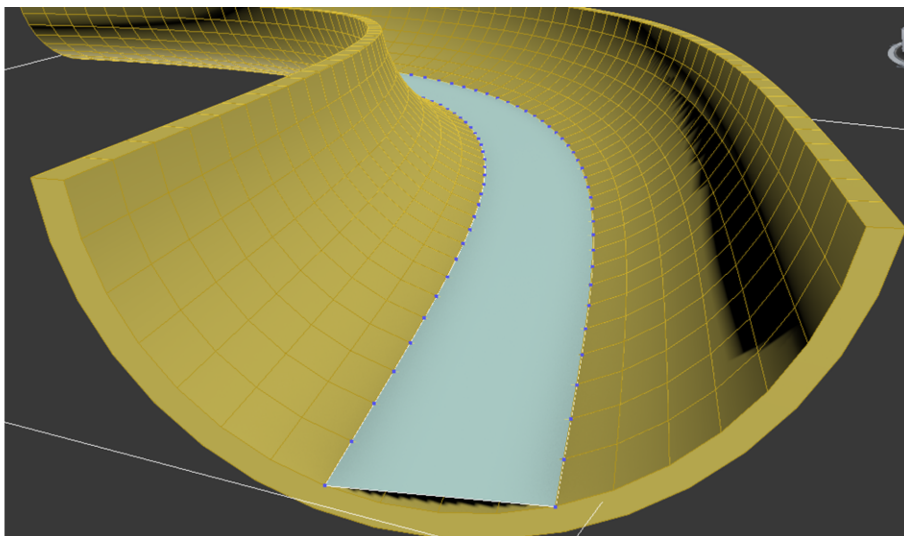
Pri lastni izvedbi premikanja igralca po vodnem toboganu je bilo potrebno paziti na več stvari. Najprej je bilo potrebno, da je med spuščanjem po toboganu igralec vedno ležal na površini tobogana in je bil poravnan s površino pod njim. Poleg tega je moral igralec ob zavoju tobogana tudi sam ustrezno zaviti. Med vsemi temi premiki je moral obstati vedno na istem relativnem mestu tobogana, torej če je bil na sredini tobogana je moral ostati na sredini, če je bil bolj na levi strani tobogana je moral tam tudi ostati, dokler ga ni igralec igre z vrtenjem mobilnega telefona premaknil v drug relativni položaj.

Premikanje igralca smo realizirali tako, da postavimo igralca v nevidno kroglo, ki je ravno take velikosti, da je njen premer enak premeru tobogana. Ta krogla se nato drsa po toboganu in je vedno usmerjena v smeri tobogana. Ko igralec igre zavrti mobilni telefon, da bi premaknil igralca levo ali desno, se krogla ustrezno zavrti okoli svoje osi Z in s tem postavi igralca na pravo lokacijo v toboganu. Vsakič, ko se krogla premakne, se s pomočjo streljanja žarkov ugotovi, kam se mora krogla premakniti in kako mora biti obrnjena, da je njena os Z vedno poravnana s smerjo tobogana. Streljanje žarkov je metoda, ki se v Unity pogosto uporablja za različne izračune. Deluje tako, da izberemo 3D lokacijo, iz katere želimo izstreliti žarek, in vektor smeri, v katero želimo izstreliti žarek. Žarek se izstreli in ko zadene kakšen objekt lahko izvemo različne informacije o mestu zadetka, npr. kakšen je normalni vektor, ki kaže pravokotno na mesto zadetka. Nam pride streljanje žarkov prav pri poravnavi krogle, ki vsebuje igralca, da z njegovo pomočjo izračunamo smer, v katero kaže tobogan, in nato kroglo ustrezno poravnamo s to smerjo. Smer izračunamo tako, da izstrelimo eden žarek v levo stran tobogana in drug

žarek v desno stran tobogana. Dobimo dva normalna vektorja, ki kažeta pravokotno na podlago. Zdaj samo še izračunamo križni produkt med tema dvema vektorjema in dobimo smer, ki kaže ravno v smeri tobogana. Pri poravnavanju s smerjo tobogana je bilo treba paziti še na eno stvar, in sicer, da se je premike ublažilo, saj bi se v nasprotnem primeru opazilo trzanje. Do trzanja pride zaradi tega, ker so krivine pri toboganu predstavljene z omejeno natančnostjo z določenim številom odsekov. Ob vsakem premiku krogle se tudi znova izračuna, kje se nahaja sredina tobogana, da lahko kroglo postavimo točno na sredino tobogana. Tudi tukaj izstrelimo dva žarka v levo in desno stran. Dobimo dve razdalji, koliko smo oddaljeni od leve strani in koliko smo oddaljeni od desne strani. S pomočjo teh dveh razdalj izračunamo sredino tobogana in tam postavimo kroglo. Zadnjo stvar, ki jo moramo popraviti pa je višina, na kateri je krogla. Tukaj izstrelimo žarek proti dnu tobogana in ugotovimo, za koliko moramo premakniti kroglo gor ali dol, da bo ležala točno na dnu tobogana.

8.3 Cekini

V naši igri so po toboganu nastavljeni cekini, ki jih mora igralec čim več pobrati. Model zanje smo našli na internetu. Model smo uvozili v Unity in ustvarili več objektov cekinov, ki smo jih nastavili na različne lokacije po toboganu tako, da se mora igralec premikati iz ene strani tobogana na drugo, da bi jih čim več pobral. Cekinom smo v Unity dodali komponento trkalnika in napisali ustrezno logiko, ki zazna trk med igralcem in cekinom, poveča igralčev rezultat in cekin uniči. Igralčev rezutat se sproti tudi izpisuje v zgornjem desnem kotu. Pri izrisu le-tega smo si v Unity pomagali z razredom *GUI*.



Slika 8.3: Izdelava vodne površine v programu 3ds Max.

8.4 Voda

V igri simuliramo tudi vodo, ki teče po toboganu. Za to smo morali v programu za modeliranje 3ds Max narediti za vsak sestavni del tobogana tudi ustrezno površino, ki je malo dvignjena od dna tobogana in predstavlja vodo. Paziti je bilo treba na to, da se je popolnoma ujemala z določenim delom tobogana. Kako to zgleda v programu 3ds Max si lahko pogledamo na Sliki 8.3.

To površino smo podobno kot tobogan uvozili v Unity ter vse dele tobogana in vodne površine sestavili tako, da izgleda kot ena celota. Na to vodno površino smo simulirali vodo z uporabo kode in senčnika za izris vode, ki sta že vgrajena v Unity.

8.5 Hobotnica

Naša igra vsebuje tudi oviro v obliki hobotnice. V primeru, da se hobotnici ne izognemo in se zaletimo vanjo, se nam naše vidno polje zakrije s črnilom. To črnilo nam zakriva pogled in nam otežuje pobiranje cekinov, dokler črnila ne odstranimo. Črnilo odstranimo tako, da s prstom povlečemo po telefonu,

Slika 8.4: Posnetek naše igre, preden se zaletimo v hobotnico.

od leve strani do desne strani. Na Sliki 8.4 si lahko ogledamo posnetek naše igre preden se zaletimo v hobotnico, na Sliki 8.5 pa posnetek igre po trku v hobotnico. To funkcijo igre smo realizirali tako, da smo poiskali model hobotnice in ga uvozili v Unity. Nato je bilo potrebno napisati logiko, ki zazna trk s hobotnico. Poiskali smo tudi primerno sliko črnila in jo malenkost popravili v grafičnem programu, da nam je bolje ustrezala. Črnilo na zaslonu izrišemo s pomočjo skripte, ki jo kot komponento dodamo kameri. Ta skripta je že vgrajena v Unity in nam omogoča, da na zaslon izrišemo poljubno teksturo. Ob trku s hobotnico aktiviramo to skripto in jo pustimo aktivno, dokler ne zaznamo ukaza za deaktivacijo, ki ga sproži uporabnik s potegom prsta po zaslonu. V Unity v osnovi ni vgrajenega sistema, ki bi sam prepoznal določene kretnje s prsti po zaslonu. Unity nam samo pove podatke, kot so, koliko prstov se dotika zaslona, lokacije teh dotikov, fazo dotika (ali se je dotik začel, se premika po zaslonu, je stacionaren ali se je dotik zaključil) in za koliko se je od zadnje spremembe dotik premaknil po zaslonu. Iz teh podatkov moramo sami prepoznavati kretnje po zaslonu.

Slika 8.5: Posnetek naše igre, po trku s hobotnico.

8.6 Testiranje

Našo igro smo testirali na Androidnem telefonu Samsung Galaxy Ace. To je dve leti star telefon srednjega razreda. Na njem igra teče tekoče. Sprva je igra na trenutke delovala počasneje, dokler nismo ustrezno označili kateri objekti v igri so statični in kateri ne. To dejanje je poenostavilo delo fizikalnemu pogonu. Čas nalaganja stopnje ni takojšen, ampak traja nekje 11 sekund od trenutka, ko kliknemo na gumb ”Play”v začetnem meniju, do trenutka, ko se igranje stopnje začne. Igra kar obremeni telefon in prav dosti dodatnih grafično zahtevnih funkcionalnosti igre ne bi več prenesel. Poskusili smo dodati tudi sistem za delce, da bi simulirali pršenje vodnih kaplic, ko se spuščamo po toboganu, a je izvajanje tega konkretno upočasnilo igro, tako da smo to opustili.

V Unity urejevalnik je vgrajeno tudi orodje za spremljanje delovanja igre (*profiler*). Z njim lahko preverjamo, kako igra obremenjuje naš telefon med njenim izvajanjem. Preverjamo lahko obremenitev procesorja, grafične kar-

tice, porabo pomnilnika in kateri deli kode se izvajajo najdlje. Na našem testnem telefonu nam podatkov o uporabi grafične kartice ni prikazovalo, tako da nismo mogli vedeti, kako obremenjena je bila grafična kartica oziroma kateri deli igre jo najbolj obremenjujejo. To je zato, ker je preverjanje obremenitve grafične kartice podprto samo na določenih grafičnih karticah.

Unity ima za Android tudi mobilno aplikacijo Unity Remote, ki nam omogoča, da igro poganjamo kar na računalniku in uporabimo telefon kot upravljalec. To nam pride prav pri hitrejšem testiranju, saj nam ni treba za vsako spremembo ponovno zgraditi igre in jo prenesti na telefon. Pri takem testiranju igro izrisuje računalnik in posreduje sliko telefonu preko aplikacije Unity Remote, aplikacija pa posreduje nazaj računalniku podatke iz telefona (pospeškometer in dotike zaslona).

Poglavje 9

Sklepne ugotovitve

V diplomskem delu smo se seznanili s postopkom izdelave 3D igre za platformo Android in tudi v praksi izdelali igro. Pri tem smo spoznali določene koncepte, ki jih uporabljamo pri računalniški grafiki. Spoznali smo se tudi s postopki, ki so potrebni za izdelavo 3D igre. Pridobili smo si osnovna znanja uporabe programov za 3D modeliranje. Preizkusili smo tudi v praksi, kako poteka izdelava iger s pomočjo pogona za igre. Dobro smo se spoznali z uporabo pogona za igre Unity. Pogledali smo tudi v drobovje platforme Android in spoznali njeno arhitekturo. V praksi smo izdelali igro za mobilni telefon in jo tudi testirali na Android napravi. Samo igro bi se seveda dalo še v marsičem izboljšati.

Po vsem tem lahko rečemo, da izdelovanje iger za pametne telefone ni enostavno. Za izdelavo igre, ki bo všeč igralcem, bo imela realističen videz, dobro igralnost in bo ekonomsko uspešna, je potrebnih kar nekaj znanj. V primeru, da igro izdeluje ena oseba mora biti ta oseba pravi multipraktik. Imeli mora smisel za grafiko in znati uporabljati grafične programe, znati mora programirati in uporabljati vsa orodja, ki jih pri tem potrebuje, imeti mora domišlijo pri sestavljanju zgodbe igre itd.

Literatura

[1] (2013) Skeletan Animation Dostopno na:
http://en.wikipedia.org/wiki/Skeletal_animation

[2] (2013) Android (operating system) Dostopno na:
http://en.wikipedia.org/wiki/Android_(operating_system)

[3] (2013) Android platforma Dostopno na:
http://android.fri.uni-lj.si/index.php/Platforma

[4] (2013) Android Developers - Dashbords Dostopno na:
http://developer.android.com/about/dashboards/index.html

[5] (2013) Unity (game engine) Dostopno na
http://en.wikipedia.org/wiki/Unity_(game_engine)

[6] (2013) Unity Script Reference Dostopno na:
http://docs.unity3d.com/Documentation/ScriptReference/index.html

[7] (2013) Physics engine Dostopno na:
http://en.wikipedia.org/wiki/Physics_engine

[8] (2013) PhysX Dostopno na:
http://en.wikipedia.org/wiki/PhysX

[9] (2013) PhysX Features Dostopno na:
https://developer.nvidia.com/physx-features-2x

[10] (2013) Popular Physics Engines comparison: PhysX, Havok and ODE Dostopno na:
http://physxinfo.com/articles/?page_id=154

[11] (2013) Havok (software) Dostopno na:
http://en.wikipedia.org/wiki/Havok_(software)

[12] (2013) PhysX Downloads Dostopno na:
https://developer.nvidia.com/physx-downloads

[13] (2013) Open source physics engines Dostopno na:
http://www.ibm.com/developerworks/opensource/library/os-physicsengines/index.html

[14] (2013) Bullet (software) Dostopno na:
http://en.wikipedia.org/wiki/Bullet_(software)

[15] (2013) Collision detection Dostopno na:
http://en.wikipedia.org/wiki/Collision_detection

[16] (2013) Game engine Dostopno na:
http://en.wikipedia.org/wiki/Game_engine

[17] (2013) Game Development Team Composition Study - Changes over time. Dostopno na:
http://web.cs.wpi.edu/ id111x/c05/slides/intro.ppt

[18] (2013) Going Indie: The Story Of Independent Android Game Development From Concept To Completion Dostopno na:
http://www.androidpolice.com/2012/07/30/going-indie-the-story-of-independent-android-game-development-from-concept-to-completion/

[19] (2013) Unity - Shaders Dostopno na:
http://docs.unity3d.com/Documentation/Manual/Shaders.html